

Head First BLOGE

English Edition

A guided learning track for the BLOGE orchestration engine

BLOGE Project

Version 0.9.9-RC1

Contents

Learning Arc — Arc introduction 5

Arc 1 — Seeing Workflows

Graph shape, dependencies, and data flow

Goal: see responsibilities and dependencies before adding control or scale.

Chapter 1 — What BLOGE Is and Why It Exists . . . 6
Chapter 2 — Your First Graph . . . 15
Chapter 3 — Thinking in Dependencies . . . 23
Chapter 4 — Data That Flows . . . 31
Arc 1 Recap — Introductory (Chapters 1–4) . . . 41

Learning Arc — Arc introduction 43

Arc 2 — Maintainable Authoring

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

Chapter 5 — Branches That Decide . . . 44
Chapter 6 — Resilience by Design . . . 58
Chapter 7 — Designing Good Operators . . . 70
Chapter 8 — Turning a Personal DSL Draft into a Team Asset . . . 83
Chapter 9 — Tooling Workflow . . . 89
Arc 2 Recap — Maintainable Authoring (Chapters 5–9) . . . 101

Learning Arc — Arc introduction 103

Arc 3 — Composition and Time

Subgraphs, iteration, suspension, and crash recovery

Goal: compose work across collections, external time, and process restarts.

Chapter 10 — Reuse with Subgraphs . . . 104
Chapter 11 — Batch and Iteration . . . 114
Chapter 12 — Waiting for the World . . . 129
Chapter 13 — Durable Execution . . . 136
Arc 3 Recap — Composition and Time (Chapters 10–13) . . . 149

Learning Arc — Arc introduction 151

Arc 4 — Lifecycle Models

Sessions, state machines, nested ownership, and bounded agents

Goal: choose one explicit owner for every long-lived decision.

Chapter 14 — Multi-Turn Sessions . . . 152
Chapter 15 — State Machines . . . 163
Chapter 16 — Composing Sessions and State Machines . . . 173
Chapter 17 — Putting a Nondeterministic Agent Inside a Deterministic Lifecycle . . . 182
Arc 4 Recap — Lifecycle Models (Chapters 14–17) . . . 188

Learning Arc — Arc introduction 190

Arc 5 — Technical Confidence

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

Chapter 18 — Testing Your Graphs . . . 191
Chapter 19 — Observability in Production . . . 203
Chapter 20 — Spring and Production Wiring . . . 212
Chapter 21 — Scheduling and Complexity in One JVM . . . 222
Chapter 22 — Distributed Runtime and Capacity Validation . . . 227
Arc 5 Recap — Technical Confidence (Chapters 18–22) . . . 232

Learning Arc
Arc introduction 234

Arc 6 — Business Promises

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

Chapter 23 — Business Correctness Is Not Test Pass . . . 235
Chapter 24 — Your First Loan Business Verification . . . 240
Chapter 25 — Policy, Fixture, and Assurance . . . 247
Chapter 26 — Durable and Temporal Verification . . . 252
Chapter 27 — Contract Sensitivity: Can the Suite Reject a Wrong Answer? . . . 257
Chapter 28 — Bounded Property Verification and Useful Counterexamples . . . 261
Chapter 29 — From Report to Source-bound Evidence . . . 265
Chapter 30 — From Case Facts to an Exact Requirement Contribution . . . 270
Chapter 31 — Project, Reactor, and Claim Capability . . . 273
Chapter 32 — Change Attribution, Maven, and Release Decisions . . . 277
Arc 6 Recap — Business Evidence . . . 282

Learning Arc
Arc introduction 284

Arc 7 — Selection and Capstone

Constraint-first selection and an operable graduation project

Goal: use BLOGE only when justified, then deliver a result others can challenge.

Chapter 33 — Migration and Comparison . . . 285
Chapter 34 — Labs . . . 297
Arc 7 Recap — Graduation . . . 310
Appendix A — Built-in Functions . . . 311
Appendix B — Session Cheat Sheet . . . 319
Appendix C — Streaming . . . 321
Appendix D — State Machines . . . 326
Appendix E — Agent Extension . . . 332
Appendix F — Decision Table . . . 341
Appendix G — DSL Engineering Reference . . . 348
Appendix H — Durable and Scale Runtime Reference . . . 350
Appendix I — Spring Production Reference . . . 352
Appendix J — BLOGE Coding and Refactoring Handbook . . . 359

Foreword {-}

Head First BLOGE is a guided learning track for the BLOGE orchestration engine. It starts with graphs, nodes, dependencies, and data flow, then works toward durability, observability, and production wiring through real examples drawn from this repository.

BLOGE in one sentence: BLOGE is a Java-first orchestration engine: you describe business steps as a graph, and the engine takes care of ordering, parallelism, retry, timeout, branching, and recovery.

> *BLOGE ships as a small core plus opt-in extensions: `bloge-agent-ext` (LLM-driven agent orchestration with tool dispatch), `bloge-session-ext` + `bloge-session-durable` (crash-safe multi-turn sessions), and `bloge-state-ext` + `bloge-state-durable` (crash-safe state machines). These are covered in later chapters.*

Three core properties

- **Visible workflow** — the shape of the process is explicit instead of hidden across service methods.
- **Dependency-driven execution** — you declare who depends on whom; the engine decides what can run now.
- **Built-in resilience** — retry, timeout, fallback, waiting, and durable progress are part of the model, not hand-written glue code.

Who this book is for

- **Primary audience** — Java developers new to orchestration who want to understand how BLOGE models, executes, and recovers business workflows.
- **Secondary audience** — developers evaluating BLOGE against Temporal, Camunda, Spring Batch, or hand-written orchestration.

How to read this book

1. **Go in order.** Each chapter builds on the previous one.
2. **Run the examples.** The code comes from real files in this repository — open them, execute them, break them.
3. **Use the reference docs for exact syntax.** This series teaches *how to think*; the specs define *what is legal*.

The seven learning arcs

- **Arc 1 — Introductory**: build the mental model for graphs, nodes, dependencies, and flowing data.
- **Arc 2 — Maintainable authoring**: turn DSL changes into readable, testable, reviewable engineering work.
- **Arc 3 — Composition and time**: practice reuse, iteration, suspension, and crash recovery.
- **Arc 4 — Lifecycle models**: model sessions, state machines, nested ownership, and bounded agent loops.
- **Arc 5 — Technical confidence**: match tests, telemetry, Spring wiring, and capacity evidence to precise claims.
- **Arc 6 — Business promises**: verify owner-approved contracts with controlled inputs, source-bound evidence, governance, and attribution.
- **Arc 7 — Selection and capstone**: decide when BLOGE fits, then build the smallest justified graduation project.

Learning Arc

Arc 1 — Seeing Workflows

Graph shape, dependencies, and data flow

Goal: see responsibilities and dependencies before adding control or scale.

CHAPTER
01

Arc 1 — Seeing Workflows

Chapter 1 — What BLOGE Is and Why It Exists

Graph shape, dependencies, and data flow

Goal: see responsibilities and dependencies before adding control or scale.

This chapter establishes a working understanding of what BLOGE is, why it exists, *and* which structural properties define it most clearly.

1.1 Learning Goals

1. Describe BLOGE in one sentence without falling back to framework jargon.
2. Name the three properties that make BLOGE different from ad-hoc orchestration: **visible**, **structured**, and **resilient**.
3. Describe what a *graph* is at the conceptual level — a directed acyclic graph of business steps with explicit data and control flow.
4. Explain why the engine, not the developer, should be responsible for concurrency, retry, and failure propagation.

1.2 Prerequisites

- Basic Java knowledge (you don't need to be an expert).
- A working JDK 25+ and Maven 3.6+ installation (see Getting Started).

1.3 Source Examples

File	What it shows
`ch05/order-process.bloge`	A complete teaching workflow with fan-out, fan-in, branching, and resilience
`ch03/bff-dashboard.bloge`	A BFF aggregation graph with five parallel fetches and mixed fallback strategies

1.4 30-Second Introduction

At a high level, BLOGE is an orchestration engine for business workflows. It does not replace business code itself; instead, it extracts runtime concerns such as dependency relationships, parallel execution, and failure handling from scattered implementations and organizes them into an explicit graph.

1.4.1 Defining BLOGE

BLOGE is a Java-first orchestration engine. It represents business workflows as graphs of nodes and dependencies, while the engine handles ordering, parallelism, retry, timeout, branching, waiting, and recovery. Runtime logic that would otherwise be scattered across service code is therefore collected into an explicit structure.

BLOGE extends beyond DAG-based graphs into higher-level orchestration patterns: multi-turn sessions (`bloge-session-ext` + `bloge-session-durable`), explicit state machines (`bloge-state-ext` + `bloge-state-durable`), and LLM-driven agent loops with tool dispatch (`bloge-agent-ext`). These capabilities are covered in later chapters — for now, the graph model is the foundation.

`BLOGE` stands for **Biz Logic Orchestration Graph Engine**. The expanded name states the system boundary directly: it is concerned with *business logic*, it solves an *orchestration* problem, it uses a *graph* as the execution model, and it relies on an *engine* to run that model.

In that sense, BLOGE is neither only a DSL nor only a diagramming tool. It combines a workflow model with a runtime engine; once workflow shape, dependencies, and failure policies are declared, the engine applies them to execution, concurrency control, and resilience behavior.

1.4.2 Three Design Properties Used in This Book

This book uses three properties to compare BLOGE with ad-hoc orchestration. They describe how the workflow is read, how execution is organized, and how runtime behavior is attached to the model; they are an explanatory frame, not an independent benchmark result.

1. **Visible** — the workflow is presented directly as a graph, so business paths, branch points, and upstream-downstream relationships can be read from the model instead of reconstructed from service code.
2. **Structured** — dependencies belong to the workflow model itself. Which nodes may run in parallel and which nodes must wait is determined by declared graph relationships rather than by the order of handwritten code.
3. **Resilient** — retry, timeout, fallback, waiting, and durable progress are attached to the workflow as runtime policies. Failure handling therefore becomes part of the model instead of being scattered across ad-hoc glue code.

1.4.3 What BLOGE Is Not

Defining what BLOGE is not helps establish the boundary of the system just as clearly. The following interpretations are incomplete because they describe surface form without describing BLOGE as a workflow model plus runtime engine.

1. **It is not another wrapper around sequential code.** If the only change were syntactic, execution shape would still be determined mainly by writing order; in BLOGE, execution shape is determined by dependencies declared in the graph.
2. **It is not only a diagramming tool.** A graph here is not a static picture for documentation; it is a workflow model that the engine can read, schedule, and execute.
3. **It is not a system that leaves orchestration mechanics to each application team.** Concurrency, retry, timeout, fallback, and waiting do not have to be rebuilt around every business step; the engine applies them consistently from the model.

The simplest accurate mental shorthand is this: **BLOGE turns workflow intent into an explicit graph, then lets the engine carry the operational burden.**

1.5 Why Not Hand-Written Orchestration

Knowing what BLOGE is is only the first step. The harder question is why teams should stop writing orchestration as sequential service code with ad-hoc retry, timeout, and recovery logic spread across multiple classes. That is where hand-written orchestration starts to fail.

1.5.1 Why This Matters

Most business systems start simple:

```
1. Fetch the user
2. Fetch the products
3. Calculate the price
4. Check credit
5. Create the order (or reject it)
```

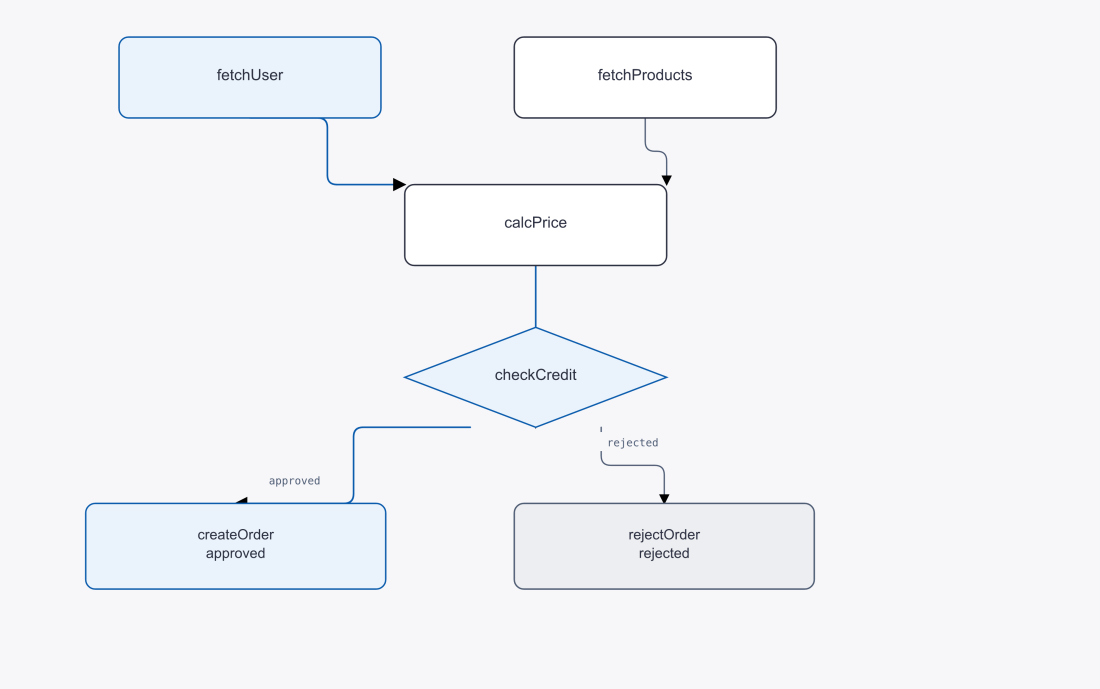
In a plain service, you write this as sequential method calls. It works — until it doesn't:

- **Steps 1 and 2 are independent** but you run them in sequence, wasting wall-clock time.
- **Step 4 can fail transiently**, so you wrap it in a retry loop — and now your business logic is tangled with infrastructure.
- **Nobody can *see* the workflow.** When something breaks at 2 AM, a developer has to read the code to reconstruct what happened.

This is the problem BLOGE exists to solve. Once you know that BLOGE is an engine for explicit workflow graphs, the value proposition becomes much clearer: you declare the *shape* of your workflow — which steps depend on which, what to do when something fails — and the engine handles scheduling, concurrency, and resilience for you.

1.5.2 Mental Model

Think of a BLOGE graph as a **recipe card** for the engine:



Each box is a **node**. Each arrow is a **dependency**. The engine reads this card and figures out:

- `fetchUser` and `fetchProducts` have no dependencies on each other → **run them in parallel**.
- `calcPrice` needs both → **wait for both, then start**.

- `checkCredit` feeds a **branch** — the engine picks the right downstream node based on the result.

You describe *what depends on what*. The engine decides *when to run what*.

1.5.3 First Working Example

Here is a simplified extract from `ch05/order-process.bloge`:

```
graph orderProcess {

  node fetchUser : FetchUserOperator {
    input { userId = ctx.userId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchProducts : FetchProductsOperator {
    input { productIds = ctx.productIds }
    timeout = 5s
  }

  node calcPrice : CalcPriceOperator {
    depends_on = [fetchUser, fetchProducts]
    input {
      user     = fetchUser.output
      products = fetchProducts.output
    }
  }

  node checkCredit : CreditCheckOperator {
    depends_on = [fetchUser, calcPrice]
    input {
      userId = fetchUser.output.id
      amount = calcPrice.output.total
    }
    retry   = { attempts: 3, backoff: 100ms, strategy: jitter }
    fallback = { approved: false, reason: "credit service unavailable" }
  }

  branch on checkCredit.output.approved {
    true  -> createOrder
    false -> rejectOrder
  }
}
```

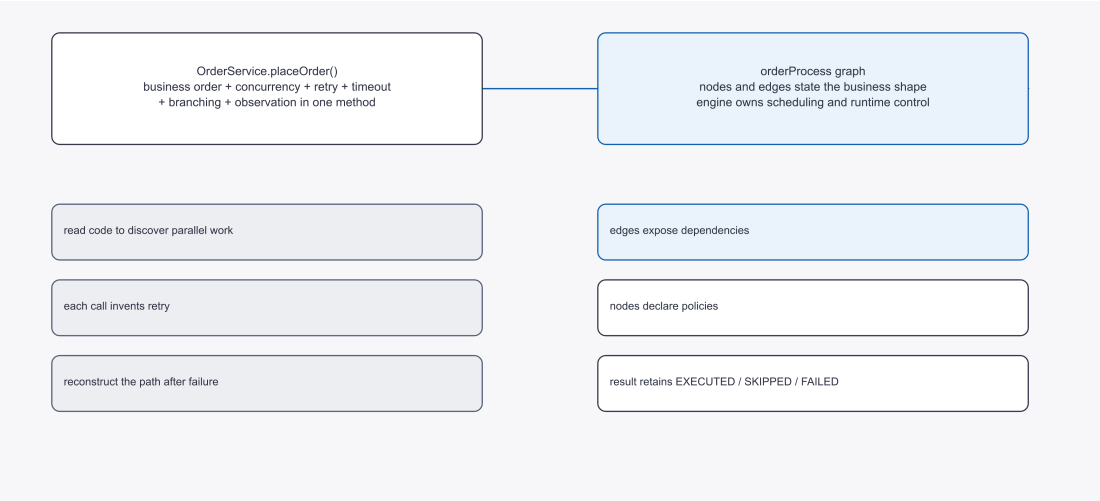
Read it top to bottom:

1. **Two independent fetches** — the engine will run them concurrently. `fetchUser` even has its own retry policy.
2. **A calculation node** with an explicit `depends_on` that fans in from both fetches — the engine could also infer this from the `.output` references.
3. **A credit check** (`CreditCheckOperator`) with retry and fallback — resilience declared right where the node is defined.
4. **A branch** — not an `if` statement in your Java code, but a first-class routing decision in the graph.

> ***Key insight:*** *You never wrote a thread pool, a retry loop, or a try/catch for the fallback. The engine owns that.*

1.6 Follow One Order: Move Responsibilities, Not Methods

Imagine `OrderService.placeOrder()` started as five readable calls. Six months later it also opens futures, retries credit checks, times out inventory, catches half a dozen exceptions, emits metrics, and decides whether to create or reject the order. The method still ”works”, but one place now owns both the business story and every mechanism required to run it.



1.6.1 Read the diagram as an ownership change

BLOGE does not improve the design merely by drawing boxes around those calls. The useful change is that each concern gets one visible owner:

Question	Visible owner
What work exists?	Nodes such as `fetchUser`, `calcPrice`, and `createOrder`
What must finish first?	Dependency edges
What happens after a transient failure?	The node's retry and timeout policy
Which business path was not selected?	The result status `SKIPPED`
What did the business operation actually do?	The Operator output and external evidence

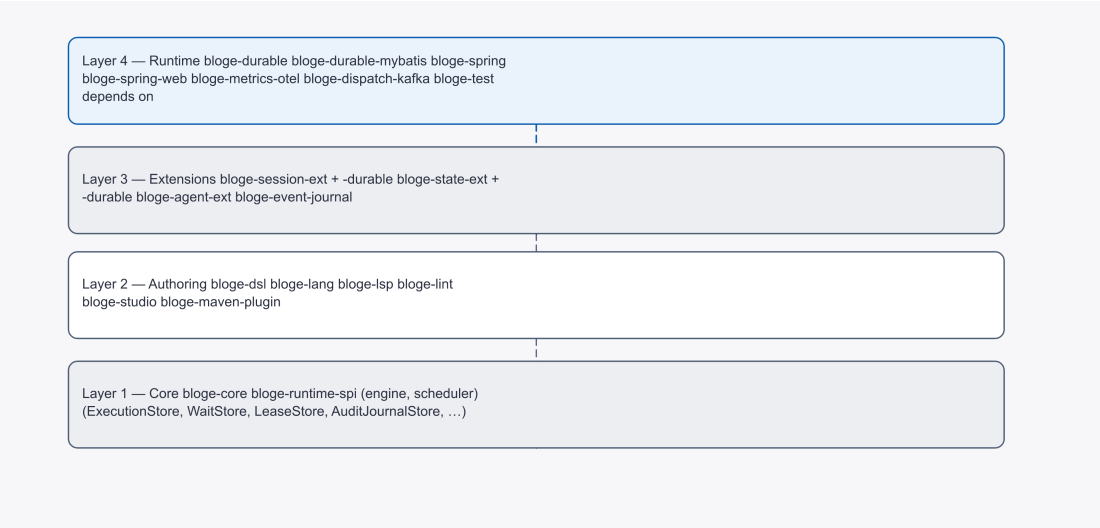
This separation matters during change. Adding an inventory check changes the graph shape. Changing the pricing formula changes `CalcPriceOperator`. Changing the retry delay changes a node policy. A reviewer no longer has to rediscover all three decisions inside one control-flow method.

1.6.2 The first boundary to remember

An explicit graph makes the **declared orchestration** inspectable; it does not prove that an Operator charged the right amount or that the workflow satisfies the business requirement. Chapters 18 and 23–32 add those stronger forms of evidence. For now, the win is narrower and concrete: the execution shape has stopped hiding inside incidental Java control flow.

1.6.3 The Module Map — Now You Have a Reason to Read It

The order story gives each layer a job. Core runs the graph; authoring tools make the declaration readable; extensions add larger interaction models; runtime integrations retain and observe execution.



Layer 1 — Core

Module	What it provides
`bloge-core`	Graph model, scheduler, listener/interceptor APIs, and `GraphEngine`.
`bloge-runtime-spi`	Execution, checkpoint, wait, timer, lease, audit, and inbox store contracts.

Layer 2 — Authoring

Module	What it provides
`bloge-dsl`	`.bloge` parser, compiler, and code generator.
`bloge-lang` / `bloge-lsp`	Shared language model and editor diagnostics.
`bloge-lint` / `bloge-studio`	Reproducible rules and visual graph authoring.
`bloge-maven-plugin`	Operator metadata generation.

Layer 3 — Extensions

Module	What it provides
`bloge-session-ext` / `bloge-state-ext`	Multi-turn sessions and explicit state machines.
`bloge-agent-ext`	Model/tool loops with bounded capabilities.
`bloge-event-journal`	Append-only execution and agent events.

Layer 4 — Runtime and integrations

Module	What it provides
`bloge-durable` / `bloge-durable-mybatis`	Recovery and persistent execution state.
`bloge-spring` / `bloge-spring-web`	Boot wiring, web errors, and the operations console.
`bloge-metrics-otel` / `bloge-dispatch-kafka`	Telemetry and remote-worker dispatch.
`bloge-test` / `bloge-verification`	Graph tests and business-correctness evidence.

Start with `bloge-core` and `bloge-dsl`. Add a layer only when the story reaches its problem; the rest of the book returns to this map at those moments.

1.7 Break It Apart

Concept	What it means
`graph`	A named container for the workflow — a DAG of nodes
`node`	A single processing step that runs an *operator*
`input { … }`	Bindings that wire upstream outputs and context values into the node's input
`ctx.userId`	A value from the *graph context* — the initial data passed to the engine at execution time
`fetchUser.output`	The output of an upstream node — referencing it creates an implicit dependency
`timeout`	A wall-clock duration limit for the node
`retry`	Automatic re-execution on failure with a backoff strategy
`fallback`	A static result to use if the node still fails after retries
`branch on`	Conditional routing that activates one downstream path based on a node's output

---

1.8 Common Trap

1.8.1 [X] Treating nodes like function calls

```
// DON'T think like this:
User user = fetchUser(ctx.getUserId());
List<Product> products = fetchProducts(ctx.getProductIds());
Price price = calcPrice(user, products);
```

In procedural code the *order of lines* is the execution order. In BLOGE the *dependency graph* is the execution order. If two nodes don't reference each other's output, the engine is free to run them at the same time.

Think in dependencies, not in sequences.

1.9 Guided Rewrite

Look at the `ch03/bff-dashboard.bloge` example. It fetches five data sources for a dashboard:

```
graph bffDashboard {

  node fetchProfile : FetchProfileOperator {
    input { userId = ctx.userId }
    timeout = 2s
    retry = { attempts: 1, backoff: 100ms }
  }

  node fetchOrders : FetchOrdersOperator {
    input { userId = ctx.userId }
    timeout = 3s
    fallback = { recentOrderIds: [], totalOrders: 0 }
  }

  // … fetchRecommendations, fetchNotifications, fetchLoyalty …

  node aggregate : AggregateOperator {
    depends_on = [fetchProfile, fetchOrders, fetchRecommendations, fetchNotifications, fetchLoyalty]
    input {
      profile         = fetchProfile.output
```

```
      orders          = fetchOrders.output
      recommendations = fetchRecommendations.output
      notifications   = fetchNotifications.output
      loyalty         = fetchLoyalty.output
    }
  }
}
```

Questions to consider:

1. How many nodes run in parallel? *(Answer: all five fetch nodes — none depends on another.)*
2. What happens if `fetchOrders` fails or times out? *(Answer: `fetchOrders` has no retry policy — it only has a fallback. If it fails the fallback value `{ recentOrderIds: [], totalOrders: 0 }` is used and the graph continues with partial data.)*
3. If you added a sixth fetch that depended on `fetchProfile.output`, would it still run in parallel with the other fetches? *(Answer: no — it would wait for `fetchProfile` to complete first.)*

1.10 Brain Check

1. What are the three properties that distinguish a BLOGE graph from ad-hoc orchestration code? *(Visible, structured, resilient.)*
2. Who decides whether two nodes run concurrently — the developer or the engine? *(The engine, based on the dependency graph.)*
3. In the order-process example, what happens if `checkCredit` fails after all three retry attempts? *(The fallback value is used: `{ approved: false, reason: "credit service unavailable" }`.)*

1.11 Lab

Open `ch05/order-process.bloge` in your editor.

1. **Add a `fetchInventory` node** that runs in parallel with `fetchUser` and `fetchProducts`. Give it a `timeout` of `2s`.
2. **Wire its output into `calcPrice`** by adding an input binding.
3. **Predict the new execution shape**: which nodes now run in parallel? Which wait?

Compilation can wait here. The immediate goal is to make the shape of the DSL readable before moving on to execution in Chapter 2.

1.12 Experiment acceptance card

- **Expected and observed:** Dependencies and resilience policies are visible in DSL, and the execution shape can be labelled.
- **Failure and recovery:** Reference a missing output; remove the bad edge and redraw the DAG.
- **Proof boundary:** Proves reviewable design intent, not compilation, execution, or business correctness.
- **Exercise contract:** Order DSL; add only the inventory node and one binding; deliver a dependency sketch; stop when every edge has a source and the graph is acyclic.

1.13 Recap

- BLOGE replaces hand-written orchestration with a **visible, structured, resilient** graph.
- A **graph** is a DAG of **nodes**. Each node runs an **operator**.
- The engine reads **dependencies** (explicit or inferred from `.output` references) and schedules nodes accordingly.
- **Resilience** (timeout, retry, fallback) is declared on the node, not buried in business code.
- **Branches** route control flow based on a node's output.

1.14 Next Step

Chapter 2 — Your First Graph continues with the smallest graph that can be written, compiled, and executed end to end.

1.15 Reference Links

- Project README — module overview and quick start
- Getting Started — full setup guide
- Core Architecture — engine internals
- `ch05/order-process.bloge` — full example source
- `ch03/bff-dashboard.bloge` — full example source

CHAPTER
02

Arc 1 — Seeing Workflows

Chapter 2 — Your First Graph

Graph shape, dependencies, and data flow

Goal: see responsibilities and dependencies before adding control or scale.

This chapter walks through the smallest runnable BLOGE graph, including how it is written, compiled, and executed in both DSL form and the fluent Java API.

2.1 Learning Goals

1. Write a minimal `.bloge` file that the compiler accepts.
2. Load and execute that file from Java.
3. Build the same graph using the fluent Java API.
4. Understand what `ctx`, `input`, `output`, and `timeout` mean at runtime.

2.2 Prerequisites

- Chapter 1 — What BLOGE Is and Why It Exists
- JDK 25+ with `--enable-preview`, Maven 3.6+ (see Getting Started)

2.3 Source Examples

File	What it shows
`ch02/hello-world.bloge`	The absolute minimum: one graph, one node, one input binding
`ch02/two-node-chain.bloge`	Two nodes with an explicit dependency
`input-bindings.bloge`	Conformance fixture: basic input binding syntax

2.4 Why This Matters

Chapter 1 established BLOGE's definition, core properties, and the limits of hand-written orchestration. This chapter carries those ideas into the smallest runnable example so that graph authoring, loading, and execution can be seen in concrete form.

A one-node graph is already enough to place four runtime concepts on a single execution path:

- How data enters the graph (the *context*).
- How a node receives data (the *input block*).
- How the engine invokes your code (the *operator*).
- How to read the result.

Everything else in this book builds on these four ideas.

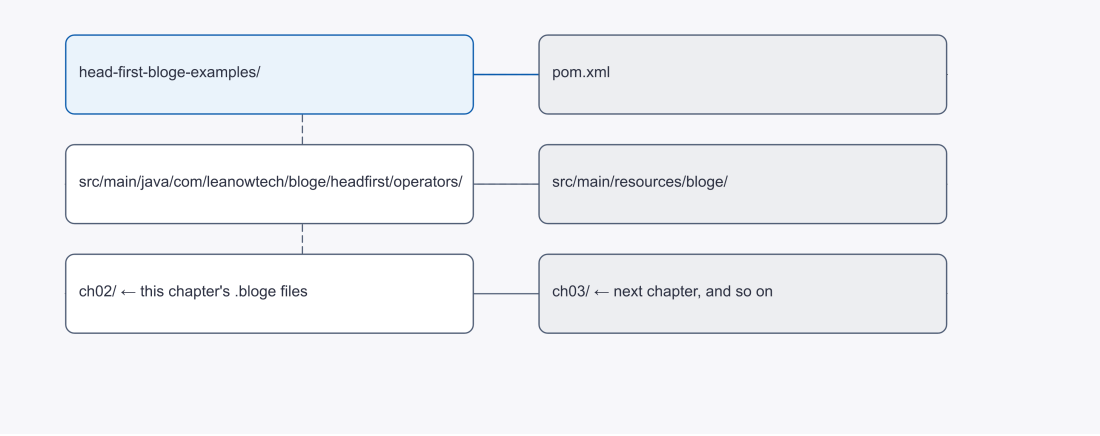
2.5 Setup

If you want to follow this chapter hands-on, use the companion project that now lives beside the book in this repository.

1. Install the core BLOGE artifacts that the standalone companion module depends on:
   ```
   mvn -q -pl bloge-core,bloge-dsl,bloge-test -am install -DskipTests -Dspotbugs.skip=true
   ```
2. Run the companion example tests:
   ```
   mvn -f head-first-bloge-examples/pom.xml test
   ```
3. Open the Chapter 2 companion files:
   - `head-first-bloge-examples/src/main/resources/bloge/ch02/hello-world.bloge`
   - `head-first-bloge-examples/src/main/resources/bloge/ch02/two-node-chain.bloge`
   - `head-first-bloge-examples/README.md`

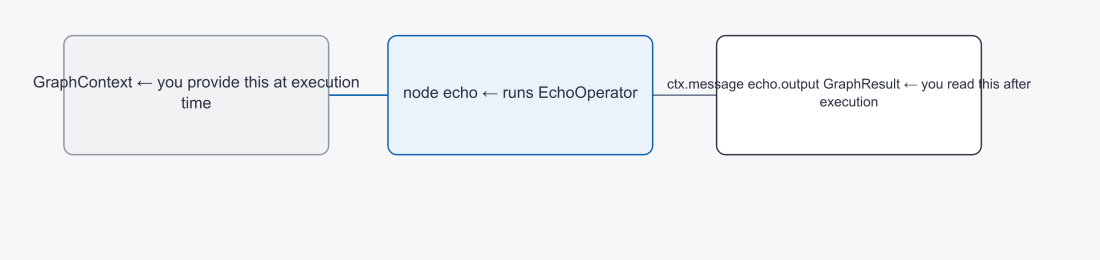
The point is not only to read the snippets in this chapter. Run them, inspect the tests, and keep the book open next to the companion project so the example stays grounded in something executable.

2.5.1 Project layout



Each chapter's `.bloge` files live in `src/main/resources/bloge/chNN/`. The test classes under `src/test/java/` compile or execute those files. As you progress through the book, run the matching chapter examples so your understanding stays tied to something real.

2.6 Mental Model



1. You create a **GraphContext** and put values in it.
2. The engine resolves **input bindings** — expressions like `ctx.message` — to pull data from the context (or from upstream node outputs).
3. Each node's **operator** receives the assembled input, does its work, and returns an output.
4. After execution, you read results from the **GraphResult**.

2.7 First Working Example

2.7.1 The DSL way

This is the content of `ch02/hello-world.bloge`:

```
graph helloWorld {

  node echo : EchoOperator {
    input {
      message = ctx.message
    }
    timeout = 1s
  }
}
```

That is a complete, valid BLOGE graph. The table below breaks it down:

Token	Meaning
`graph helloWorld`	Declares a graph named `helloWorld`
`node echo`	Declares a node named `echo`
`: EchoOperator`	Binds the node to the operator class `EchoOperator`
`input { message = ctx.message }`	The node's input field `message` is bound to the context value `message`
`timeout = 1s`	The node must complete within 1 second

To load and run this from Java:

```
// 1. Register your operator
OperatorRegistry registry = new DefaultOperatorRegistry();
registry.register("EchoOperator", (input, opCtx) -> input);

// 2. Load the DSL file
GraphLoader loader = new GraphLoader(registry);
Graph graph = loader.load(Path.of("src/main/resources/bloge/hello-world.bloge"));

// 3. Build the context
```

```
GraphContext ctx = new GraphContext();
ctx.put("message", "hello");

// 4. Execute
GraphEngine engine = GraphEngine.builder().registry(registry).build();
GraphResult result = engine.execute(graph, ctx);
```

2.7.2 The Java API way

The same graph, built with the fluent Java API:

```
Graph graph = Graph.builder("helloWorld")
    .node("echo", (Map<String, Object> input, OperatorContext opCtx) ->
        Map.of("message", opCtx.graphContext().get("message", String.class)))
        .timeout(Duration.ofSeconds(1))
    .build();

GraphContext ctx = new GraphContext();
ctx.put("message", "hello");

GraphEngine engine = GraphEngine.builder().build();
GraphResult result = engine.execute(graph, ctx);
```

Both approaches produce the same runtime graph. The DSL is declarative and external; the Java API is type-safe and in-code. Most teams start with DSL for prototyping and move stable operator logic into Java classes.

2.8 Read Three Outcomes Before Adding Configuration

A first graph is not complete when it merely compiles. You should be able to tell, from a small result, whether execution succeeded and where two common failures stopped. That gives every later option a purpose instead of turning the Builder API into a catalogue to memorise.



2.8.1 Outcome 1 — a value came back

The RC1 foundation probe runs `hello-world` with `message=hello` and checks the result rather than relying on a lack of exceptions:

```
graph: hello-world    status: SUCCESS
node:  echo           status: EXECUTED
output.message:       hello
```

Three facts are separate: the graph completed, `echo` executed, and its output has the expected value. Keep all three

when the graph grows.

2.8.2 Outcomes 2 and 3 — execution never began

Stage	Observable result	Repair the smallest owner
Compile	A node has no Operator binding; diagnostic points at the declaration	Fix the `.bloge` node declaration
Load	`MissingOperator` is not in the registry	Register the capability or correct its name
Execute	A bound Operator returns or throws	Inspect node status, output, and failure evidence

Compile and load failures should show `operator calls = 0`. Retrying the graph cannot repair either one because the business code was never dispatched.

Only now do Builder options become useful: listeners observe these stages, interceptors wrap actual invocations, concurrency changes scheduling, and a checkpoint store changes retained execution state. Start from an observed problem, then select the option that owns it.

2.9 Break It Apart

2.9.1 Adding a second node

Here is `ch02/two-node-chain.bloge`:

```
graph twoNodeChain {

  node normalizeName : NormalizeNameOperator {
    input { name = ctx.name }
    timeout = 1s
  }

  node buildGreeting : BuildGreetingOperator {
    depends_on = [normalizeName]
    input { value = normalizeName.output.value }
    timeout = 1s
  }
}
```

New concepts:

Concept	What it means
`depends_on = [normalizeName]`	Explicit dependency — `buildGreeting` waits for `normalizeName`
`normalizeName.output.value`	Read the `value` field from the upstream node's output

Note that even without `depends_on`, the engine would infer the dependency from the `normalizeName.output.value` reference. The explicit declaration is optional but makes intent clear.

2.9.2 Conformance baseline

The conformance suite includes a minimal input-binding fixture (`input-bindings.bloge`):

```
graph g {
  node a : Op {
    input {
```

```
      x = ctx.foo
      y = "bar"
    }
  }
}
```

This shows that input bindings can be context references (`ctx.foo`) *or* literal values (`"bar"`).

2.9.3 Add configuration only when an observed problem owns it

Do not copy every Builder option into the first graph. Keep this routing table and return when the story reaches the corresponding problem:

Observed need	Configuration owner and later chapter
Reject an incompatible input/output shape	`Graph.builder().schemaValidation(...)` and `.contracts(...)` — Chapter 8
Observe or wrap each invocation	`GraphEngine.builder().listeners(...)` and `.interceptors(...)` — Chapter 19
Bound simultaneous work	`.maxGlobalConcurrency(...)` — Chapter 21
Retain or migrate execution state	`.executionCheckpointStore(...)`, `.checkpointCodec(...)`, `.versionMismatchPolicy(...)` — Chapter 13
Resolve tenant scope and admission	`.tenantContextResolver(...)`, `.tenantResourcePolicy(...)` — Chapter 21
Replace scheduling time	`.schedulerTimerSupport(...)` — Chapters 11–12

`Graph.builder()` describes one graph; `GraphEngine.builder()` configures the runtime that can execute many graphs. If you cannot name the observed problem, keep the default.

2.10 Common Trap

2.10.1 [X] Forgetting the operator binding

```
// WRONG — the compiler requires an operator type
node echo {
  input { message = ctx.message }
}
```

Every node must declare an operator with the `: OperatorName` syntax. Without it, the compiler does not know which code to run.

```
// CORRECT
node echo : EchoOperator {
  input { message = ctx.message }
}
```

2.11 What Goes Wrong

2.11.1 Missing operator binding — the compiler error

If you forget the operator type:

```
node echo {
```

```
  input { message = ctx.message }
}
```

the parser stops immediately. The failure message comes from the grammar itself:

```
GraphDefinitionException: Expected ':' after node id
```

If you keep the colon but omit the operator name, the parser follows up with `Expected operator reference`.

Fix: Always declare the operator with `: OperatorName`.

2.11.2 Unregistered operator — the load-time error

If the DSL names an operator that does not exist in the registry:

```
registry.register("EchoOperator", (input, opCtx) -> input);
Graph graph = loader.load(Path.of("hello-world.bloge"));
// But the DSL references "ShoutOperator" which was never registered
```

loading fails before execution begins:

```
GraphDefinitionException: Node 'shout' references unregistered operator 'ShoutOperator'
```

Fix: Register every operator name that appears in your `.bloge` file before loading or executing the graph.

2.12 Guided Rewrite

Start from the hello-world graph and extend it:

1. **Add a second node** called `shout` that converts the echo output to uppercase. Bind its input to `echo.output.message`.
2. **Give it a timeout** of `2s`.
3. **Predict the execution order.** Which node runs first? Why?

Your DSL should look something like:

```
graph helloWorld {

  node echo : EchoOperator {
    input { message = ctx.message }
    timeout = 1s
  }

  node shout : ShoutOperator {
    input { message = echo.output.message }
    timeout = 2s
  }
}
```

Because `shout` references `echo.output`, the engine infers the dependency and runs `echo` first.

2.13 Brain Check

1. What are the four things you need to execute a graph? *(A Graph, an OperatorRegistry, a GraphContext, and a GraphEngine.)*
2. If a node's input binding references `ctx.userId`, where does that value come from? *(From the GraphContext you create before execution.)*
3. Can you mix literal values and upstream references in the same input block? *(Yes — see the conformance fixture where `x = ctx.foo` and `y = "bar"` coexist.)*
4. You are sketching a workflow in a workshop and want the fastest edit-run feedback loop. Would you start with

the DSL file or the fluent Java API, and what would make you switch later? *(Most teams start with DSL because the graph shape is easier to see and edit. They switch to Java API when they need stronger in-code reuse, tighter typing, or deeper application-level composition.)*

5. *Design question:* If a graph has ten nodes but only two of them need data from the context, does every node need an `input` block? *(No — only nodes that consume external data or upstream outputs need `input`. A node with no input block receives an empty input map.)*

2.14 Lab

1. Create a new file `my-first-graph.bloge` with a `graph greet` containing:
   - A `fetchName` node that reads `ctx.name`.
   - A `buildGreeting` node that reads `fetchName.output.name` and produces a greeting string.
2. In Java, register both operators, load the file, execute the graph with `ctx.put("name", "Alice")`, and print the result.
3. Verify that `buildGreeting` ran *after* `fetchName` by checking the `GraphResult`.

Refer to Getting Started for project setup details.

2.15 Experiment acceptance card

- **Expected and observed:** The minimal graph returns SUCCESS, marks nodes EXECUTED, and exposes the final output.
- **Failure and recovery:** Misspell one operator name; restore it after the compile/load error and rerun.
- **Proof boundary:** Proves minimal wiring and result reading, not retries, durability, or business correctness.
- **Exercise contract:** Hello graph; change one operator identifier; deliver success and failure receipts; stop when both repeat.

2.16 Recap

- A minimal BLOGE graph needs a `graph` block, at least one `node` with an operator binding, and an `input` block.
- **GraphContext** carries the initial data; **input bindings** wire it into nodes.
- The **fluent Java API** and the **`.bloge` DSL** produce the same runtime graph.
- Dependencies can be **explicit** (`depends_on`) or **inferred** from `.output` references.

2.17 Next Step

Chapter 3 — Thinking in Dependencies extends the discussion into dependency-based scheduling and why good graph design starts from dependencies rather than line-by-line sequence.

2.18 Reference Links

- Getting Started — full setup guide
- DSL Specification — complete grammar
- `ch02/hello-world.bloge` — companion source
- `ch02/two-node-chain.bloge` — companion source
- `head-first-bloge-examples/README.md` — companion project usage
- `input-bindings.bloge` — conformance fixture

CHAPTER
03

Arc 1 — Seeing Workflows

Chapter 3 — Thinking in Dependencies

Graph shape, dependencies, and data flow

Goal: see responsibilities and dependencies before adding control or scale.

Promise: *By the end of this chapter you will know how to read a graph as a dependency structure, predict its execution order, and stop thinking in sequential steps.*

3.1 Learning Goals

1. Distinguish *implicit* dependencies (inferred from `.output` references) from *explicit* dependencies (`depends_on`).
2. Predict which nodes run in parallel and which wait, given a graph definition.
3. Understand how the engine's completion-driven scheduler turns a DAG into an execution plan.
4. Recognise when an unnecessary dependency kills parallelism.

3.2 Prerequisites

- Chapter 2 — Your First Graph

3.3 Source Examples

File	What it shows
`ch03/order-process.bloge`	Fan-out, fan-in, and branching in a realistic workflow
`ch03/bff-dashboard.bloge`	Five-way parallel fan-out with mixed resilience
`depends-timeout.bloge`	Explicit dependency + timeout

3.4 Why This Matters

In Chapter 2 you ran a graph with one or two nodes. That was enough to learn the mechanics. But real workflows have **many nodes with complex dependency relationships** — and the execution order is no longer obvious from reading

top to bottom.

If you don't understand how the engine resolves dependencies, you will:

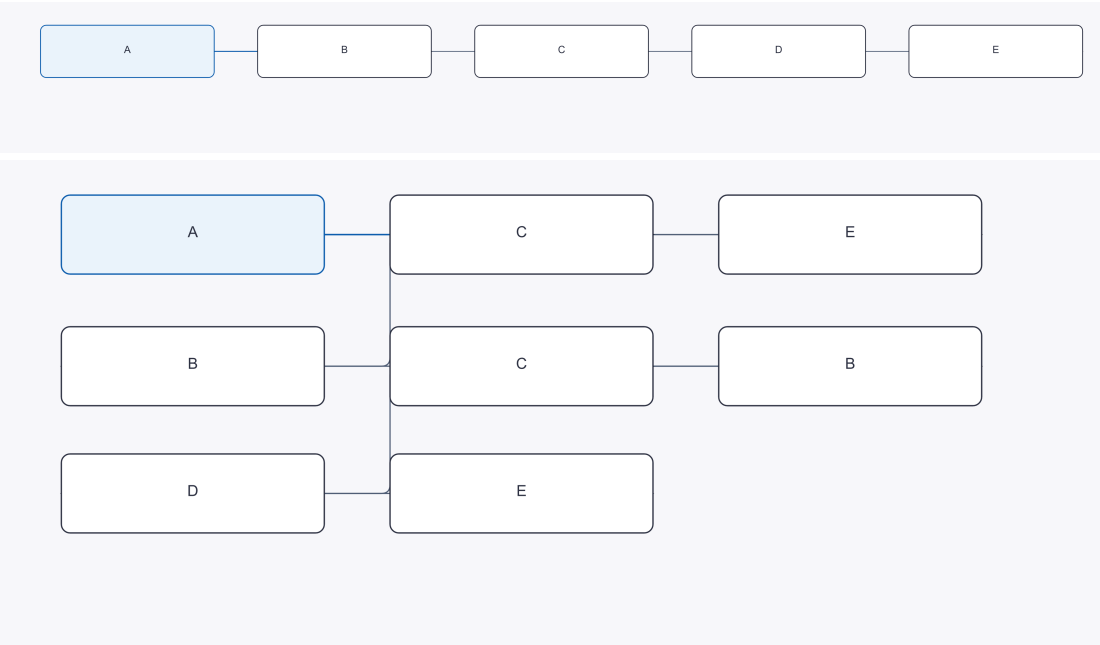
- Accidentally **serialise** nodes that could run in parallel.
- Introduce **circular dependencies** that the compiler rejects.
- Struggle to **debug timing** because you expect a sequence that the engine never promised.

The shift in thinking is: **you declare *what depends on what*; the engine decides *when to run what*.**

3.5 Mental Model

3.5.1 The DAG as a promise

A BLOGE graph is a **directed acyclic graph** (DAG). Each node is a vertex; each dependency is a directed edge. (The diagram below shows the same structure as a rendered graph.)



Reading this DAG:

- **A** and **B** have no incoming edges → they are *source nodes* → the engine starts them immediately, in parallel.
- **C** depends on both A and B → it waits for *both* to complete.
- **D** depends on B only → it starts as soon as B finishes (even if A is still running).
- **E** depends on C and D → it waits for *both*.

This is the **completion-driven** scheduling model. When a node completes, the engine checks whether any downstream node now has all its dependencies satisfied. If yes, that node is dispatched.

3.5.2 Implicit vs. explicit dependencies

There are two ways to create an edge:

Style	Syntax	When to use
Implicit	Reference `someNode.output` in an `input` binding	Most of the time — the compiler infers the edge automatically
Explicit	`depends_on = [someNode]`	When you need ordering but don't consume the output

Both produce the same runtime edge. Implicit is more common because you usually *do* need the upstream data.

3.6 First Working Example

Look at the dependency structure of `ch03/order-process.bloge`:

```
graph orderProcess {

  node fetchUser : FetchUserOperator {
    input { userId = ctx.userId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchProducts : FetchProductsOperator {
    input { productIds = ctx.productIds }
    timeout = 5s
  }

  node calcPrice : CalcPriceOperator {
    depends_on = [fetchUser, fetchProducts]
    input {
      user     = fetchUser.output
      products = fetchProducts.output
    }
  }

  // ...
}
```

Dependency analysis:

Node	Depends on	Why
`fetchUser`	*(nothing)*	Only reads `ctx` — a source node
`fetchProducts`	*(nothing)*	Only reads `ctx` — a source node
`calcPrice`	`fetchUser`, `fetchProducts`	Explicit `depends_on` plus `.output` references

Execution timeline:



`fetchUser` and `fetchProducts` start at the same moment. `calcPrice` starts only after *both* finish. The graph declaration expresses this dependency shape, so business code does not need to assemble a thread pool or `CompletableFuture.allOf()`. Runtime executor capacity remains an application configuration concern.

The same dependency shape built with the fluent Java API. This excerpt mirrors the real builder style used in `bloge-examples/.../OrderProcessingExample.java`:

```
var builder = Graph.builder("orderProcess")
    .node("fetchUser", FETCH_USER)
        .input((results, ctx) -> new UserQuery(ctx.get("userId", String.class)))
```

```
        .timeout(Duration.ofSeconds(3))
        .retry(2, Duration.ofMillis(200), BackoffStrategy.EXPONENTIAL)
    .node("fetchProducts", FETCH_PRODUCTS)
        .input((results, ctx) -> new ProductQuery(ctx.get("productIds", List.class)))
        .timeout(Duration.ofSeconds(5))
    .node("calcPrice", CALC_PRICE)
        .dependsOn("fetchUser", "fetchProducts")
        .input((results, ctx) -> new PriceInput(
                results.get("fetchUser", User.class),
                results.get("fetchProducts", ProductList.class)));

Graph graph = builder.build();
```

The DSL version and the Java version produce the same runtime graph. The DSL is typically preferred for authoring workflows; the Java API is used for programmatic graph construction and testing.

3.7 Watch the Ready Set Move in Four Frames

”A and B run in parallel” is easy to repeat and surprisingly easy to misunderstand. The scheduler does not move a cursor from the first line to the last. It maintains a set of nodes whose dependencies are satisfied, then reacts to completion events.



3.7.1 What the timestamps prove

The times in the diagram are a rounded, normalised trace; their exact values are not a performance promise. The RC1 probe asserts the ordering relationships:

Observation	Required inequality
Both sources are eligible together	`start(fetchUser)` and `start(fetchProducts)` occur before either is released
One completed source is insufficient	`start(calcPrice)` remains unset while the other source is blocked
Fan-in starts after both	`start(calcPrice) > start(fetchUser)` and `start(calcPrice) > start(fetchProducts)`

At 42 ms, `fetchUser` is finished but `calcPrice` still has one unresolved dependency. At 67 ms, the second completion decrements that count to zero and adds `calcPrice` to the ready set. This is the technical mechanism behind the visual fan-in.

3.7.2 Change one edge, predict one consequence

Remove the dependency on `fetchProducts` and `calcPrice` may start at 42 ms, before product data exists. Add an unnecessary edge from `fetchUser` to `fetchProducts` and the two source calls become serial. A dependency is

therefore not a drawing preference: it is an executable claim about data and ordering.

3.8 Break It Apart

3.8.1 Fan-out

When multiple source nodes read only from `ctx`, the engine starts them all at once. This is **fan-out**:

```
// Five independent fetches — all run in parallel
node fetchProfile       : FetchProfileOperator       { input { userId = ctx.userId } }
node fetchOrders        : FetchOrdersOperator        { input { userId = ctx.userId } }
node fetchRecommendations : FetchRecommendationsOperator { input { userId = ctx.userId } }
node fetchNotifications : FetchNotificationsOperator { input { userId = ctx.userId } }
node fetchLoyalty       : FetchLoyaltyOperator       { input { userId = ctx.userId } }
```

(From `ch03/bff-dashboard.bloge`.)

3.8.2 Fan-in

When a node references multiple upstream outputs, it **fans in** — it waits for all of them:

```
node aggregate : AggregateOperator {
  depends_on = [fetchProfile, fetchOrders, fetchRecommendations, fetchNotifications, fetchLoyalty]
  input {
    profile         = fetchProfile.output
    orders          = fetchOrders.output
    recommendations = fetchRecommendations.output
    notifications   = fetchNotifications.output
    loyalty         = fetchLoyalty.output
  }
}
```

3.8.3 Explicit dependency without data

Sometimes you need ordering without consuming the output. For example, you want node B to run after node A completes, even though B doesn't use A's data:

```
node a : OpA {}
node b : OpB {
  depends_on = [a]
}
```

(From `depends-timeout.bloge`.)

The explicit `depends_on` creates the same kind of edge as an implicit `.output` reference.

3.9 Common Trap

3.9.1 [X] Accidental serialisation

```
node fetchUser : FetchUserOperator {
  input { userId = ctx.userId }
}

node fetchProducts : FetchProductsOperator {
  // WRONG — this creates an unnecessary dependency on fetchUser
```

```
  depends_on = [fetchUser]
  input { productIds = ctx.productIds }
}
```

Because `fetchProducts` only needs `ctx.productIds`, the `depends_on` is unnecessary. It forces `fetchProducts` to wait for `fetchUser`, killing parallelism.

Rule of thumb: Only declare a dependency when you genuinely need the upstream node to finish first — either because you need its data or because of a side-effect ordering requirement.

3.10 What Goes Wrong

3.10.1 Circular dependency — the compiler rejection

If you accidentally create a cycle:

```
node a : OpA {
  input { x = b.output.value }
}
node b : OpB {
  input { y = a.output.value }
}
```

the compiler rejects the graph with a `GraphDefinitionException`. The exact message includes `cycle`, because the dependency graph can no longer be topologically ordered:

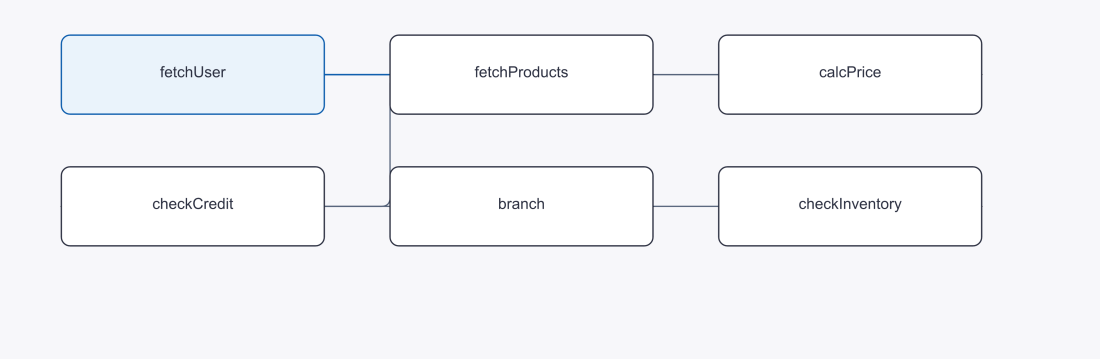
```
GraphDefinitionException: ... cycle ...
```

Fix: Redesign so data flows in one direction. If two nodes genuinely need each other's output, introduce an intermediate node or rethink the decomposition.

3.11 Guided Rewrite

Take the order-process graph and imagine a new requirement: before creating the order, you must check inventory *and* credit, independently.

Sketch the dependencies:



Questions:

1. Can `checkInventory` run in parallel with `checkCredit`? *(Yes, if `checkInventory` depends on `calcPrice` or earlier, but not on `checkCredit`.)*
2. What should the branch fan-in from? *(Both `checkCredit` and `checkInventory`, so you'd reference both outputs.)*
3. What is the minimum wall-clock time if all nodes take 1 second? *(3 seconds: one for the fetches in parallel, one for calcPrice, one for the checks in parallel, then the branch is essentially instantaneous.)*

3.12 Brain Check

1. What is a *source node*? *(A node with no incoming dependencies — it reads only from `ctx`.)*
2. If node C references `A.output` and `B.output` in its input block, what is the minimum set of dependencies the engine infers? *(C depends on A and C depends on B.)*
3. Why does the engine use a DAG and not allow cycles? *(A cycle would mean a node waits for itself — the graph could never complete. The compiler detects cycles using Kahn's algorithm and rejects them.)*
4. If you add `depends_on = [fetchUser]` to a node that only reads `ctx`, what happens? *(The node is forced to wait for `fetchUser` even though it doesn't need the data — accidental serialisation.)*
5. *Design question:* You have a graph where nodes A and B are independent source nodes, and node C depends on both. A new requirement says C should also use data from the `ctx`. Should you add `depends_on = [A, B]` explicitly, or is the implicit dependency from `A.output` and `B.output` enough? *(The implicit dependency from `.output` references is sufficient — adding explicit `depends_on` would be redundant. Only use explicit dependencies when you need ordering without consuming the output.)*

3.13 Lab

1. Open `ch03/bff-dashboard.bloge`.
2. Draw the dependency graph on paper (or in ASCII). Mark source nodes with `[S]` and the terminal node with `[T]`.
3. Predict the execution timeline. If each fetch takes 200 ms and the aggregate takes 50 ms, what is the total wall-clock time? *(≈ 250 ms — all fetches run in parallel, then aggregate runs.)*
4. Now add a `fetchWishlist` node that depends on `fetchProfile.output.userId`. Re-draw the graph. What is the new wall-clock time? *(≈ 450 ms — fetchWishlist must wait for fetchProfile, then aggregate waits for all six.)*

3.14 Experiment acceptance card

- **Expected and observed:** Two sources enter the ready set together; fan-in starts after both complete.
- **Failure and recovery:** Add a dependency between sources; remove it to restore parallelism.
- **Proof boundary:** Proves dependency-driven scheduling, not external latency or thread safety.
- **Exercise contract:** Pricing graph; change one depends_on edge; deliver before/after timelines; stop when starts match the prediction.

3.15 Coding practice

3.15.1 BP-08 — Retain real dependencies and explicit data

Pass business facts through input/output rather than shared mutable context. Remove accidental serialization but retain required control ordering, even without a field reference.

Try one change: Let credit and income checks start independently, then join their facts. Add a charge that must follow approval and explain that control prerequisite.

Reasoning, counterexamples, and limits

3.16 Recap

- A BLOGE graph is a **DAG**. Nodes are vertices; dependencies are directed edges.
- Dependencies can be **implicit** (from `.output` references) or **explicit** (`depends_on`).
- The engine uses a **completion-driven scheduler**: when a node finishes, it checks which downstream nodes are now ready.

- **Fan-out** happens when multiple source nodes have no dependencies on each other.
- **Fan-in** happens when a node waits for multiple upstream nodes.
- Unnecessary dependencies **kill parallelism** — only declare them when you genuinely need ordering.

3.17 Next Step

In Chapter 4 — Data That Flows you will learn how data moves through the graph — input bindings, output references, path expressions, transforms, and when to use an operator vs. a projection without Operator scheduling projection.

3.18 Reference Links

- Core Architecture — engine scheduling design
- DSL Specification — `depends_on` and input binding grammar
- `ch03/order-process.bloge` — source
- `ch03/bff-dashboard.bloge` — source
- `depends-timeout.bloge` — conformance fixture

CHAPTER
04

Arc 1 — Seeing Workflows

Chapter 4 — Data That Flows

Graph shape, dependencies, and data flow

Goal: see responsibilities and dependencies before adding control or scale.

Promise: *By the end of this chapter you will understand how data enters, moves through, and exits a BLOGE graph — and you will know when to use an input binding, a transform, or a full operator.*

4.1 Learning Goals

1. Explain the three data sources available inside an input block: **context values**, **upstream outputs**, and **literal values**.
2. Use **path expressions** to reach nested fields, including safe navigation (`?.`) and null coalescing (`??`).
3. Reach into lists and maps with **index access** (`xs[i]`, `xs?[i]`, `xs[-1]`, `m["key"]`) and use **dynamic-member access** on expression results (`(expr).field`, `(expr)?.field`).
4. Distinguish a **transform** (projection without Operator scheduling) from an **operator node** (business logic with scheduling overhead).
5. Use **lambda expressions** in input bindings for collection operations (`map`, `filter`, `reduce`).

4.2 Prerequisites

- Chapter 3 — Thinking in Dependencies

4.3 Source Examples

File	What it shows
`order-enrichment-lambda.bloge`	Lambda collection operations in input bindings
`order-summary-index-access.bloge`	Index, negative-index, and dynamic-member access
`basic.bloge` (transform)	A minimal transform block
`safe-navigation.bloge`	Safe navigation (`?.`) and null coalescing (`??`)
`lambda-map.bloge`	Lambda `map` in an input binding
`ch03/order-process.bloge`	Transform block in a realistic workflow

File	What it shows

4.4 Why This Matters

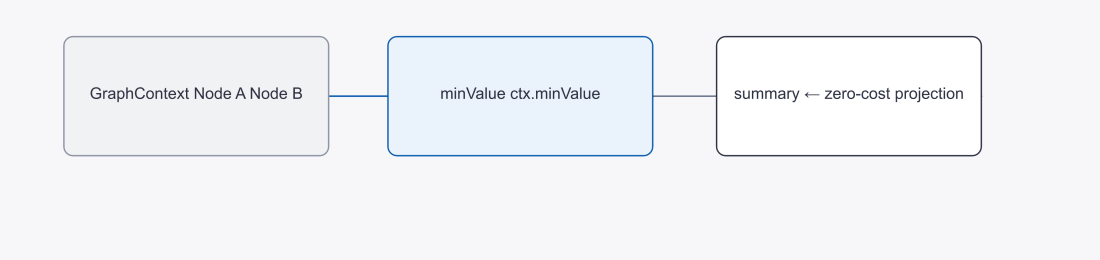
In Chapter 3 you learned to think in dependencies. But **dependencies only determine *when* a node runs — not *what data* it receives.**

Getting data flow right is the difference between a clean, maintainable graph and one where every node does too much work reshaping data for the next node. BLOGE gives you a layered toolkit for data:

Layer	Tool	Scheduling cost	When to use
Routing	Input binding (`input { … }`)	None — resolved at assembly time	Passing fields from context or upstream outputs
Projection	Transform block (`transform { … }`)	None — virtual node, no operator	Reshaping or renaming fields without business logic
Computation	Operator node (`node … : Op { … }`)	Full — scheduled, timed, retriable	Business logic, I/O, or anything with side effects

Choosing the right layer keeps your graphs fast and readable.

4.5 Mental Model



- **Input bindings** pull data from `ctx` or from `someNode.output.field`.
- **Transforms** are virtual — the engine never schedules them on a thread. They simply create a new data shape from existing values.
- **Operator nodes** execute real code with timeout, retry, and fallback.

4.6 First Working Example

4.6.1 Input bindings — the basics

Every `input { … }` block is a set of key-value bindings. The right-hand side is an **expression** that the engine evaluates at input-assembly time.

```
// Simplified teaching example — demonstrates all three data sources in one block
node calcPrice : CalcPriceOperator {
  input {
    user     = fetchUser.output          // whole output object
    products = fetchProducts.output      // whole output object
```

```
    userId   = fetchUser.output.id        // nested field
    tax      = ctx.taxRate               // context value
    label    = "order"                   // literal string
  }
}
```

Three data sources, one block:

Source	Syntax	Example
Context	`ctx.fieldName`	`ctx.taxRate`
Upstream output	`nodeName.output` or `nodeName.output.field`	`fetchUser.output.id`
Literal	A quoted string or number	`"order"`, `42`

4.6.2 Path expressions

Path expressions let you navigate nested structures:

```
input {
  city = fetchUser.output.address.city
}
```

This traverses the output object: `output` → `address` → `city`.

4.6.3 Safe navigation and null coalescing

When a field might be null, use the **safe navigation operator** (`?.`) to avoid null-pointer errors, and the **null coalescing operator** (`??`) to provide a default:

```
// From safe-navigation.bloge
node normalize : Normalize {
  input {
    profileName = ctx.request?.user?.name
    result      = fetch?.result?.value ?? "missing"
    names       = ctx.items?.map(x -> x.profile?.name)
  }
}
```

(From `safe-navigation.bloge`.)

Operator	Meaning
`?.`	If the left side is null, stop and return null instead of throwing
`??`	If the left side is null, use the right-side default

These compose naturally: `a?.b?.c ?? "default"` means "navigate into `a.b.c`; if anything along the way is null, use `"default"`."

4.6.4 Accessing lists and maps

Path expressions reach **fields**. To reach **elements** of a list or **entries** of a map, use the index access operators:

Syntax	Meaning
`xs[i]`	Element at position `i`; throws if the index is out of range
`xs?[i]`	Safe element access; yields `null` instead of throwing
`xs[-1]`	Negative indices count from the tail (`-1` is the last element)

Syntax	Meaning
`m["key"]`	Look up a map entry by its string key
`(expr).field`	Read a field from an arbitrary expression result
`(expr)?.field`	Safe field read on an expression result

Index access composes with everything else you already know — safe navigation, null coalescing, lambdas — and is resolved at input-assembly time, with **no extra scheduling overhead**:

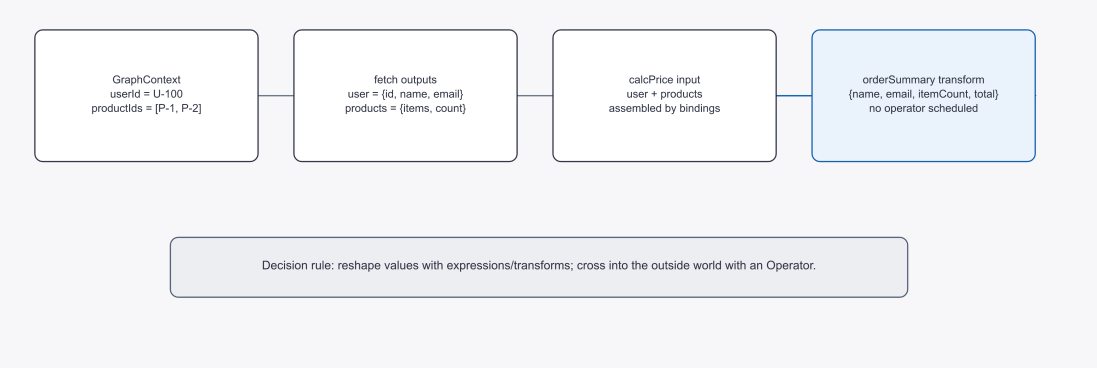
```
// From ch04/order-summary-index-access.bloge
node fetchOrders : FetchOrdersOperator {
  input {
    customerId = ctx.customerId
  }
}

transform summary {
  firstOrderId = fetchOrders.output.recentOrderIds[0]
  lastOrderId  = fetchOrders.output.recentOrderIds[-1]
  safeFirst    = fetchOrders.output?.recentOrderIds?[0] ?? "none"
  regionLabel  = ctx.regionMap["north-east"]
  totalOrders  = fetchOrders.output.totalOrders
}
```

> ***Common Trap — `xs[i]` vs `xs?[i]`***
> *Both forms read element `i`, but they behave very differently when the list is shorter than expected:*
>
> - *`xs[i]` is **strict**. An out-of-range read fails fast with an expression-evaluation error. Use it when an empty or short list is a real bug — for example, when fetching an order you just created.*
> - *`xs?[i]` is **lenient**. An out-of-range read yields `null`, which pairs with `??` to provide a default. Use it when "no data yet" is a normal outcome — for example, when reading the latest entry from an optional history list.*
>
> *Reach for the strict form by default; switch to the safe form only when you have a concrete reason to tolerate missing data.*

4.7 Follow One Order Value Through Four Shapes

Expression syntax becomes easier when you stop reading it token by token and follow one business object. The order begins as identifiers in `GraphContext`, gains facts from two Operators, becomes the input to pricing, and ends as a smaller summary for the next step.



4.7.1 Name the shape at every boundary

Boundary	Shape available there	Why it exists
Graph entry	`{userId, productIds}`	Stable request facts supplied by the caller
After fetch	`{user}` and `{products}`	External capabilities have enriched the request
Pricing input	`{user, products}`	Input bindings assemble only what pricing needs
Summary transform	`{name, email, itemCount, total}`	A cheap projection for downstream readers

This is data lineage in miniature. When `customerEmail` is wrong, walk left: the transform reads `fetchUser.output.email`, so inspect the user output before debugging the price Operator. The graph has already told you which producer owns the fact.

4.7.2 Choose by effect, not by line count

Use an expression or `transform` when the operation is a deterministic reshape of values already present. Use an Operator when the step crosses into the outside world, consumes time, can fail independently, or needs retry and timeout policy. A ten-line projection is still a transform; a one-line HTTP call is still an effectful Operator.

The boundary is testable: no Operator invocation should appear for `orderSummary`, while `fetchUser` and `fetchProducts` must have invocation evidence. That observation is more useful than calling transforms "zero cost" without explaining what the engine did not schedule.

4.8 Break It Apart

4.8.1 Transforms — projections without Operator scheduling

A **transform** is a virtual node. It reshapes data without running an operator:

```
// From ch03/order-process.bloge
transform orderSummary {
  customerName  = fetchUser.output.name
  customerEmail = fetchUser.output.email
  itemCount     = fetchProducts.output.items.size
  total         = calcPrice.output.total
}
```

The engine does **not** schedule a transform on a thread. It resolves the bindings inline and makes the result available as `orderSummary.customerName`, `orderSummary.itemCount`, etc.

A minimal transform from the conformance suite (`basic.bloge`):

```
graph g {
  node source : Op {}
  transform summary {
    greeting: String = "hello"
    result: String   = source.output.value
  }
}
```

Notice that transforms can include **type annotations** (`greeting: String`).

When to use a transform vs. an operator:

Use a transform when …	Use an operator when …
You only rename or pick fields	You call an external service

Use a transform when …	Use an operator when …
No side effects	You run business rules that matter
No need for timeout/retry	You need resilience policies
Pure data reshaping	You need independent observability

4.8.2 Lambda expressions in input bindings

For collection operations, BLOGE supports **lambda expressions** directly in input bindings. These are evaluated at input-assembly time with **zero scheduling overhead**.

From `lambda-map.bloge`:

```
node a : OpA {
  input {
    names = ctx.items.map(x -> x.name)
  }
}
```

The expression `ctx.items.map(x -> x.name)` takes a list of items from the context and extracts the `name` field from each one.

A richer example from `order-enrichment-lambda.bloge`:

```
node enrichOrders : EnrichOrdersOperator {
  depends_on = [fetchOrders, fetchProducts]
  input {
    enrichedOrders = fetchOrders.output.orders
      .map(o -> {
        orderId:     o.orderId,
        totalValue:  o.price * o.quantity,
        taxAmount:   o.price * o.quantity * 0.1,
        productName: fetchProducts.output.catalogue.associate(p -> p.id, p -> p.name)
      })
      .filter(o -> o.totalValue > ctx.minValue)
      .sortBy(o -> o.totalValue)

    totalRevenue = fetchOrders.output.orders
      .filter(o -> o.price * o.quantity > ctx.minValue)
      .reduce(0, (acc, o) -> acc + o.price * o.quantity)
  }
}
```

Available collection operations:

Operation	Syntax	Result
`map`	`.map(x -> expr)`	New list with each element transformed
`filter`	`.filter(x -> condition)`	List with only matching elements
`reduce`	`.reduce(init, (acc, x) -> expr)`	Single accumulated value
`sortBy`	`.sortBy(x -> key)`	Sorted list

Lambda expressions can produce **object literals** (`{ field: value, … }`) and be **chained** (`.map(…).filter(…).sortBy(…)`).

4.9 Common Trap

4.9.1 [X] Using an operator node for pure data reshaping

```
// WRONG — this is a transform, not business logic
node formatOutput : OutputFormatterOperator {
  input {
    customerName = fetchUser.output.name
    total        = calcPrice.output.total
  }
}
```

If `OutputFormatterOperator` just picks fields and renames them, it's paying the cost of a full node (scheduling, timeout tracking) for something a transform does for free:

```
// CORRECT — projection without an Operator invocation
transform formatOutput {
  customerName = fetchUser.output.name
  total        = calcPrice.output.total
}
```

Rule of thumb: If there are no side effects, no I/O, and no business rules, use a transform.

4.10 What Goes Wrong

4.10.1 Null path access without safe navigation

If you write:

```
input {
  city = fetchUser.output.address.city
}
```

and `address` is null at runtime, the node fails while evaluating the binding because the plain `.city` access tries to walk through a null segment. BLOGE only short-circuits this lookup when you use `?.`.

Fix: Use safe navigation to handle missing data:

```
input {
  city = fetchUser.output.address?.city ?? "Unknown"
}
```

With `?.`, a null `address` produces null instead of an error. The `??` operator then substitutes the default.

4.11 Guided Rewrite

Start from this graph that uses too many operator nodes:

```
graph report {
  node fetchSales   : FetchSalesOperator   { input { region = ctx.region } }
  node fetchReturns : FetchReturnsOperator { input { region = ctx.region } }

  // This node just picks fields — wasteful as an operator
  node buildSummary : SummaryBuilderOperator {
    input {
```

```
      totalSales   = fetchSales.output.total
      totalReturns = fetchReturns.output.total
    }
  }
}
```

Rewrite `buildSummary` as a transform:

```
graph report {
  node fetchSales   : FetchSalesOperator   { input { region = ctx.region } }
  node fetchReturns : FetchReturnsOperator { input { region = ctx.region } }

  transform buildSummary {
    totalSales   = fetchSales.output.total
    totalReturns = fetchReturns.output.total
    netRevenue   = fetchSales.output.total
  }
}
```

Now `buildSummary` adds no Operator invocation or separately scheduled node. If you need a computed field like `netRevenue = totalSales - totalReturns`, that can be expressed as an arithmetic expression in the binding — no operator required.

4.12 Brain Check

1. Name the three data sources available in an input binding. *(Context values via `ctx.*`, upstream node outputs via `node.output.*`, and literal values.)*
2. What does `?.` do in `ctx.user?.profile?.name`? *(If `user` or `profile` is null, the entire expression returns null instead of throwing a null-pointer error.)*
3. When should you use a transform instead of an operator node? *(When you only need to reshape or rename data without side effects, I/O, or business rules.)*
4. What is the scheduling cost of a lambda expression in an input binding? *(Zero — it is evaluated at input-assembly time, not scheduled as a separate node.)*
5. *Design question:* You have a transform with six fields and an operator node that uses three of those fields. Should the operator read from the transform or from the original upstream outputs? *(Either works, but reading from the transform is cleaner if those fields are already projected and named there. The general rule: minimise the number of distinct upstream references in a single input block.)*

4.13 Lab

1. Open `order-enrichment-lambda.bloge`.
2. Add a new input binding that computes the **average order value** using `reduce` to sum totals and then divides by the list size.
3. Add a **transform** node that combines the `totalRevenue` with a `ctx.budgetTarget` to produce a `percentOfBudget` field.
4. Verify mentally: which parts add no separately scheduled Operator (transform + lambda), and which would require one?

4.14 Experiment acceptance card

- **Expected and observed:** An order value crosses context, output, binding, and transform shapes.
- **Failure and recovery:** Point a strict path at a missing field; restore the path or use an explicit safe read.
- **Proof boundary:** Proves binding and path behavior, not truthful or complete input.
- **Exercise contract:** Order sample; change one field path; deliver the value and failure reason; stop when both are attributable.

4.15 Coding practice

4.15.1 BP-01 — Check existing functions

For a field rename or lightweight computation, inspect the runtime catalog before creating an Operator. Check null, order, precision, and locale semantics; an existing name alone does not establish a fit.

Try one change: Replace a trimming node with a binding, then test a blank and a null name. Explain why the valid-data contract still needs validation.

Reasoning, counterexamples, and limits

4.15.2 BP-02 — Name bounded computation

Extract an ordinary helper when a calculation needs a clear name. Publish a typed function only when graph authors need it; keep business inputs explicit and test metadata against behavior.

Try one change: Use the Chapter 34 policy-code function to compare the same input under root and Turkish locales. Decide whether trim alone satisfies the contract.

Reasoning, counterexamples, and limits

4.15.3 BP-08 — Retain real dependencies and explicit data

Pass business facts through input/output rather than shared mutable context. Remove accidental serialization but retain required control ordering, even without a field reference.

Try one change: Let credit and income checks start independently, then join their facts. Add a charge that must follow approval and explain that control prerequisite.

Reasoning, counterexamples, and limits

4.16 Recap

- **Input bindings** wire data from context, upstream outputs, and literals into node inputs — evaluated at assembly time with no scheduling overhead.
- **Path expressions** navigate nested structures; `?.` provides safe navigation and `??` provides null coalescing.
- **Transforms** are virtual nodes that reshape data without an Operator invocation or separately scheduled task; expression evaluation still does work.
- **Lambda expressions** (`map`, `filter`, `reduce`, `sortBy`) operate on collections directly in input bindings with no scheduling overhead.
- **Choose the right layer:** input bindings for routing, transforms for projection, operator nodes for business logic.

4.17 Next Step

In Chapter 5 — Branches That Decide you will learn how to add conditional control flow to your graphs — routing execution down different paths based on a node's output.

4.18 Reference Links

- DSL Specification — expression grammar and transform syntax
- Operator Design Specification — data transformation layers and operator boundaries
- Getting Started — transform and expression examples
- `order-enrichment-lambda.bloge` — source
- `safe-navigation.bloge` — conformance fixture
- `lambda-map.bloge` — conformance fixture
- `basic.bloge` (transform) — conformance fixture

CHAPTER

—

Arc 1 Recap — Introductory (Chapters 1–4)

You now have the vocabulary and mental model to read a small BLOGE DAG and predict how its declared data dependencies move.

0.1 What You Learned

Chapter 1 — What BLOGE Is and Why It Exists gave you the three defining properties of BLOGE: workflows are **visible** (the graph shape is explicit), execution is **dependency-driven** (you declare edges, the engine decides order), and resilience is **built in** (retry, timeout, fallback are part of the model).

Chapter 2 — Your First Graph made the theory concrete. You built a graph, ran it, inspected `GraphResult`, and saw that the engine — not you — decides what can run in parallel.

Chapter 3 — Thinking in Dependencies sharpened your eye for dependency edges. You learned to read a DAG, spot unnecessary serialisation, and distinguish control dependencies from data dependencies.

Chapter 4 — Data That Flows completed the picture. You learned how data enters via context, flows between nodes through input bindings, and exits through node outputs. You practiced path expressions, safe navigation (`?.`), null coalescing (`??`), transforms, and lambda input bindings.

0.2 What You Should Be Able to Do Now

- Describe BLOGE in one sentence without jargon.
- Draw the dependency graph of a small workflow on paper and predict its execution order.
- Write a simple `.bloge` file with three or four nodes, explicit dependencies, and input bindings.
- Read a `GraphResult` and extract typed outputs safely.
- Explain the difference between a transform and an operator node.

0.3 Common Mistakes at This Stage

Mistake	Why It Happens	Fix
Adding unnecessary `depends_on` edges	Confusing ”this runs after that” with ”this actually needs that node's data”	Ask: does the downstream node **read** the upstream output? If not, drop the edge.
Ignoring `findOutput` in favor of `getOutput`	Assuming every node always completes	Use `findOutput` or `getOutputOrDefault` when the node could be skipped or fail.

Mistake	Why It Happens	Fix
Putting business logic in transforms	Transforms are projections without an Operator invocation or a separately scheduled node, but their expressions still consume compute	If the code has side effects or can fail, it belongs in an operator.
Hard-coding values instead of using `ctx.*`	Copy-pasting from an example without parameterising	Move environment- or caller-specific values into the graph context.

0.4 Key Vocabulary

Term	Meaning
Graph	A directed acyclic graph of business steps with explicit data and control flow.
Node	One step in the graph, backed by an operator.
Operator	A typed function `(I, GraphContext) → O` that does the real work.
Dependency edge	A directed link that says ”this node cannot start until that node finishes.”
Input binding	An expression that resolves a node's input from context, upstream outputs, or literals.
Transform	A projection-only node — reshapes data but carries no scheduling overhead.
GraphResult	The immutable record returned after execution: success flag, outputs, statuses, timings.

0.5 Try It: Remove a False Dependency

Draw breakfast as four nodes: boil water, toast bread, brew coffee, and serve. Add only the dependencies that are physically necessary. Then remove one edge and explain whether you gained safe parallelism or introduced a race.

0.6 Where to Go Next

Arc 2 teaches you to make decisions inside a graph (branches), handle failure (resilience), design clean operators, and write maintainable, reviewable DSL files. Start with Chapter 5 — Branches That Decide.

Learning Arc

Arc 2 — Maintainable Authoring

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

CHAPTER
05

Arc 2 — Maintainable Authoring

Chapter 5 — Branches That Decide

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

Promise: *By the end of this chapter you will know how to route execution down different paths based on a node's output — and you will understand exactly what happens to the paths that are* not *taken.*

5.1 Learning Goals

1. Write a `branch on` block that routes to one of several target nodes based on a runtime value.
2. Use `otherwise` to provide a default path when no explicit case matches.
3. Explain how the engine marks un-chosen target nodes as **SKIPPED**.
4. Distinguish `branch on` (control-plane routing) from the `when` expression (data-plane value selection).
5. Diagnose common branch mistakes, then use **inclusive branching** (`branch mode=inclusive on ...`) when multiple matches must run in parallel; explain how that OR-split differs from the default XOR-split.

5.2 Prerequisites

- Chapter 4 — Data That Flows

5.3 Source Examples

File	What it shows
`ch05/ticket-routing.bloge`	Three-way branch with string cases and `otherwise`
`ch05/loan-approval.bloge`	Branch after a multi-stage risk pipeline with fan-out, transform, and `otherwise`
`boolean.bloge`	Minimal boolean branch — `true` / `false`
`otherwise.bloge`	Minimal `otherwise`-only branch
`ch05/multi-flag-approval.bloge`	Inclusive branching — multiple review tracks run in parallel based on flags

5.4 Why This Matters

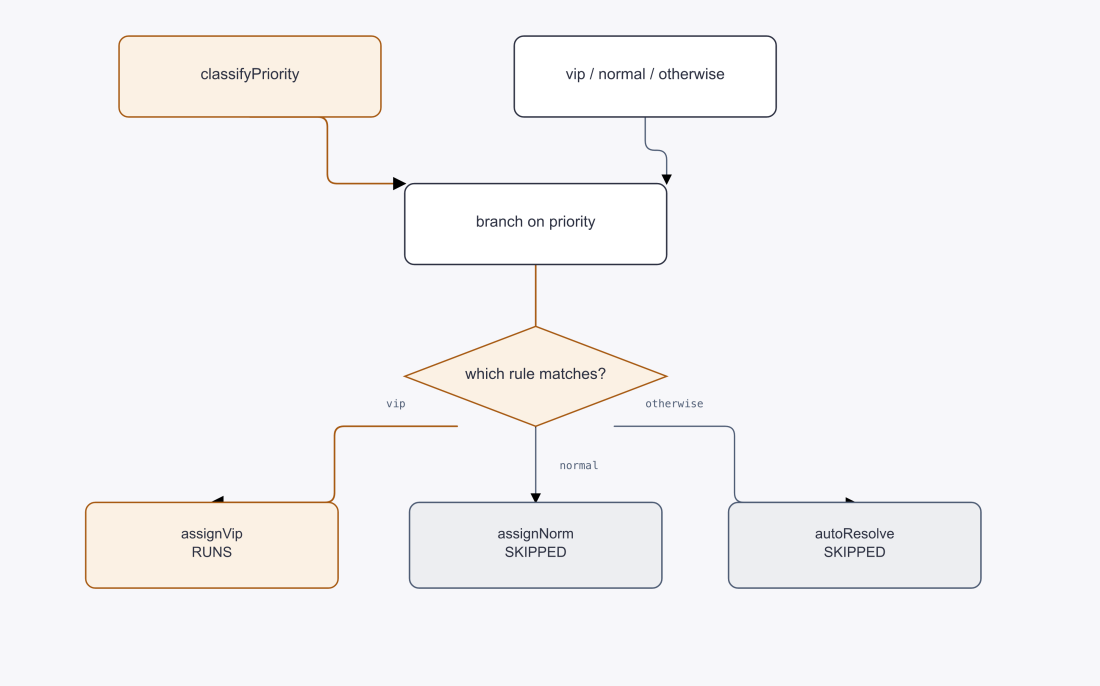
Until now every node in your graphs has always executed. Real workflows aren't like that:

- A credit check **approves or rejects** — you don't run both paths.
- A support ticket is routed to **one** handler, not all of them.
- A loan decision triggers approval, rejection, **or** manual review.

In procedural code you'd write `if/else` blocks. In BLOGE the equivalent is `branch on` — a first-class routing decision declared inside the graph, not buried in operator code. The engine evaluates the condition, activates exactly one downstream node, and **skips** the rest.

This keeps routing decisions visible in the graph definition and in studio tooling, instead of hiding them inside an operator's implementation.

5.5 Mental Model



Think of a branch as a **railroad switch**. One track is selected; the other tracks are blocked. The engine:

1. Waits for the **source node** (`classifyPriority`) to complete.
2. Reads the **condition field** (`.priority`) from that node's output.
3. Tests each **case** in declaration order until one matches.
4. Dispatches the matching target node; marks every other target as `SKIPPED`.

If no case matches and there is no `otherwise`, **none** of the target nodes run. That is almost always a bug — which is why you should include `otherwise` whenever the set of values is not exhaustively known.

5.6 First Working Example

Here is the branch from `ch05/ticket-routing.bloge`:

This 63-line graph is kept whole because the branch result depends on the two parallel fetches, their fan-in, and all

three declared targets; trimming any of those parts would hide the control path the reader must trace.

```
graph ticketRouting {

  node fetchCustomer : FetchCustomerOperator {
    input { customerId = ctx.customerId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchTicketHistory : FetchTicketHistoryOperator {
    input { customerId = ctx.customerId }
    timeout = 3s
  }

  node analyzeSentiment : AnalyzeSentimentOperator {
    depends_on = [fetchCustomer, fetchTicketHistory]
    input {
      customer = fetchCustomer.output
      history  = fetchTicketHistory.output
      message  = ctx.message
    }
    timeout = 5s
    retry = { attempts: 1, backoff: 500ms, strategy: exponential }
    fallback = { sentiment: "neutral", score: 0.0, keywords: [] }
  }

  node classifyPriority : ClassifyPriorityOperator {
    depends_on = [analyzeSentiment]
    input {
      customer  = fetchCustomer.output
      sentiment = analyzeSentiment.output
    }
  }

  branch on classifyPriority.output.priority {
    "vip"    -> assignVipAgent
    "normal" -> assignNormalAgent
    otherwise -> autoResolve
  }

  node assignVipAgent : AssignVipAgentOperator {
    depends_on = [classifyPriority]
    input {
      customerId = fetchCustomer.output.id
      priority   = "vip"
    }
  }

  node assignNormalAgent : AssignNormalAgentOperator {
    depends_on = [classifyPriority]
    input {
      customerId = fetchCustomer.output.id
      priority   = "normal"
    }
  }

  node autoResolve : AutoResolveOperator {
```

```
    depends_on = [classifyPriority]
    input {
      customerId = fetchCustomer.output.id
      keywords   = analyzeSentiment.output.keywords
    }
  }
}
```

Read it as a story:

1. **Two parallel fetches** — customer profile and ticket history.
2. **Sentiment analysis** fans in from both, with retry and a fallback.
3. **Priority classification** produces `"vip"`, `"normal"`, or something else.
4. **The branch** — `branch on classifyPriority.output.priority` — picks exactly one handler. If the priority is `"vip"`, only `assignVipAgent` runs. If it's `"normal"`, only `assignNormalAgent`. Anything else falls through to `autoResolve` via `otherwise`.

> ***Key insight:*** *The three handler nodes all declare `depends_on = [classifyPriority]`, but the branch ensures only one of them actually executes. The others receive `SKIPPED` status.*

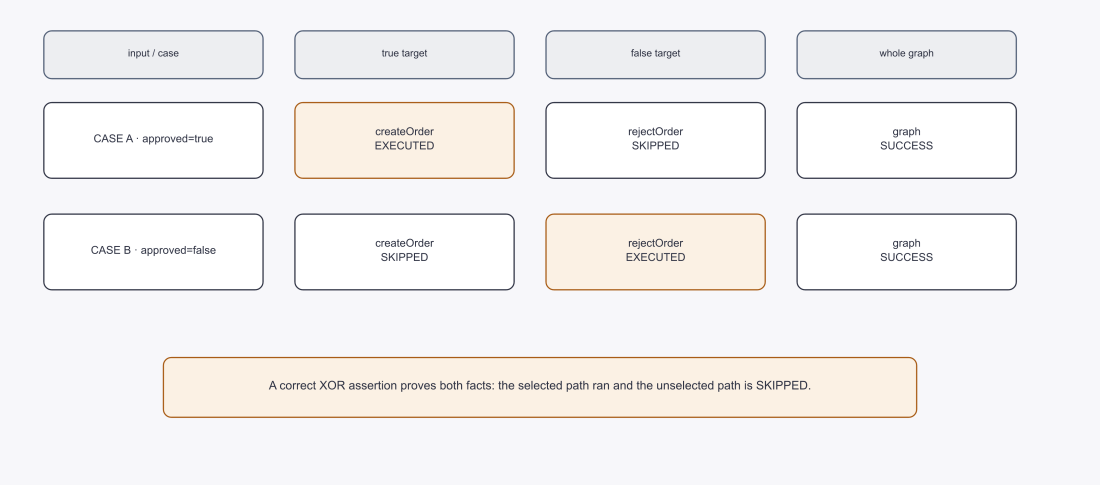
The same `branch on` written with the fluent Java API:

```
Graph graph = Graph.builder("ticketRouting")
    // ... nodes defined as before ...
    .branch("classifyPriority")
        .on("priority")
        .when(value -> "vip".equals(value), "assignVipAgent")
        .when(value -> "normal".equals(value), "assignNormalAgent")
        .otherwise("autoResolve")
    .build();
```

`Edge.conditional()` is the underlying mechanism. Each `.when()` creates a conditional edge with a value predicate; `.otherwise()` creates the default edge. The engine's branch evaluation logic is identical regardless of whether the graph was built from DSL or the Java API.

5.7 Prove XOR with Two Cases, Not One Happy Path

An exclusive branch makes two promises: exactly one matching target is selected, and the other target does not execute. Testing only the approval path proves half of that contract.



5.7.1 Read the unselected path as a result

Case	`createOrder`	`rejectOrder`	Business meaning
`approved=true`	`EXECUTED`	`SKIPPED`	The order was created and rejection code did not run
`approved=false`	`SKIPPED`	`EXECUTED`	Rejection ran and no order was created

`SKIPPED` is not missing data and not a failure. It is positive evidence that the graph evaluated the branch and excluded that node. The RC1 foundation probe therefore asserts both `assertNodeExecuted("createOrder")` and `assertNodeSkipped("rejectOrder")`.

5.7.2 Keep the first decision narrow

Start with XOR while learning because one input selects one path. Use inclusive branching only when the business statement genuinely allows several actions at once, such as "notify compliance **and** request extra documents". Use a decision table when multiple independent conditions determine an outcome and the rule set needs coverage analysis. Those are stronger models, not cleverer spellings of a two-way choice.

One mutation makes the boundary visible: change `approved=true` to `false` without changing anything else. If only the selected and skipped statuses swap, the branch is isolated. If unrelated nodes change, the decision is coupled to more graph state than the picture admits.

5.8 Break It Apart

5.8.1 Anatomy of a `branch on` block

```
branch on <sourceNode>.output.<field> {
  <value1> -> <targetNode1>
  <value2> -> <targetNode2>
  otherwise -> <defaultNode>
}
```

Part	What it means
`branch on`	Keyword pair that opens a conditional routing block
`<sourceNode>.output.<field>`	A **node output path** expression — the runtime value the engine reads to decide
`<value> -> <target>`	A **case**: if the condition equals `<value>`, dispatch `<target>`
`otherwise -> <target>`	The **default case**: if no other case matches, dispatch this target

5.8.2 Case matching rules

The compiler builds a `Predicate<Object>` for each case using **type-lenient comparison**:

Case literal	How it matches at runtime
`true` / `false`	If the runtime value is a `Boolean`, direct `==`. Otherwise, case-insensitive `String.valueOf()` comparison.
`"someString"`	String equality, or `String.valueOf()` of the runtime value.
42 (number)	If the runtime value is a `Number`, `doubleValue()` comparison. Otherwise, no match.

Cases are evaluated **in declaration order** — the first match wins.

5.8.3 Boolean branches

The simplest branch uses boolean values. From the conformance fixture `boolean.bloge`:

```
graph g {
  node a : Op {}
  node b : Op {}
  node c : Op {}
  branch on a.output.approved {
    true -> b
    false -> c
  }
}
```

If `a.output.approved` is `true`, node `b` runs and `c` is skipped. If `false`, `c` runs and `b` is skipped.

5.8.4 The `otherwise` catch-all

From the conformance fixture `otherwise.bloge`:

```
graph g {
  node a : Op {}
  node b : Op {}
  branch on a.output.status {
    otherwise -> b
  }
}
```

An `otherwise`-only branch means "whatever the value is, always dispatch `b`." This is unusual but valid — it can be useful as a placeholder during prototyping.

5.8.5 What `SKIPPED` means

When the engine selects one branch target, every other target node is marked `SKIPPED` in the `NodeStatus` enum. A skipped node:

- **Never executes** its operator.
- **Produces no output** — any downstream node that references a skipped node's output will not have that data available.
- Appears as `SKIPPED` in the `GraphResult` and in execution listeners.

5.9 Inclusive Branching: When Multiple Paths Should Run

Everything above describes **exclusive branching** (XOR-split) — the default behaviour where exactly one matching branch fires and the rest are skipped. `branch on` also supports an **inclusive** mode (OR-split) that lets *all* matching branches fire in parallel.

5.9.1 Why you need it

Consider a loan application that must be reviewed along several independent dimensions. After assessing risk flags, the application might simultaneously require a human review, a compliance check, **and** a pricing review. With an exclusive branch you would have to pick one — but the real workflow needs all of them.

5.9.2 XOR-split vs OR-split at a glance

	Exclusive (XOR-split, default)	Inclusive (OR-split, `inclusive`)
Branches that fire	Exactly one — the first match	All that match, in parallel
`otherwise` fires when	No explicit case matches	*Zero* branches matched
Unmatched targets	Marked `SKIPPED`	Marked `SKIPPED`
DSL keyword	`branch on ... { }`	`branch mode=inclusive on ... { }`
Java API	`.branch().on().when()...`	`.branch().on().inclusive().when()...`

5.9.3 DSL syntax

Set `mode=inclusive` before `on`:

```
branch mode=inclusive on assessFlags.reviewFlags {
  "NEEDS_REVIEW"     -> manualReview
  "NEEDS_COMPLIANCE" -> complianceReview
  "NEEDS_PRICING"    -> pricingReview
  otherwise               -> autoApprove
}
```

If `assessFlags.reviewFlags` contains both `"NEEDS_REVIEW"` and `"NEEDS_PRICING"`, **both** `manualReview` and `pricingReview` run in parallel while `complianceReview` is skipped. The `otherwise` target (`autoApprove`) only runs when *none* of the cases matched.

5.9.4 Java API

Use the `.inclusive()` method on `BranchBuilder`:

```
graph.branch("assessFlags")
     .on("reviewFlags")
     .inclusive()                          // enable OR-split
     .when(v -> v instanceof List<?> l
               && l.contains("NEEDS_REVIEW"), "manualReview")
     .when(v -> v instanceof List<?> l
               && l.contains("NEEDS_COMPLIANCE"), "complianceReview")
     .when(v -> v instanceof List<?> l
               && l.contains("NEEDS_PRICING"), "pricingReview")
     .otherwise("autoApprove");
```

Under the hood this sets the `inclusive` field on `Edge.Conditional` to `true`.

5.9.5 Worked example — multi-flag approval

See `ch05/multi-flag-approval.bloge` for a complete graph. The key flow:

1. `fetchApplication` retrieves the loan application.
2. `assessFlags` analyses risk and produces a list of review flags.
3. The **inclusive branch** fans out to every matching review track:
   - `"NEEDS_REVIEW"` → `manualReview`
   - `"NEEDS_COMPLIANCE"` → `complianceReview`
   - `"NEEDS_PRICING"` → `pricingReview`
4. If the flags list is empty, `otherwise` routes to `autoApprove`.

Because the branch is inclusive, a single application can trigger two or even all three review tracks simultaneously.

5.9.6 When to use inclusive vs exclusive

Use exclusive (default) when…	Use inclusive when…
Exactly one outcome applies (approve / reject / defer)	Multiple independent outcomes can apply at the same time
The condition is a single-valued enum or boolean	The condition is a collection or bitmask of flags
You want first-match short-circuit semantics	You want fan-out to all matching downstream nodes

5.9.7 `otherwise` behaviour difference

- **Exclusive:** `otherwise` fires when no explicit case matches — it acts as a catch-all default.
- **Inclusive:** `otherwise` fires **only** when *zero* cases matched. If at least one case matched, `otherwise` is skipped — even if some other cases did not match.

5.10 When You Need a Node That Always Runs — `upstream_policy = ALL_RESOLVED`

The default scheduling rule is: a node runs only if **at least one upstream completed**. That is what you want for ordinary fan-in. But some nodes are deliberately wired downstream of a branch and need to run **regardless of which path was taken** — for example:

- An **outcome recorder** that logs which path won
- A **cleanup** node that runs after both success and failure paths
- A **metrics** collector that does not care about the business outcome

If you declare `depends_on = [branchTargetA, branchTargetB, branchTargetC]` and the branch only picked one path, the others are `SKIPPED` — and your recorder is also `SKIPPED`, because it has zero `COMPLETED` upstreams.

`upstream_policy = ALL_RESOLVED` tells the engine: "run me as soon as **all** my upstreams have reached a terminal state, including `SKIPPED` and `CANCELLED`." The name reads as "wait for all upstreams to be resolved", not "wait for all to succeed".

```
node recordOutcome : RecordOutcomeOperator {
  depends_on      = [assignVipAgent, assignNormalAgent, autoResolve]
  upstream_policy = ALL_RESOLVED
  input {
    // SKIPPED upstreams have no output — pair with safe navigation
    vip    = assignVipAgent.output?.agentId    ?? null
    normal = assignNormalAgent.output?.agentId ?? null
    auto   = autoResolve.output?.ticketId      ?? null
  }
}
```

```
.node("recordOutcome", new RecordOutcomeOperator())
    .dependsOn("assignVipAgent", "assignNormalAgent", "autoResolve")
    .upstreamPolicy(UpstreamPolicy.ALL_RESOLVED)
```

> ***Brain check:*** *what is the difference between `upstream_policy = ALL_RESOLVED` and connecting `recordOutcome` to* only the path that actually ran*? Hint: think about how many edges you have to write, and what happens when you add a fourth branch.*

5.11 Common Trap

5.11.1 [X] Branching on a non-output expression

```
// WRONG — branch condition must be a node output path
branch on ctx.priority {
  "high" -> fastTrack
  otherwise -> normalQueue
}
```

The compiler requires a **node output path** (or transform field path) as the condition expression. You cannot branch directly on a context value. If you need to route on context data, introduce a node or transform that surfaces the value as output:

```
// CORRECT — use a transform to expose the context value
transform routing {
  priority = ctx.priority
}

branch on routing.priority {
  "high" -> fastTrack
  otherwise -> normalQueue
}
```

5.11.2 [X] Forgetting `otherwise` with open-ended values

```
// DANGEROUS — what if decision is "review"?
branch on makeDecision.output.decision {
  "approved" -> approveLoan
  "rejected" -> rejectLoan
}
```

If `makeDecision` returns `"review"`, no case matches and **no target node runs**. Always include `otherwise` when the set of possible values is not exhaustively known:

```
// SAFE
branch on makeDecision.output.decision {
  "approved" -> approveLoan
  "rejected" -> rejectLoan
  otherwise  -> manualReview
}
```

5.11.3 [X] Confusing `branch on` with `when`

`branch on` and `when` share arrow syntax (`->`) and `otherwise`, but they are **completely different**:

	`branch on`	`when`
Plane	Control — decides which *nodes* run	Data — decides which *value* a field takes
Where it appears	Graph body, top level	Inside an expression (input binding or transform)
Effect	Activates one node, skips others	Produces a single value

Don't use `branch on` when you just need to compute a value. Use `when` instead.

5.11.4 [X] Confusing inclusive branching with parallel execution

```
// WRONG assumption — this is still an exclusive branch!
branch on classifyRisk.level {
  "HIGH"   -> escalate
```

```
  "MEDIUM" -> review
  otherwise     -> autoApprove
}
```

Adding concurrency elsewhere does **not** make an exclusive branch "run things in parallel". Without `mode=inclusive`, only the first matching case fires. If you need multiple branches to fire simultaneously, you must explicitly add `mode=inclusive` to the branch block:

```
// CORRECT — mode=inclusive enables OR-split
branch mode=inclusive on assessFlags.reviewFlags {
  "NEEDS_REVIEW"     -> manualReview
  "NEEDS_COMPLIANCE" -> complianceReview
  otherwise               -> autoApprove
}
```

Also remember: inclusive branching is for cases where the condition value can match *multiple* cases (e.g., a list of flags). If the condition is a single-valued field like a string enum, inclusive and exclusive will behave identically — prefer exclusive for clarity.

5.11.5 [X] Using `branch on` as a decision table

`branch on` is the right tool for **routing** — deciding which downstream node runs. It starts to hurt when you use it for **classification** or **policy lookup**, where a set of inputs maps to one output value.

The smell is easy to spot: every branch target exists only to return the same shape of result.

```
// Awkward — routing nodes just to classify a value
branch on creditScore.output.band {
  "platinum" -> assignPlatinumTier
  "gold"     -> assignGoldTier
  otherwise  -> rejectApplication
}
```

That design creates three problems:

1. **Buried outputs.** The result is spread across several node output paths instead of one stable place.
2. **Weak multi-match semantics.** `branch on` can choose one arm or fan out, but it cannot express "collect all matching rules and return them" cleanly.
3. **No policy lint.** Tooling cannot warn you that a rule set is incomplete or that two policy rows conflict.

Use a `decision_table` when the logic is really a rule table:

```
graph creditScreening {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

The result is always at `credit_tier.output.value`, no matter how many rules you add later.

Need	Prefer
Route execution to one or more downstream nodes	`branch on`
Classify inputs into one value	`decision_table hit=first` or `hit=unique`
Accept multiple matches only if they agree	`decision_table hit=any`
Return all matching outcomes	`decision_table hit=collect`

Full syntax, hit policies, lint rules, and error codes live in Appendix F — Decision Table. Chapter 9 shows the `decision-table/missing-otherwise` and `decision-table/collect-otherwise` lint recipes, and Chapter 34 turns this trap into a hands-on lab.

5.12 What Goes Wrong

5.12.1 Missing `otherwise` with an unexpected value

```
branch on makeDecision.output.decision {
  "approved" -> approveLoan
  "rejected" -> rejectLoan
}
```

If `makeDecision` returns `"review"`, no case matches:

```
GraphResult {
  status: COMPLETED,
  nodeStatuses: {
    approveLoan: SKIPPED,
    rejectLoan:  SKIPPED
  }
}
```

The graph completes, but **no downstream work happened**. Any node that depended on `approveLoan` or `rejectLoan` output will have missing data.

Fix: Add `otherwise -> manualReview` to ensure at least one branch always activates.

5.13 Guided Rewrite

Look at the full `ch05/loan-approval.bloge` example. It runs four risk checks in parallel, aggregates them, and branches:

```
branch on makeDecision.output.decision {
  "approved"  -> approveLoan
  "rejected"  -> rejectLoan
  otherwise   -> manualReview
}
```

Questions to work through:

1. **How many nodes run before the branch?** Count them: `fetchApplication`, then four parallel checks (`checkCredit`, `detectFraud`, `verifyIncome`, `checkBlacklist`), then `aggregateRisk`, then `makeDecision` — seven nodes total.
2. **What happens if `makeDecision` returns `"escalated"`?** The `otherwise` case catches it and routes to `manualReview`.
3. **Can `approveLoan` and `rejectLoan` ever both run?** No — exactly one branch target is selected.
4. **What does the `riskSummary` transform do relative to the branch?** It projects upstream outputs for audit logging. Because it doesn't depend on `makeDecision`, it can resolve independently — but it has no effect on which branch path is taken.

Now try modifying the graph yourself:

- Add a fourth case: `"deferred" -> deferApplication`.
- Declare the new `deferApplication` node with a `DeferApplicationOperator`, depending on `makeDecision`.

- Predict: what happens to the other three target nodes when the decision is `"deferred"`?

5.14 Brain Check

1. What keyword pair opens a conditional routing block? *(`branch on`.)*
2. What type of expression must follow `branch on`? *(A node output path expression like `nodeId.output.field`, or a transform field path like `transformId.field`.)*
3. If a branch has three cases and no `otherwise`, what happens when none of the cases match? *(No target node runs — all targets are skipped.)*
4. Can two branch targets ever execute in the same graph run with a default (exclusive) `branch on`? *(No — exactly one target is selected per exclusive `branch on` evaluation.)*
5. What is the difference between `branch on` and `when`? *(`branch on` is control-plane routing that determines which nodes run. `when` is a data-plane expression that determines which value a field takes.)*
6. *Design question:* You have a graph where two independent features each need a branch. Can a single graph contain two separate `branch on` blocks? If so, how does the engine handle them? *(Yes — each `branch on` is evaluated independently when its source node completes. They do not interact. This is common in complex workflows where multiple decisions happen in parallel paths.)*
7. What keyword do you add to `branch on` to make it an inclusive (OR-split) branch, and how does `otherwise` behave differently? *(Add `inclusive` after the condition expression. In an inclusive branch, `otherwise` fires only when zero cases matched, whereas in an exclusive branch it fires when no explicit case matches.)*

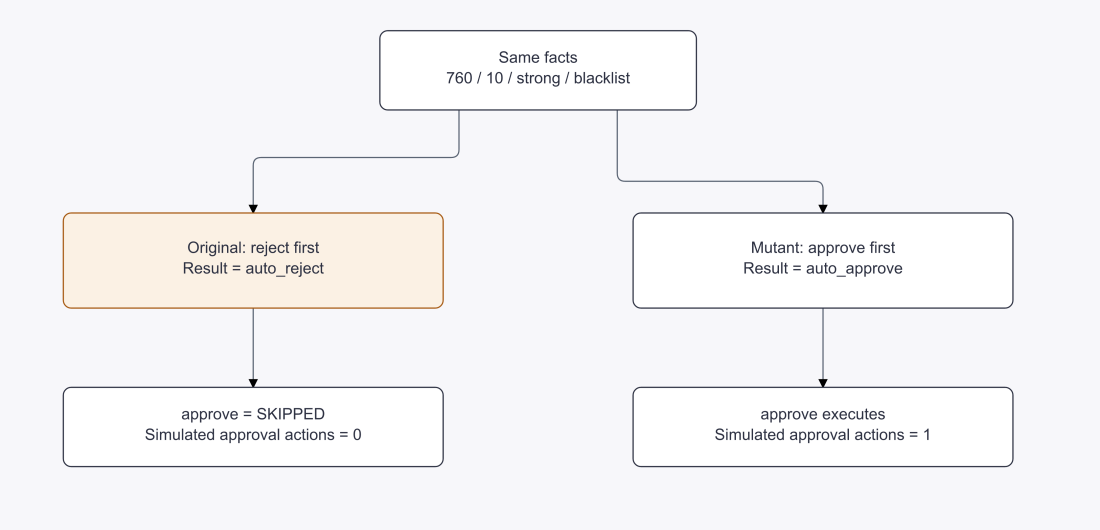
5.15 Lab

1. Open `ch05/ticket-routing.bloge` in your editor.
2. **Add a fourth handler node** called `escalateToManager` with an `EscalateToManagerOperator`. It should depend on `classifyPriority` and take `fetchCustomer.output.id` and `analyzeSentiment.output` as inputs.
3. **Add a new case** to the branch: `"critical" -> escalateToManager`.
4. **Draw the execution flow** for each possible priority value:
   - `"vip"` → which node runs?
   - `"critical"` → which node runs?
   - `"normal"` → which node runs?
   - `"unknown"` → which node runs? (Hint: `otherwise`.)
5. **Bonus:** Rewrite the branch to use only `otherwise` (no explicit cases). What does this mean for the graph's behaviour? Is it useful?

5.16 Experiment acceptance card

- **Expected and observed:** Two XOR cases swap EXECUTED and SKIPPED.
- **Failure and recovery:** Make every condition miss; repair the threshold or otherwise path.
- **Proof boundary:** Proves branch selection for fixed inputs, not every business boundary.
- **Exercise contract:** Approve and reject cases; change one threshold; deliver a status matrix; stop when each case selects one path.

5.17 Coding practice



5.17.1 BP-06 — Expose multi-fact policy

More than three independent conditions is a review prompt for a decision table, not a mandatory threshold. Separate computing a business decision from routing on that decision.

Try one change: For credit 760, fraud 10, strong income, and a blacklist hit, predict the result before opening the Chapter 34 before/after graphs. Four enum routes alone do not require a table.

Reasoning, counterexamples, and limits

5.17.2 BP-07 — Choose hit policy from overlap semantics

Use first for declared priority, unique for expected exclusivity, any for matching outputs, and collect for contributions. Current collect otherwise always appends an item.

Try one change: Replace first with unique for the high-score blacklisted applicant. Explain why a conflict is not an equivalent rejection, then move approval ahead of blacklist and identify the leaked action.

Reasoning, counterexamples, and limits

5.18 Recap

- `branch on` is **control-plane routing** — it decides which downstream nodes execute and which are skipped.
- The condition must be a **node output path** or **transform field path** expression.
- Cases use **type-lenient comparison** and are tested in **declaration order** — the first match wins.
- `otherwise` is the **default case**. Always include it when the set of possible values is not exhaustively known.
- Un-chosen target nodes are marked **`SKIPPED`** — they never execute and produce no output.
- `branch on` (control plane) and `when` (data plane) share syntax style but are **semantically different**. Use `branch on` to route execution; use `when` to compute a value.
- **Inclusive branching** (`branch mode=inclusive on ...`) is an OR-split: all matching branches fire in parallel.

Use it when a condition can match multiple cases simultaneously (e.g., a list of flags). `otherwise` only fires when *zero* cases matched.

5.19 Next Step

In Chapter 6 — Resilience by Design you will learn how to declare timeout, retry, and fallback policies on nodes — keeping your graphs reliable without cluttering your operator code.

5.20 Reference Links

- DSL Specification — formal grammar and compilation rules
- Core Architecture — `Edge.Conditional`, `NodeStatus.SKIPPED`, and engine dispatch
- `ch05/ticket-routing.bloge` — full example source
- `ch05/loan-approval.bloge` — full example source
- `boolean.bloge` — conformance fixture
- `otherwise.bloge` — conformance fixture
- `ch05/multi-flag-approval.bloge` — inclusive branching example

CHAPTER
06

Arc 2 — Maintainable Authoring

Chapter 6 — Resilience by Design

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

Promise: *By the end of this chapter you will know how BLOGE's built-in resilience chain — **retry → timeout → fallback** — protects each node, and how **compensation** lets you undo already-completed work when a graph fails downstream.*

6.1 Learning Goals

1. Explain the resilience evaluation order and *why* retry wraps per-attempt timeout while fallback is the final degradation.
2. Configure `retry`, `timeout`, and `fallback` on a node and predict the runtime behaviour when the operator throws.
3. Choose the right `backoff` strategy (`fixed`, `exponential`, `jitter`) for a given scenario.
4. Declare `compensate` blocks to express best-effort, node-level saga cleanup — and understand why this is *not* a distributed-transaction rollback.

6.2 Prerequisites

- Chapter 5 — Branches That Decide

6.3 Source Examples

File	What it shows
`ch06/error-handling.bloge`	Retry + timeout + fallback on a payment node with a summarise step
`ch21/retry-with-backoff.bloge`	Loop-based retry with custom exponential backoff and carry state
`ch21/order-saga.bloge`	Three-node saga with per-node compensation
`retry-fallback-output.bloge`	Conformance fixture: retry + fallback on a minimal node
`node-compensation.bloge`	Conformance fixture: single-node compensation
`compensation.bloge`	Conformance fixture: two-node compensation chain
ADR-007	Architecture decision record for the Retry → Timeout → Fallback order

File	What it shows

6.4 Why This Matters

In Chapters 1–4 you saw `retry`, `timeout`, and `fallback` on a few nodes. They looked like simple annotations. In reality, the *order* these policies execute in determines whether your graph recovers cleanly or masks real failures.

Consider a payment node that is flaky under load:

- If fallback ran **before** retries were exhausted, you would silently degrade on the first hiccup — charging no one and generating phantom ”manual review” records.
- If a single timeout wrapped **all** retry attempts, one slow attempt could eat the entire budget, leaving no room for the remaining tries.
- If compensation tried to act like a distributed transaction, a single compensation failure would halt cleanup and leave your system half-undone.

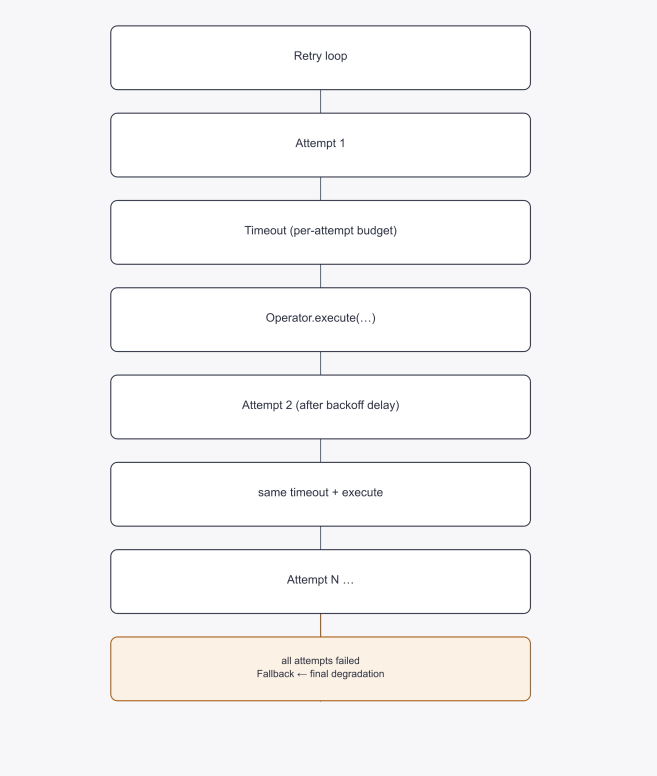
BLOGE eliminates these traps with two clear rules:

1. **Retry wraps per-attempt timeout; fallback is the final catch.**
2. **Compensation is best-effort node-level cleanup, not an atomic rollback.**

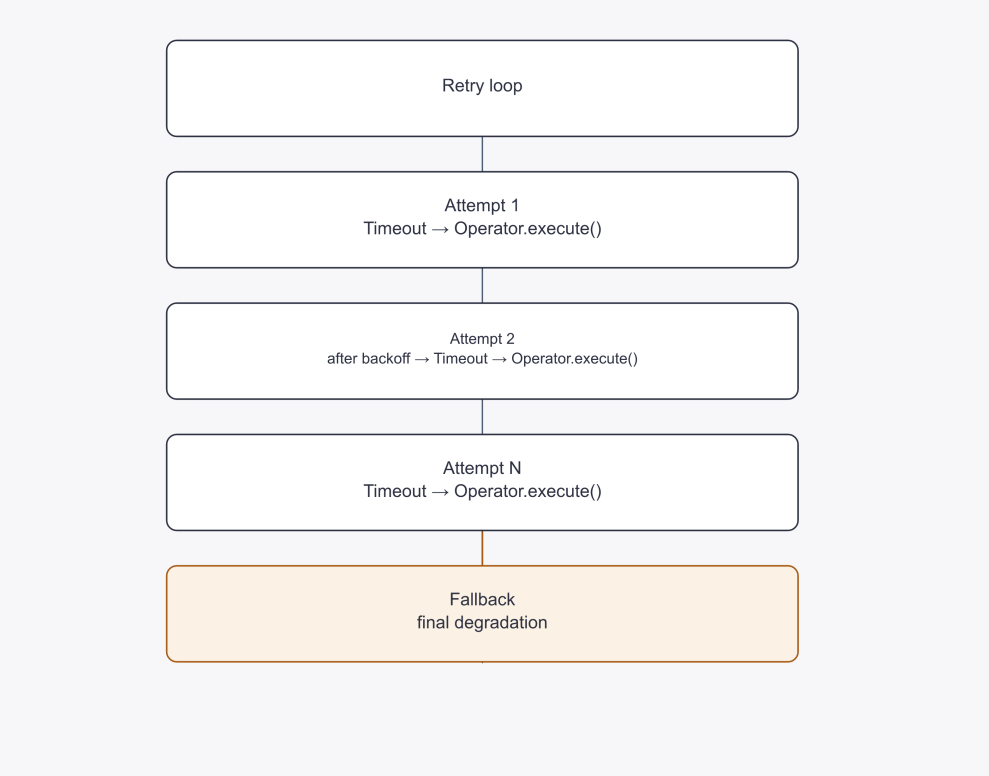
Once you internalise these rules you can declare resilience *next to the node it protects* and trust the engine to apply it correctly.

6.5 Mental Model

6.5.1 The resilience chain



The same chain as a rendered diagram:

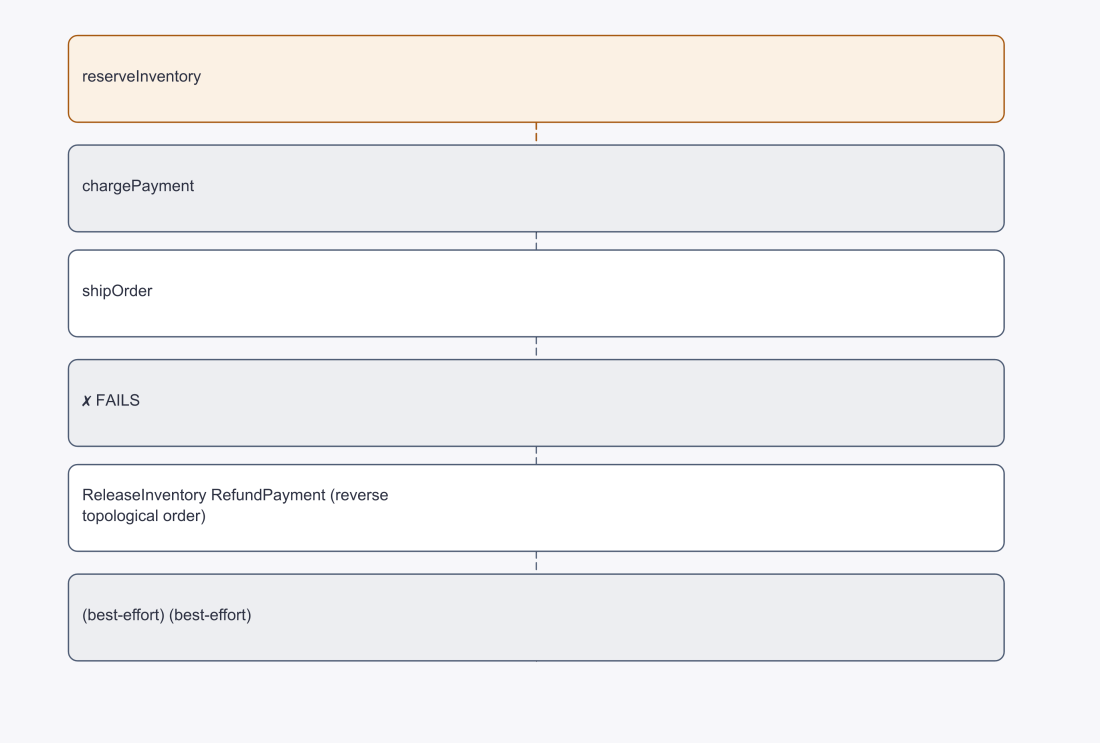


Read the diagram inside-out:

1. **Operator** — your business code runs first.
2. **Timeout** — each *individual attempt* gets its own clock. If it expires, the attempt fails with `OperatorTimeoutException`.
3. **Retry** — on failure, the engine waits the backoff duration and tries again, up to `attempts` times. Each new attempt starts a fresh timeout.
4. **Fallback** — only if *every* attempt has failed (including timeouts) does the engine substitute the fallback value. This is the last resort, not an early exit.

> ***Key insight:*** *Overall wall-clock time can exceed a single* `timeout` *value because each retry attempt gets its own budget. For example,* `timeout = 1s` *with* `attempts: 3` *could take up to 3 × 1 s + backoff delays.*

6.5.2 Compensation — the undo path



When a downstream node fails and the graph cannot continue, the engine walks back through **completed** nodes in reverse topological order and invokes each node's `compensate` operator — if one is declared. Compensation is **best-effort**: if `ReleaseInventory` itself throws, the error is captured in `CompensationResult` and `RefundPayment` still runs. This is saga-style cleanup, not distributed-transaction rollback.

6.6 First Working Example

Here is the full `ch06/error-handling.bloge`:

```
graph errorHandlingShowcase {
  node chargePayment : ChargePaymentOperator {
    input {
      orderId         = ctx.orderId
      amount          = ctx.amount
      simulateFailure = ctx.simulateFailure
    }
    retry = { attempts: 1, backoff: 25ms, strategy: exponential }
    timeout = 1s
    fallback = { approved: false, note: "Gateway unavailable; queued for manual review" }
  }

  node summarizeOutcome : SummarizeOutcomeOperator {
    depends_on = [chargePayment]
    input {
      approved = chargePayment.output.approved
      note     = chargePayment.output.note
    }
    timeout = 1s
  }
```

```
}
```

Walk through the scenario where `ChargePaymentOperator` keeps throwing:

1. **Attempt 1** starts. The operator has up to **1 s** to complete. It throws.
2. The engine waits **25 ms** (exponential backoff from the base), then starts **attempt 2** with a fresh 1 s timeout. It throws again.
3. The configured single retry is exhausted. The engine activates the **fallback**: `{ approved: false, note: "Gateway unavailable; queued for manual review" }`.
4. `summarizeOutcome` receives the fallback output via `chargePayment.output.approved` and `chargePayment.output.note` — the graph completes successfully with a degraded result.

> ***Notice:*** *`summarizeOutcome` has no retry or fallback. If it fails, the graph fails. Resilience is declared only where it is needed.*

The same retry + timeout + fallback configuration in the fluent Java API. The important shape is the real builder chain: lambda-based input assembly, `retry(...)`, `timeout(...)`, then `fallback(...)`.

```
Graph graph = Graph.builder("errorHandlingShowcase")
    .node("chargePayment", new ChargePaymentOperator())
        .input((results, ctx) -> new ChargePaymentInput(
                ctx.get("orderId", String.class),
                ctx.get("amount", BigDecimal.class),
                ctx.get("simulateFailure", Boolean.class)))
        .retry(1, Duration.ofMillis(25), BackoffStrategy.EXPONENTIAL)
        .timeout(Duration.ofSeconds(1))
        .fallback(ex -> new PaymentResult(
                false,
                "Gateway unavailable; queued for manual review"))
    .node("summarizeOutcome", new SummarizeOutcomeOperator())
        .dependsOn("chargePayment")
        .input((results, ctx) -> {
            PaymentResult payment = results.get("chargePayment", PaymentResult.class);
            return new SummaryInput(payment.approved(), payment.note());
        })
        .timeout(Duration.ofSeconds(1))
    .build();
```

Notice the builder mirrors the DSL declaration order: retry, then timeout, then fallback. The engine applies them in the same Retry → Timeout → Fallback chain regardless of how the graph is constructed.

6.7 The Failure That Looks Safe: Charged After Timeout

Suppose `ChargeCardOperator` sends a request with `idempotencyKey=order-42`. The payment service commits the charge at 430 ms, but its response is delayed. At 500 ms the graph times out and marks the node failed. ”The node failed” and ”the charge did not happen” are different facts.



6.7.1 Separate execution control from effect truth

Observation	What it establishes	What it does not establish
Node status is `FAILED` with timeout	The engine stopped waiting within its policy	The remote system rolled back
Payment ledger contains `order-42`	The charge effect exists	The graph received the response
A retry returns the same charge id	The provider honoured the idempotency key	Every provider action is generally safe to retry

The RC1 foundation probe makes the smallest version of this problem observable: the fake ledger increments before the Operator response is delayed; the node then reaches `FAILED`, while the ledger remains at `1`. This does not emulate a payment network. It proves the narrower point that engine timeout is not evidence of effect absence.

6.7.2 Design the recovery conversation

For effectful Operators, define three things before adding retry:

1. A stable idempotency key derived from the business operation, not the attempt number.
2. A reconciliation read that can answer ”did this effect happen?” after an ambiguous response.
3. A compensation action with its own failure handling when the business truly needs an undo.

Retry is safe only inside that contract. Compensation is a new effect, not a time machine, and can fail independently. The engine controls attempts and timing; the integration owns the truth of the external ledger.

6.8 Break It Apart

6.8.1 Retry and backoff strategies

```
retry = { attempts: 3, backoff: 200ms, strategy: exponential }
```

Field	Meaning
`attempts`	Maximum number of retries *after* the initial execution (so `attempts: 3` means up to 4 total executions)
`backoff`	Base delay between retries
`strategy`	How the delay grows: `fixed`, `exponential`, or `jitter`

The three strategies:

Strategy	Behaviour	Good for
`fixed`	Same delay every time	Simple cases, predictable latency
`exponential`	Delay doubles each retry (25 ms → 50 ms → 100 ms …)	Back-pressure on overloaded services
`jitter`	Exponential with random perturbation	Avoiding thundering-herd retries from parallel nodes

6.8.2 Fallback

```
fallback = { approved: false, reason: "credit service unavailable" }
```

A fallback is a **static value** that replaces the node's output when all retry attempts and timeouts have been exhausted. It is the final degradation — the graph continues, but with substitute data.

The conformance fixture `retry-fallback-output.bloge` shows the minimal form:

```
graph g {
  node a : Op {
    retry = { attempts: 3, backoff: 200ms, strategy: exponential }
    fallback = { score: 0 }
    output {
      result: String
    }
  }
}
```

That fixture is intentionally tiny and focuses on fallback syntax. In a real workflow, make sure the fallback payload matches the node's declared output contract.

6.8.3 Compensation

Compensation is declared *inside* the node that produced the side effect:

```
node reserveInventory : ReserveInventoryOperator {
  output {
    reservationId: String
  }

  compensate : ReleaseInventoryOperator {
    input {
      reservationId = reserveInventory.output.reservationId
    }
  }
}
```

(From `ch21/order-saga.bloge`.)

Key rules:

- **Only completed nodes are compensated.** If `chargePayment` never finished, there is nothing to undo.
- **Reverse topological order.** The engine compensates the latest completed node first and works backwards.
- **Best-effort.** A compensation failure is recorded in `CompensationResult` but does not stop subsequent compensations.
- **Not a distributed transaction.** There is no two-phase commit. Each `compensate` block runs independently. Design your compensation operators to be idempotent.

The conformance fixture `compensation.bloge` shows two nodes with compensation chaining:

```
graph g {
  node pay : PaymentOp {
    input { amount = ctx.amount }
    compensate : ReversePayment {
      input { txId = pay.output.transactionId }
    }
  }
  node ship : ShipOp {
    depends_on = [pay]
    compensate : CancelShipment {
      input { trackingId = ship.output.trackingId }
    }
  }
}
```

If `ship` fails, the engine compensates `pay` (the only *completed* node) by running `ReversePayment` with the original transaction ID.

6.9 Common Trap

6.9.1 [X] Treating compensation as a distributed-transaction rollback

```
// WRONG mental model — expecting atomic all-or-nothing
node reserveInventory : ReserveInventoryOperator {
  compensate : ReleaseInventoryOperator {
    input { reservationId = reserveInventory.output.reservationId }
  }
}

node chargePayment : ChargePaymentOperator {
  depends_on = [reserveInventory]
  compensate : RefundPaymentOperator {
    input { chargeId = chargePayment.output.chargeId }
  }
}
```

If `RefundPaymentOperator` throws, the reservation is *still* released — the engine does not stop compensating. And if `ReleaseInventoryOperator` also throws, you end up with *two* compensation failures, not a clean rollback.

Design compensation operators to be idempotent and to tolerate partial failure. Log aggressively. Use the `CompensationResult` list in the `GraphResult` to detect and alert on compensation failures.

6.10 What Goes Wrong

6.10.1 All retries exhausted, no fallback

```
node chargePayment : ChargePaymentOperator {
  retry = { attempts: 3, backoff: 200ms, strategy: exponential }
  timeout = 1s
  // No fallback declared
}
```

If the operator fails on every attempt:

```
GraphResult {
  status: FAILED,
  failedNode: "chargePayment",
  error: "chargePayment failed after the initial attempt
    and all configured retries"
}
```

The graph stops. Every downstream node is marked `NOT_REACHED`.

Fix: Add a `fallback` when graceful degradation is acceptable, or let the failure propagate when it is genuinely unrecoverable.

6.11 Guided Rewrite

Start from the order-saga graph and add resilience:

```
graph orderSaga {
  node reserveInventory : ReserveInventoryOperator {
    timeout = 3s
    retry = { attempts: 2, backoff: 100ms, strategy: jitter }
    output {
      reservationId: String
    }

    compensate : ReleaseInventoryOperator {
      input {
        reservationId = reserveInventory.output.reservationId
      }
    }
  }

  node chargePayment : ChargePaymentOperator {
    depends_on = [reserveInventory]
    timeout = 5s
    retry = { attempts: 3, backoff: 200ms, strategy: exponential }
    fallback = { chargeId: null, status: "payment_deferred" }
    output {
      chargeId: String
    }

    compensate : RefundPaymentOperator {
      input {
        chargeId = chargePayment.output.chargeId
      }
    }
  }

  node shipOrder : ShipOrderOperator {
    depends_on = [chargePayment]
```

```
    timeout = 10s
    input {
      failShipping = ctx.failShipping
    }
  }
}
```

Questions to consider:

1. If `chargePayment` fails after all 3 retries, what happens? *(The fallback value `{ chargeId: null, status: "payment_deferred" }` is used. `shipOrder` receives it. No compensation runs because no node has failed the graph — the fallback kept it alive.)*
2. If `shipOrder` then fails (no fallback), which compensation operators run? *(Both `RefundPaymentOperator` and `ReleaseInventoryOperator`, in that order — reverse topological. `chargePayment` completed with the fallback value, and `reserveInventory` completed normally.)*
3. What is the worst-case wall-clock time for `chargePayment` alone? *(Initial attempt: 5 s + 3 retries × (backoff + 5 s). With exponential backoff from 200 ms: ≈ 5 + 0.2 + 5 + 0.4 + 5 + 0.8 + 5 = ~21.4 s.)*

6.12 Brain Check

1. What is the resilience evaluation order in BLOGE? *(Retry → Timeout (per attempt) → Fallback.)*
2. Why does each retry attempt get its own timeout rather than sharing one global timeout? *(So that one slow attempt cannot consume the entire budget, leaving no time for subsequent retries. See ADR-007.)*
3. When does a fallback activate? *(Only after all retry attempts — including their individual timeouts — have been exhausted. It is the final degradation.)*
4. Does a compensation failure stop the engine from compensating earlier nodes? *(No. Compensation is best-effort. Each failure is captured in a `CompensationResult` and the engine continues to the next node.)*
5. What order does the engine compensate nodes in? *(Reverse topological order, only for nodes whose primary execution completed successfully.)*
6. *Design question:* Your graph calls three external payment gateways in a fan-out. Each has retry + fallback. If all three fallback, the downstream aggregation node receives three degraded results. How should you design the aggregation node to handle a mix of real and fallback data? *(The aggregation operator should inspect a flag or sentinel value in each input — for example, a `"status": "degraded"` field from the fallback — and decide whether to proceed with partial data or escalate. The graph-level design decision is whether "all degraded" should still complete or should fail the graph.)*

6.13 Lab

1. Open `ch06/error-handling.bloge`.
   - Change the retry strategy from `exponential` to `jitter` and increase `attempts` to `3`.
   - Predict: if every attempt takes exactly 800 ms before failing, will the 1 s timeout ever fire? *(No — 800 ms < 1 s, so each attempt fails from the operator exception, not from timeout.)*
2. Open `ch21/order-saga.bloge`.
   - Add `timeout = 2s` and `retry = { attempts: 1, backoff: 50ms, strategy: fixed }` to `chargePayment`.
   - Add a `fallback` to `reserveInventory` that returns `{ reservationId: "NONE" }`.
   - Trace what happens when `reserveInventory`'s operator always throws: does `ReleaseInventoryOperator` run? *(No — the fallback kept `reserveInventory` alive, so the graph continues. Compensation only triggers if the graph ultimately fails.)*
3. Write a new graph with three independent service calls, each with different backoff strategies (`fixed`, `exponential`,

`jitter`). Add a fan-in node that requires all three. Give two of the three a fallback. Predict: under what conditions does the graph fail vs. degrade gracefully?

6.14 Experiment acceptance card

- **Expected and observed:** The graph can fall back after timeout while the external charge may already exist.
- **Failure and recovery:** Delay the effect beyond timeout; recover with an idempotency key and reconciliation.
- **Proof boundary:** Proves node completion and external effect are separate facts, not exactly-once.
- **Exercise contract:** Charge probe; change only response delay; deliver graph status and ledger count; stop when both facts are separate.

6.15 Coding practice

6.15.1 BP-09 — Give retries one primary owner

Coordinate graph, Operator, and SDK retries against one budget. Keep business request keys stable across attempts; idempotency declarations do not perform provider deduplication.

Try one change: Count attempts when two layers each allow three, then execute the workshop effect-key test. Keep durable recovery failures separate from call retry counts.

Reasoning, counterexamples, and limits

6.15.2 BP-10 — Preserve unknown and degraded outcomes

A timeout stops waiting, not necessarily the remote action. Mark fallback semantics explicitly and compensate only confirmed, reversible actions using the original receipt.

Try one change: A charge succeeds but its response is lost. Explain the reconciliation step, then compare it with a credit lookup that may safely degrade to manual review.

Reasoning, counterexamples, and limits

6.16 Recap

- BLOGE applies resilience in a strict order: **Retry → Timeout → Fallback**. Each retry attempt gets its own timeout budget; fallback is the final degradation after all attempts are exhausted.
- Three **backoff strategies** control delay between retries: `fixed`, `exponential`, and `jitter`.
- **Fallback** is a static substitute value — the graph continues with degraded data instead of failing.
- **Compensation** (`compensate` blocks) provides best-effort, node-level saga cleanup. The engine runs compensation in reverse topological order, only for completed nodes, and captures failures without halting.
- Compensation is **not** a distributed-transaction rollback. Design compensation operators to be idempotent.
- Resilience is declared *on the node*, not buried in business code — the engine owns the retry loop, the timeout clock, and the compensation walk.

6.17 Next Step

In Chapter 7 — Designing Good Operators you will learn how to write operators that are testable, composable, and play well with the resilience and scheduling features you have seen so far.

6.18 Reference Links

- ADR-007 — Resilience Order — architecture decision for Retry → Timeout → Fallback

- Core Architecture — engine internals and scheduling design
- DSL Specification — resilience and compensation grammar
- `ch06/error-handling.bloge` — full example source
- `ch21/retry-with-backoff.bloge` — loop-based retry example
- `ch21/order-saga.bloge` — saga compensation example
- `retry-fallback-output.bloge` — conformance fixture
- `node-compensation.bloge` — conformance fixture
- `compensation.bloge` — conformance fixture

CHAPTER
07

Arc 2 — Maintainable Authoring

Chapter 7 — Designing Good Operators

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

Promise: *By the end of this chapter you will know how to decide what belongs inside an operator, how to declare its schema and behavioural contracts, and how to avoid the most common design mistake — turning every piece of logic into a node.*

7.1 Learning Goals

1. Apply the **Single Capability Principle** to decide whether a piece of logic deserves to be an operator or should stay in the orchestration layer.
2. Classify an operator into the three-layer reuse pyramid: **Infrastructure → Capability → Domain**.
3. Declare **input/output schemas** so the compiler, tooling, and downstream nodes can validate data more safely.
4. Set the **idempotency** and **side-effect** contracts that help the framework make safer retry and scheduling decisions.
5. Recognise — and eliminate — trivial data-shaping operators that should be transforms or input-binding expressions.

7.2 Prerequisites

- Chapter 4 — Data That Flows (especially transforms and the data-transformation layer model)
- Chapter 6 — Resilience by Design for how node contracts influence retry and timeout choices

7.3 Source Examples

File	What it shows
`Operator.java`	The core `@FunctionalInterface` — `execute`, `idempotency`, `sideEffectType`
`OperatorContext.java`	Read-only record the engine passes to every operator
`SchemaAware.java`	Optional interface for explicit input/output schema declarations
`Idempotency.java`	`IDEMPOTENT`, `NOT_IDEMPOTENT`, `UNKNOWN`

File	What it shows
`SideEffectType.java`	`READ_ONLY`, `WRITE`, `EXTERNAL_CALL`, `MIXED`
`OperatorMeta.java`	Annotation for layer, tags, version, owner
`OperatorLayer.java`	`INFRASTRUCTURE`, `CAPABILITY`, `DOMAIN`
`Operator Design Specification`	Full normative spec (eight chapters)

7.4 Why This Matters

The previous chapters taught you how to **build graphs** and **move data**. But the quality of a BLOGE system ultimately depends on the quality of its **operators** — the building blocks every graph is made of.

Get operator boundaries wrong and you'll see:

- **Bloated graphs** where half the nodes are one-line field renames that could be a transform.
- **Monolith operators** that bundle three independent I/O calls and are impossible to retry or observe individually.
- **Opaque contracts** where nobody knows what an operator's output looks like until they read its source.

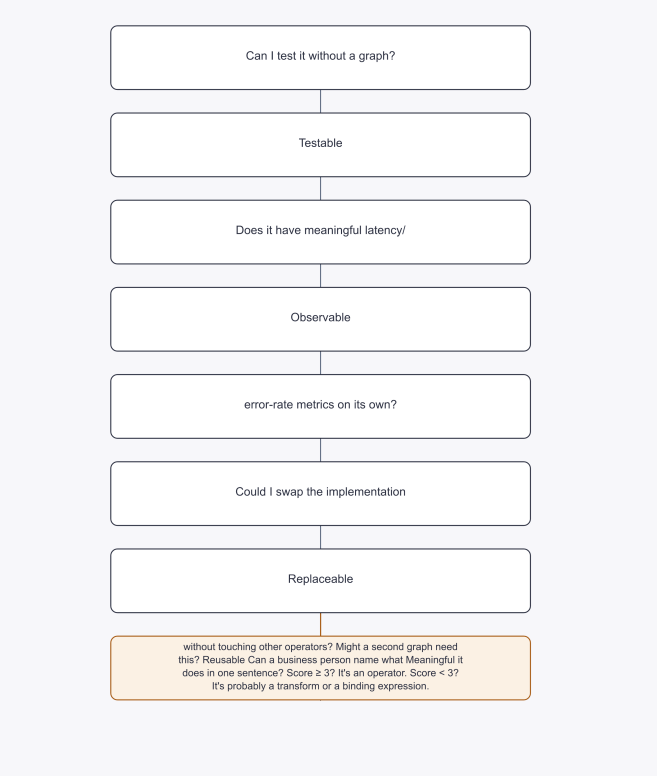
Get them right and you unlock:

- **Reuse** — a well-scoped operator can serve dozens of graphs.
- **Observability** — each operator has its own latency, error-rate, and retry metrics.
- **Safe evolution** — typed schemas make breaking changes visible at compile time, not at 2 AM.

7.5 Mental Model

7.5.1 The Single Capability Principle

One operator should encapsulate **one independently measurable business capability**. Ask five questions:



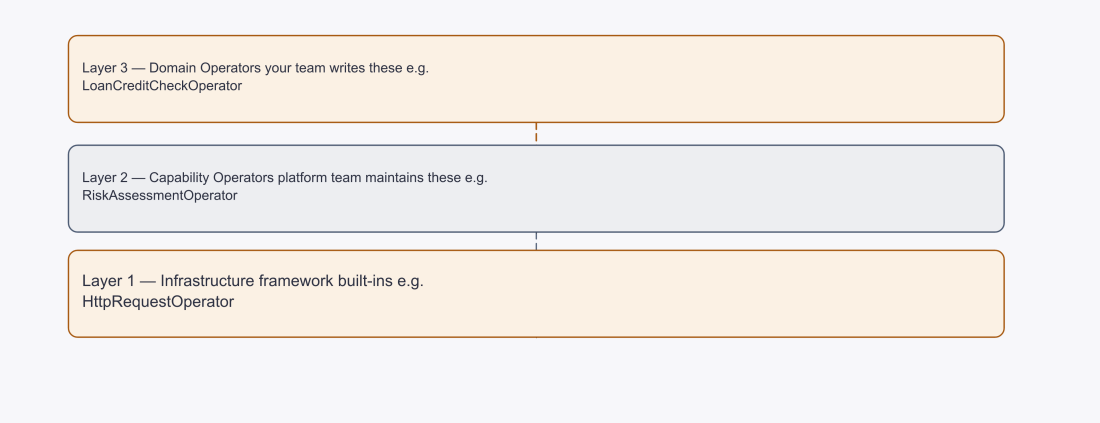
7.5.2 The four layers of data transformation

Not every piece of logic belongs in an operator. BLOGE offers four layers, each with different cost:

Layer	Tool	Cost	When to use
Routing	`input { userId = fetchUser.output.id }`	Zero	Passing or renaming a field
Light transform	`input { name = concat(a.output.first, " ", a.output.last) }`	Zero	Concatenation, type conversion, null filling
Structural adapt	`transform orderSummary { … }`	Zero (virtual node)	Multi-field reshaping reused by several downstream nodes
Business logic	`node … : Operator { … }`	Full (scheduled, timed, retriable)	Domain rules, I/O, side effects

Rule of thumb: If removing the logic only breaks *data format compatibility* — use a transform. If removing it changes *business meaning* — it must be an operator.

7.5.3 The three-layer reuse pyramid



Each layer is tagged via `@OperatorMeta(layer = …)` using the `OperatorLayer` enum (`INFRASTRUCTURE`, `CAPABILITY`, `DOMAIN`). Use it to make the operator's reuse scope explicit in code and documentation.

7.6 First Working Example

Consider an operator that fetches a user from an external service. Declaring typed I/O, schema, and behavioural contracts makes its framework boundary explicit.

7.6.1 Step 1 — Define typed input and output records

```
public record FetchUserInput(String userId) {
    public FetchUserInput {
        if (userId == null || userId.isBlank())
            throw new IllegalArgumentException("userId must not be blank");
    }
}

public record FetchUserOutput(String id, String name, String email, int vipLevel) {}
```

Records give you immutability, validation in the compact constructor, and automatic schema introspection via `SchemaIntrospector`.

7.6.2 Step 2 — Implement the operator

The 43-line implementation is kept whole so metadata, the effect-bearing call, idempotency, side-effect classification, and both schemas can be reviewed as one Operator contract.

```
@OperatorMeta(
    layer = OperatorLayer.INFRASTRUCTURE,
    tags  = {"user", "query"},
    description = "Fetch a user by ID from the user service",
    owner = "user-platform",
    since = "1.0.0"
)
public class FetchUserOperator
        implements Operator<FetchUserInput, FetchUserOutput>, SchemaAware {

    private final UserServiceClient client;

    public FetchUserOperator(UserServiceClient client) {
```

```
        this.client = client;
    }

    @Override
    public FetchUserOutput execute(FetchUserInput input, OperatorContext ctx)
            throws Exception {
        // Business data comes through `input`, not through `ctx`.
        return client.fetchUser(input.userId());
    }

    @Override
    public Idempotency idempotency() {
        return Idempotency.IDEMPOTENT;          // safe to retry
    }

    @Override
    public SideEffectType sideEffectType() {
        return SideEffectType.EXTERNAL_CALL;    // crosses a service boundary
    }

    @Override
    public SchemaDescriptor inputSchema() {
        return SchemaIntrospector.introspect(FetchUserInput.class);
    }

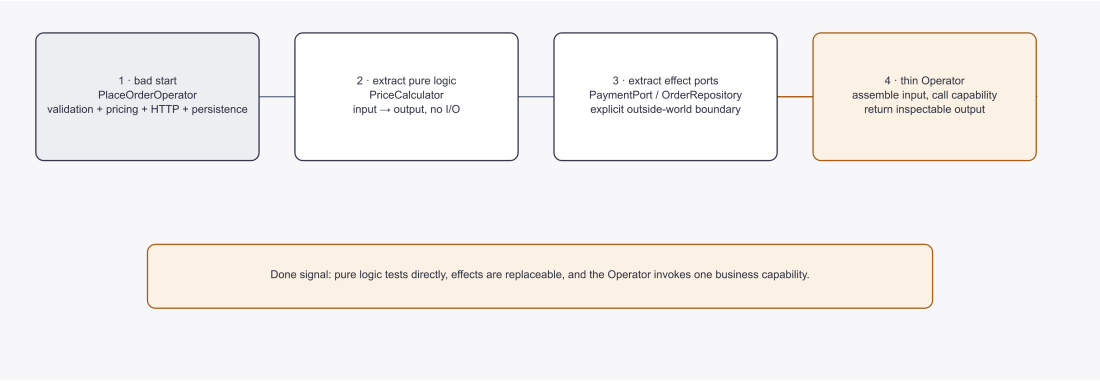
    @Override
    public SchemaDescriptor outputSchema() {
        return SchemaIntrospector.introspect(FetchUserOutput.class);
    }
}
```

Notice four things:

1. **`@FunctionalInterface`** — the core contract is a single method: `O execute(I input, OperatorContext ctx)`. No abstract base class to extend, no lifecycle hooks to learn first.
2. **Typed generics** — `Operator<FetchUserInput, FetchUserOutput>` gives the compiler and the engine a concrete type pair. The framework auto-introspects the generics to derive a `StructuredSchema` even if you don't implement `SchemaAware`.
3. **Behavioural defaults** — `idempotency()` returns `IDEMPOTENT`, so the framework can treat retry as safe. `sideEffectType()` returns `EXTERNAL_CALL`, signalling that the operator crosses a service boundary and should be paired with explicit resilience settings in the graph.
4. **`OperatorContext` is read-only metadata** — it carries `nodeId`, `graphName`, `retryAttempt`, `executionId`, and `timeSource`. All *business* data must arrive through `input`.

7.7 Refactor One Bad Operator in Four Cuts

The fastest way to recognise a good Operator is to repair a bad one. Start with `PlaceOrderOperator`: it validates fields, calculates discounts, calls payment, writes the order, sends mail, and translates every exception. Its name describes a workflow, not one capability.



7.7.1 Cut 1 and 2 — make pure decisions ordinary code

Extract validation and pricing into functions or domain objects with explicit inputs and outputs. They do not need `OperatorContext`, retry, or a node. A table-driven unit test can now cover discount boundaries without starting a graph.

7.7.2 Cut 3 and 4 — expose effects, then keep the adapter thin

Introduce narrow ports such as `PaymentPort` and `OrderRepository`. The final Operator assembles typed input, invokes one business capability, and returns an inspectable result:

Layer	Owns	Test seam
Pure domain logic	Validation and price decisions	Inputs → outputs
Effect port	Payment or persistence protocol	Fake/controlled port plus invocation evidence
Operator adapter	Graph input/output and capability call	`GraphTestRunner` node result
Graph	Ordering, branch, retry, timeout	Scenario and status assertions

The finish line is not a small class. It is a clean failure boundary: a pricing bug fails a pure test; a payment protocol bug fails the port contract; a wrong dependency fails the graph scenario. `SchemaAware`, metadata, and context APIs then describe this capability—they do not rescue a capability whose responsibilities are still mixed.

7.8 Break It Apart

7.8.1 The `Operator` interface

From `Operator.java`:

```
@FunctionalInterface
public interface Operator<I, O> {
    O execute(I input, OperatorContext ctx) throws Exception;

    default Idempotency idempotency()     { return Idempotency.UNKNOWN; }
    default SideEffectType sideEffectType() { return SideEffectType.MIXED; }
}
```

Member	Purpose
`execute(I, OperatorContext)`	The one method you must implement — receives assembled input and returns output
`idempotency()`	Tells the engine whether retry is safe (`IDEMPOTENT`), forbidden (`NOT_IDEMPOTENT`), or unspecified (`UNKNOWN`)

Member	Purpose
`sideEffectType()`	Tells the framework what kind of effects to expect so scheduling and resilience decisions stay explicit

7.8.2 `OperatorContext` — what the engine tells you

From `OperatorContext.java`:

```
public record OperatorContext(
    String nodeId,
    String graphName,
    GraphContext graphContext,   // request-scoped; writes isolated per node
    int retryAttempt,           // 0 = first execution
    String executionId,
    TimeSource timeSource       // never null; defaults to SystemTimeSource.INSTANCE
) {}
```

The `graphContext` is available for **request-level metadata** (trace ID, tenant ID). It is **not** the place for business data — that must flow through the typed `I` input.

7.8.3 `timeSource` — testable time for operators

The `timeSource` field gives every operator access to the engine's clock without hard-coding `System.currentTimeMillis()` or `Instant.now()`.

Why it matters: When an operator reads the wall clock directly, tests cannot control what time it sees. A retry-delay assertion that depends on real time is slow and flaky. With `timeSource`, the test harness injects a `ManualTimeSource` and the operator gets deterministic time for free.

Rule of thumb: Whenever your operator needs the current time, call `ctx.timeSource().now()` instead of `Instant.now()`.

```
@Override
public AuditOutput execute(AuditInput input, OperatorContext ctx) throws Exception {
    Instant timestamp = ctx.timeSource().now();   // [OK] testable
    // Instant timestamp = Instant.now();         // [X] untestable
    return auditService.record(input.action(), timestamp);
}
```

In production the engine supplies `SystemTimeSource.INSTANCE`, which delegates to `Instant.now()` and `Thread.sleep()`. In tests you swap it with `ManualTimeSource`:

```
var manualTime = new ManualTimeSource(Instant.parse("2025-01-15T10:00:00Z"));

var ctx = OperatorContext.builder()
        .nodeId("audit")
        .graphName("compliance")
        .graphContext(new GraphContext())
        .timeSource(manualTime)        // inject test clock
        .build();

var result = new AuditOperator(auditService).execute(input, ctx);

// assert the operator used the injected time, not the wall clock
assertEquals(Instant.parse("2025-01-15T10:00:00Z"), result.recordedAt());
```

This is the same `ManualTimeSource` used by `TestGraphEngine` in Chapter 18. The difference is that here you use it at the **operator** level for unit tests, while Chapter 18 uses it at the **graph** level for integration tests.

7.8.4 `SchemaAware` — explicit schema declarations

From `SchemaAware.java`:

```
public interface SchemaAware {
    default SchemaDescriptor inputSchema()  { return OpaqueSchema.INSTANCE; }
    default SchemaDescriptor outputSchema() { return OpaqueSchema.INSTANCE; }
}
```

Implementing `SchemaAware` is optional. If you skip it, the framework falls back to auto-introspection of the generic type parameters `I` and `O`. But explicit declaration is preferred because:

- It survives type erasure when generics resolve to `Map<String, Object>`.
- It lets you annotate individual fields with descriptions and constraints.
- It enables compile-time path validation in downstream input bindings.

At the Java operator layer, `DefaultOperatorRegistry` first checks `SchemaAware`; if you do not implement it, it falls back to `SchemaIntrospector` over the operator's generic type parameters.

7.8.5 `@OperatorMeta` — catalogue metadata

From `OperatorMeta.java`:

```
public @interface OperatorMeta {
    OperatorLayer layer() default OperatorLayer.DOMAIN;
    String[] tags()       default {};
    String version()      default "";
    String description()  default "";
    String owner()        default "";
    String since()        default "";

    // LLM tool metadata — consumed by bloge-agent-ext (Ch 17)
    String promptHint()             default "";
    String usageExample()           default "";
    String constraintsDescription() default "";
}
```

Every operator **should** declare `layer`, `description`, and `owner` at a minimum. Even when runtime behavior comes from the operator contract itself, this metadata keeps intent and ownership visible to readers and tooling.

The last three fields are the **LLM tool description**. When an operator is exposed as a tool to an `agent` node (Ch 17), `bloge-agent-ext` reads them to build the function spec sent to the model:

- `promptHint` — one-line description the LLM sees in tool listings.
- `usageExample` — a concrete invocation example to anchor the model.
- `constraintsDescription` — preconditions, side effects, or rate limits.

These also flow into `operator-metadata.json` (Ch 9) so Studio shows the same hints in the operator palette.

7.9 Common Trap

7.9.1 [X] Creating an operator for pure data reshaping

```
// DON'T — this operator does nothing that a transform can't do for free
public class FormatOrderSummaryOperator
        implements Operator<Map<String, Object>, Map<String, Object>> {

    @Override
    public Map<String, Object> execute(Map<String, Object> input,
                                       OperatorContext ctx) {
```

```
        return Map.of(
            "customerName", input.get("name"),
            "total",        input.get("price")
        );
    }
}
```

This operator is **independently testable** (barely), but it has no meaningful metrics, no side effects, no domain knowledge, and no reuse potential. It fails four out of five checks from the Single Capability Principle.

Use a transform instead:

```
transform orderSummary {
  customerName = fetchUser.output.name
  total        = calcPrice.output.total
}
```

The transform adds no Operator scheduling cost — no operator invocation, no timeout tracking, no retry overhead — and it keeps the reshape visible in the graph itself.

How to spot the trap in existing graphs:

- The operator's `execute` method contains no `if`, no I/O, no external call.
- The operator's class has no constructor dependencies (no service client, no repository).
- The operator could be replaced by a `return input;` pass-through without changing business outcomes.

If any of these are true, **delete the operator and use a transform or an input-binding expression**.

7.9.2 [X] Encoding a decision table as a plain operator

Sometimes an operator looks like business logic, but the implementation is really just a table of conditions mapping to output values — no I/O, no retry, no state. That is not an independent capability; it is a policy table hiding inside Java code.

Reach for a `decision_table` when:

- The logic is a **pure function** of its inputs (no side effects, no external calls).
- The conditions are **enumerable** — you can list all the cases, and the compiler or lint can warn you when the list is incomplete.
- **Audit trail** matters — you want a lint rule (`decision-table/missing-otherwise`) and structured error codes at runtime.

Keep it in an `Operator` when:

- The node performs **I/O** (HTTP call, database query, message publish) or has side effects.
- You need **retry, timeout, or fallback** policies managed by the framework.
- The logic involves **mutable state** or coordination that cannot be expressed in a rule predicate.

See Appendix F — Decision Table for the full syntax reference and all error codes.

7.10 Guided Rewrite

Here is a graph with two design problems — one operator that is too trivial and one that is too broad:

```
graph invoiceProcess {

  node fetchOrder : FetchOrderOperator {
    input { orderId = ctx.orderId }
    timeout = 3s
  }
```

```
  // Problem 1: trivial data reshape — should be a transform
  node formatAddress : FormatAddressOperator {
    input {
      street = fetchOrder.output.address.street
      city   = fetchOrder.output.address.city
      zip    = fetchOrder.output.address.zip
    }
  }

  // Problem 2: monolith — bundles payment + notification into one operator
  node processAndNotify : ProcessAndNotifyOperator {
    depends_on = [fetchOrder, formatAddress]
    input {
      order   = fetchOrder.output
      address = formatAddress.output
    }
    timeout = 10s
  }
}
```

Rewrite:

```
graph invoiceProcess {

  node fetchOrder : FetchOrderOperator {
    input { orderId = ctx.orderId }
    timeout = 3s
  }

  // Fix 1: replace the trivial operator with a transform
  transform formattedAddress {
    street = fetchOrder.output.address.street
    city   = fetchOrder.output.address.city
    zip    = fetchOrder.output.address.zip
    full   = concat(fetchOrder.output.address.street, ", ",
                    fetchOrder.output.address.city, " ",
                    fetchOrder.output.address.zip)
  }

  // Fix 2: split the monolith into two independent operators
  node processPayment : ProcessPaymentOperator {
    depends_on = [fetchOrder]
    input {
      order   = fetchOrder.output
      address = formattedAddress.full
    }
    timeout = 5s
    retry   = { attempts: 2, backoff: 500ms, strategy: exponential }
  }

  node sendNotification : SendNotificationOperator {
    depends_on = [fetchOrder]
    input {
      email   = fetchOrder.output.customerEmail
      orderId = fetchOrder.output.id
    }
    timeout = 3s
```

```
    fallback = { sent: false, reason: "notification service unavailable" }
  }
}
```

What improved:

Before	After	Why it matters
`FormatAddressOperator` (full node)	`transform formattedAddress`	Zero scheduling cost; no need for timeout/retry on a pure reshape
`ProcessAndNotifyOperator` (one fat node)	`processPayment` + `send Notification`	Independent SLAs, independent retry policies, and they can run **in parallel**

7.11 Brain Check

1. Name three of the five checks in the Single Capability Principle. *(Independently testable, independently observable, independently replaceable, independently reusable, has clear domain semantics.)*
2. An operator's `execute` method reads `ctx.graphContext().get("items")` to get its business data. What's wrong? *(Business data must arrive through the typed input `I`, not through the graph context. The graph context is for request-level metadata like trace IDs.)*
3. You have an operator that concatenates two strings. Should it exist? *(No — string concatenation is a light transform. Use `concat(a.output.first, " ", a.output.last)` in an input binding or transform.)*
4. What does `SideEffectType.EXTERNAL_CALL` tell the framework? *(That the operator crosses an external-service boundary, so readers and integrations should treat it as a side-effecting call and pair it with explicit timeout/retry policy in the graph.)*
5. If an operator does not implement `SchemaAware`, how does the framework derive its schema? *(By introspecting the generic type parameters `I` and `O` via `SchemaIntrospector`.)*

7.12 Lab

1. **Audit an existing graph.** Open `order-process-v5.bloge`. For each node, apply the five-question test. Are there any nodes that should be transforms? Write down your reasoning.
2. **Design a new operator from scratch.** Imagine a `CreditCheckOperator` that calls an external credit-scoring API.
   - Define `CreditCheckInput` and `CreditCheckOutput` as Java records.
   - Implement `Operator<CreditCheckInput, CreditCheckOutput>`.
   - Implement `SchemaAware` with explicit schemas.
   - Override `idempotency()` → `IDEMPOTENT` (read-only query).
   - Override `sideEffectType()` → `EXTERNAL_CALL`.
   - Annotate with `@OperatorMeta(layer = CAPABILITY, ...)`.
3. **Schema validation check.** After writing your records, use `SchemaIntrospector.introspect(CreditCheckOutput.class)` in a unit test and verify the returned `StructuredSchema` has the fields you expect.
4. **Refactor a bad operator.** Write a `FormatCurrencyOperator` that takes a `Number` and returns a `String`. Then delete it and replace it with a transform or an expression at the binding site. Which approach is simpler?

7.13 Experiment acceptance card

- **Expected and observed:** Splitting an operator into logic, port, adapter, and graph policy creates distinct failure seams.
- **Failure and recovery:** Return a transport error from the adapter; replace only the adapter or fallback.
- **Proof boundary:** Proves testable responsibility boundaries, not external reliability.
- **Exercise contract:** Bad service; cut one layer at a time; deliver a responsibility map and failure test; stop when one owner remains.

7.14 Coding practice

7.14.1 BP-02 — Name bounded computation

Extract an ordinary helper when a calculation needs a clear name. Publish a typed function only when graph authors need it; keep business inputs explicit and test metadata against behavior.

Try one change: Use the Chapter 34 policy-code function to compare the same input under root and Turkish locales. Decide whether trim alone satisfies the contract.

Reasoning, counterexamples, and limits

7.14.2 BP-03 — Expose independent execution responsibility

Use an Operator when a capability needs its own retry, timeout, replacement, observations, or capacity boundary. Pure expensive computation can qualify; a short HTTP call can qualify too.

Try one change: Move a hidden credit-service query out of input evaluation. Identify Operator invocation evidence and external-effect evidence separately.

Reasoning, counterexamples, and limits

7.14.3 BP-05 — Separate wiring from new capabilities

Endpoint and credential changes normally belong in configuration; provider protocol changes belong behind an adapter. Changed business guarantees require a new capability or explicit version, while ordered steps belong in a graph.

Try one change: Classify changing a payment URL, replacing a provider, and replacing authorization with charging. State which changes affect callers.

Reasoning, counterexamples, and limits

7.14.4 BP-08 — Retain real dependencies and explicit data

Pass business facts through input/output rather than shared mutable context. Remove accidental serialization but retain required control ordering, even without a field reference.

Try one change: Let credit and income checks start independently, then join their facts. Add a charge that must follow approval and explain that control prerequisite.

Reasoning, counterexamples, and limits

7.15 Recap

- An operator should encapsulate **one independently measurable business capability** — the Single Capability Principle.
- There are **four layers** of data transformation: routing (input bindings), light transforms (expressions), structural adapts (transform blocks), and business logic (operator nodes). Use the cheapest layer that fits.
- The **three-layer reuse pyramid** (Infrastructure → Capability → Domain) classifies operators by scope; tag them with `@OperatorMeta(layer = …)`.
- **Typed I/O records** + `SchemaAware` give you compile-time validation, schema-aware tooling, and safer schema evolution.
- **Behavioural contracts** — `idempotency()` and `sideEffectType()` — give the framework and your team clearer guidance for retry, scheduling, and resilience choices.

- **`OperatorContext` is for metadata, not business data.** All domain values flow through the typed input. Use `ctx.timeSource().now()` instead of `Instant.now()` for testable time.
- The most common mistake is creating an operator for pure data reshaping. **If there's no I/O, no side effect, and no domain rule — use a transform.**

7.16 Next Step

In Chapter 8 — Turning a Personal DSL Draft into a Team Asset you will bring these ideas back into the `.bloge` authoring layer — combining graph-level schemas, doc comments, transforms, branches, and operator contracts into files that are clear to read and ready to evolve.

7.17 Reference Links

- Operator Design Specification — full normative spec (boundary rules, schema governance, complexity limits, naming standards)
- `Operator.java` — core operator interface
- `OperatorContext.java` — execution context record
- `SchemaAware.java` — explicit schema declaration interface
- `SchemaDescriptor.java` — sealed schema type hierarchy
- `SchemaIntrospector.java` — automatic schema derivation from Java types
- `OperatorMeta.java` — catalogue metadata annotation
- `OperatorLayer.java` — reuse layer enum
- `SideEffectType.java` — side-effect classification
- `Idempotency.java` — idempotency levels
- Core Architecture — engine scheduling and resilience internals

CHAPTER
08

Arc 2 — Maintainable Authoring

Chapter 8 — Turning a Personal DSL Draft into a Team Asset

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

Promise: *You will follow an order graph through five small commits and see how a draft that merely compiles gains contracts, useful diagnostics, intent, and a reuse boundary—until a teammate can change it safely.*

8.1 Learning goals

By the end of this chapter, you can:

1. Explain why zero compiler errors does not mean a DSL file is maintainable.
2. Turn hidden assumptions into a team contract with graph input, field types, and stable output.
3. Use `invalid-expression-path` to locate the path and schema that disagree.
4. Decide whether an explanation belongs in a `///` doc comment or in code-review discussion.
5. Put shared behavior behind an import alias instead of copying nodes.

8.2 This chapter adds one variable

The first seven chapters gave the order flow enough behavior to load a customer, read products, calculate a price, and decide whether to create an order. This chapter adds no new business behavior. It adds one team reality: **someone else must safely change this graph next week.**

That changes the definition of done. A personal draft only needs its author to remember where `ctx.customerEmail` came from. A team asset must let the compiler, editor, and reviewer trace that assumption.

8.3 Opening problem: why a green draft can worry a reviewer

The first order-summary draft is only a few lines:

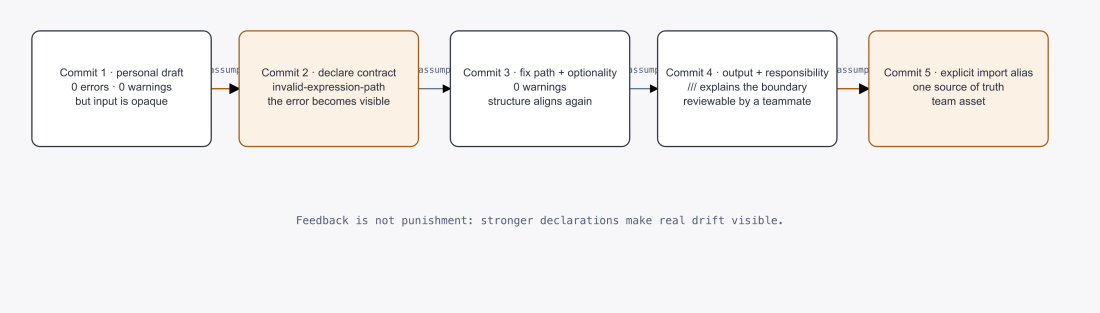
```
graph orderDraft {
  transform orderSummary {
    userId = ctx.userId
    customerEmail = ctx.customerEmail
```

```
  }
}
```

Predict before reading on. If compilation reports `errors=0 warnings=0`, which claims follow?

Claim	Proven by zero diagnostics?
The current compiler accepts the syntax	Yes
Every caller supplies `customerEmail`	No
The field may be absent	No
A downstream consumer knows the output shape	No
A rename six months later will be caught	No

Green means the compiler found no violation of a **declared contract**. When no contract exists, many mistakes are not yet checkable.



8.4 Run the whole trajectory first

The book includes five independent snapshots and an RC1 probe. Install core and DSL from the pinned BLOGE submodule, then run it:

```
cd submodule/bloge
mvn -pl bloge-core,bloge-dsl -am -DskipTests install

cd ../../probes/ch08-dsl
mvn test
```

The important output observed on 2026-09-14 at BLOGE commit `cc38fbe5` is:

```
commit-1 graph=orderDraft         errors=0 warnings=0 firstRule=none
commit-2 graph=orderContractDraft errors=0 warnings=1 firstRule=invalid-expression-path
commit-3 graph=orderTypedDraft    errors=0 warnings=0 firstRule=none
commit-4 graph=orderReviewable    errors=0 warnings=0 firstRule=none
commit-5 graph=orderTeamAsset     errors=0 warnings=0 firstRule=none
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Commit 2 is not worse than commit 1. It turns an invisible assumption into a contract, so the compiler can finally point at the problem.

8.5 Commit 1: expose the business path

The first draft projects only `userId` and an email address. It is useful for a personal experiment: "Can this data flow be expressed?"

It is not ready for a team mainline because `ctx` is opaque. The compiler cannot distinguish "the caller really supplies a top-level `customerEmail`" from "the author misremembered `customer.email`."

The review result is therefore not "failed." It is:

```
syntax accepted
input contract = opaque
safe refactor boundary = unknown
```

8.6 Commit 2: declare the input and invite the error out

The second commit declares the actual order-request shape:

```
input: OrderRequest

schema OrderRequest {
  userId: String
  customer: {
    name: String
    email: String?
  }
}
```

The summary still reads `ctx.customerEmail`. The compiler now knows that the top level contains only `userId` and `customer`, so it reports:

```
warnings=1
firstRule=invalid-expression-path
Path 'ctx.customerEmail' — field 'customerEmail' not found in graph input schema
```

That warning gives three coordinates: the root is graph input (`ctx`), the failing path is `customerEmail`, and the comparison contract is `OrderRequest`.

The recovery is not to disable schema validation. Decide whether reality contains a top-level email or a nested customer email, then align the schema and expression.

8.7 Commit 3: fix the path and state optionality

In this example, reality says `customer.email`, and some customers have no email:

```
transform customer {
  name: String = ctx.customer.name
  email: String? = ctx.customer.email
}

transform orderSummary {
  userId: String = ctx.userId
  customerEmail: String? = customer.output.email
}
```

Compilation returns to zero warnings. Two different relationships are now explicit:

- A path relationship: `orderSummary` depends on `customer.output.email`.
- A type relationship: `String?` says that absence is allowed by contract.

Do not make every field optional merely to silence diagnostics. Optionality is a business fact. If order notification requires an email, keep `String` and fail early when it is missing.

8.8 Commit 4: tell the reviewer why the DSL exists

Correct paths still do not explain intent. The fourth commit adds a graph output contract and a responsibility statement:

```
output: OrderGraphResult
```

```
schema OrderGraphResult {
  orderSummary: OrderSummary
}

/// Projects boundary fields; it does not make a business decision.
transform orderSummary { ... }
```

A `///` comment enters the AST and attaches to the element that follows, so editors and documentation tools can read it. A regular `//` comment helps source readers but is not contract metadata.

Comment	Review value
`/// Creates order summary`	Low: the name already says it
`/// Projects boundary fields; makes no business decision`	High: it defines responsibility
`/// This is important`	Low: nothing can be checked

Good documentation states responsibility and boundary. It does not narrate the syntax.

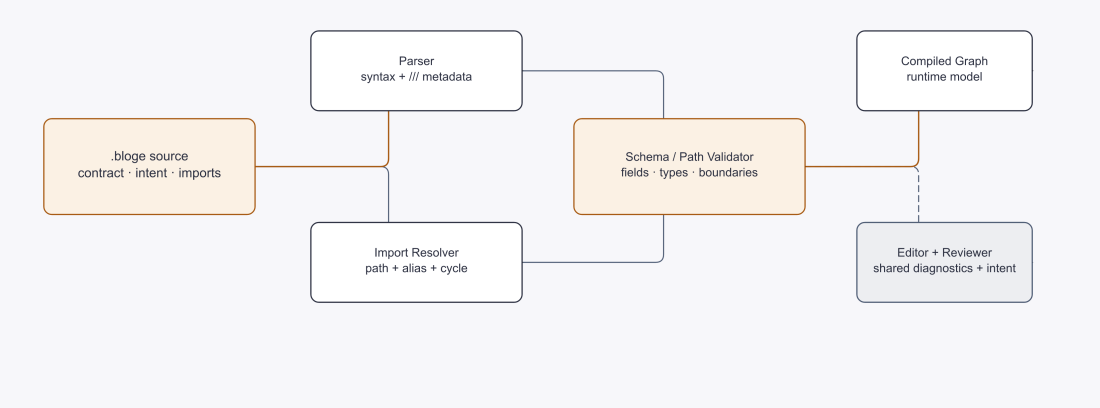
8.9 Commit 5: reuse behavior through an explicit alias

The order graph now needs payment. Copying payment nodes looks quick, but it creates multiple sources of truth for retries, amount checks, and status mapping. The fifth commit extracts a graph and names the dependency:

```
import "./payment-flow" as paymentFlow

node payment : paymentFlow {
  input {
    orderId = ctx.orderId
    amount = ctx.amount
  }
}
```

An alias is not cosmetic shorthand. It is this file's stable name for an imported capability. The compiler must resolve the relative path, compile the imported graph, validate current bindings against its input schema, and expose its output schema downstream.



If the file is missing, imports form a cycle, or two imports use one alias, compilation should stop before a graph is created. Runtime must not guess what an alias means.

8.10 Mechanism: stronger declarations create better feedback

The five commits exercise a feedback pipeline:

```
source
  → lexer/parser: is the syntax legal?
  → import resolver: which graph is referenced?
  → schema resolver: what do fields and types mean?
  → path/boundary validator: are connections compatible?
  → compiled graph: the runtime model
```

Later conclusions depend on earlier layers. Valid syntax cannot replace import resolution; a resolvable import cannot replace schema compatibility; schema compatibility cannot prove business correctness.

That is the core value of a team DSL: assumptions that once lived in an author's head become feedback that machines and people can inspect.

8.11 Failure experiment: a bad alias is not a retry problem

Keep two imported files unchanged but give both `as paymentFlow`. RC1 produces the stable diagnostic `duplicate-import-alias`. Recovery means giving distinct capabilities distinct names, or removing the duplicated source of truth. Retrying runtime execution is meaningless because the graph was never built.

Likewise, `invalid-expression-path` calls for fixing the contract or path, not adding a node retry. Compile-time feedback and runtime resilience operate in different worlds.

8.12 Deliberately moved out of the main story

The complete schema type catalog, derived aliases for bare imports, the import resolver SPI, schema evolution, and `ValidatedSchema` policy belong in the reference layer. They would interrupt the five-commit story here.

In particular, `ValidatedSchema` is a Java runtime wrapper that attaches `STRICT`, `COERCE`, or `LENIENT` policy to a schema; it is not new `.bloge` syntax required in this chapter. The compiler unwraps its expected schema for structural checks. Runtime policy details belong in the DSL engineering reference.

8.13 Real-world transfer: from a personal recipe to a restaurant card

A home cook may understand "salt to taste." Restaurant handoff needs ingredient specification, allergens, output form, reasons for critical steps, and a shared sauce version.

Restaurant handoff	Team DSL
Ingredient list	graph input schema
Finished-dish specification	graph/output schema
Station instructions	nodes/transforms and dependencies
Why a step exists	`///` responsibility comment
Shared sauce recipe	imported graph plus alias
Test-cook feedback	compiler/linter diagnostics

The goal is not a longer recipe. It is safe change by someone who knows when they departed from the standard.

8.14 Your turn: make five reviewable commits

Choose one `.bloge` file you maintain. Make one kind of change per commit:

1. Save the current parseable draft and list its undeclared input assumptions.
2. Declare graph input and let the compiler expose path problems; do not fix them yet.
3. Fix only paths and optionality, then record the diagnostic change.
4. Add a stable output and responsibility boundary; do not paraphrase syntax.
5. Extract one genuinely shared subflow and connect it through an explicit alias.

Produce one review card: which assumption changed, which diagnostic appeared, and who can review more safely as a result. Stop and split the work if one commit changes a business branch, types, and imports together; otherwise you cannot attribute the feedback.

8.15 What this chapter proved—and did not prove

- Zero diagnostics cover declared contracts only; opaque input can create false calm.
- Schema makes bad paths, missing fields, and optionality visible to the feedback loop.
- `///` records responsibility and boundary; it does not replace good naming and structure.
- Import aliases make shared graphs explicit dependencies rather than copied nodes.
- Syntax, references, structural contracts, and business correctness are successive claims. None substitutes for the next.

This probe proves that the five snapshots produce the recorded RC1 compiler feedback. It does not prove that an imported payment service is available, that the business price is correct, or that a production deployment is safe.

The next chapter adds one variable: Chapter 9 — Tooling Workflow turns the maintainable DSL into a repeatable editor, lint, and review loop before Chapter 10 introduces subgraph boundaries.

8.16 Experiment acceptance card

- **Expected and observed:** Five commits expose a bad path, repair the type, and consolidate an alias, ending at 0/0 diagnostics.
- **Failure and recovery:** Introduce a duplicate alias or bad path; repair one declaration and rerun the parser.
- **Proof boundary:** Proves localizable DSL feedback, not a correct business answer.
- **Exercise contract:** Team DSL; make one change per commit; deliver five diagnostics; stop at 0 errors/0 warnings with each transition explained.

8.17 Exact fact entry points

- `DslFeedbackLoopProbeTest.java` — executable five-commit feedback.
- `01-personal-draft.bloge` through `05-team-asset.bloge` — complete snapshots.
- `order-process-v6.bloge` — the pinned remote examples order graph.
- `DslCompiler.java` — compilation and diagnostic source.
- `FileSystemGraphResolver.java` — relative import resolution.
- `ValidatedSchema.java` — runtime validation-policy wrapper.

CHAPTER
09

Arc 2 — Maintainable Authoring

Chapter 9 — Tooling Workflow

Branching, resilience, operator boundaries, DSL, and review feedback

Goal: turn a readable workflow into a change another engineer can review.

Promise: *By the end of this chapter you will understand how the BLOGE tooling stack — from its TypeScript authoring path and Java execution path to the editor, the visual studio, and the CLI linter — works together, and you will know how to set up a productive authoring loop for* `.bloge` *files that catches errors before code review and production.*

9.1 Learning Goals

1. Map the **two implementation paths**: TypeScript `bloge-lang` for LSP/Studio authoring, and Java `bloge-dsl` plus `bloge-lint` for runtime/CI.
2. See how **diagnostics flow** from the lexer, parser, compiler, and lint rules through the LSP to red squiggles in your editor.
3. Use **format-on-save** to keep `.bloge` files consistent via the DSL code generator.
4. Run the **`bloge-lint` CLI** in CI and understand why its Java rule set overlaps with, but is not identical to, editor diagnostics.
5. Trace **`operator-metadata.json`** from `bloge-maven-plugin` into BLOGE Studio's operator palette, then use split view to round-trip between DSL text and the canvas.

9.2 Prerequisites

- Chapter 8 — Turning a Personal DSL Draft into a Team Asset — you need to be comfortable with a complete `.bloge` file.
- Chapter 2 — Your First Graph — you should know how to read compile and execution results.
- Basic familiarity with VS Code or IntelliJ IDEA.

9.3 Source Examples

File	What it shows
`bloge-lang/README.md`	TypeScript authoring core: lexer, parser, compiler, codegen, schema
`bloge-lsp/README.md`	LSP server capabilities and architecture
`bloge-lsp/src/server.ts`	LSP server entry — capability registration
`bloge-lsp/src/document-manager.ts`	Five-stage analysis pipeline (lex → parse → compile → lint → index)
`bloge-lsp/src/features/hover.ts`	Hover docs for keywords, built-in functions, nodes, schemas, and durations
`bloge-lsp/src/features/completion.ts`	Context-aware auto-completion
`bloge-lsp/src/features/formatting.ts`	Whole-document formatting via `DslCodeGenerator`
`bloge-lsp/src/lint/lint-rule.ts`	`LintRule` interface and `LintRunner`
`bloge-vscode/src/extension.ts`	VS Code extension entry — LSP client wiring
`bloge-vscode/syntaxes/bloge.tmLanguage.json`	TextMate grammar for syntax highlighting
`bloge-intellij/README.md`	IntelliJ plugin — dual-mode architecture (PSI + optional LSP overlay)
`bloge-studio/README.md`	Visual DAG editor — React 19 + React Flow
`bloge-studio/public/operator-metadata.json`	Built-in operator catalog (name, description, I/O schema)
`bloge-maven-plugin/README.md`	Maven plugin that generates `operator-metadata.json` from `@BlogeOperator` classes
`bloge-lint/README.md`	Java CLI linter — rules, `.blogerc.json` config, exit codes
`ticket-routing.bloge`	Example used throughout this chapter
`missing-timeout.bloge`	Antipattern caught by lint

9.4 Why This Matters

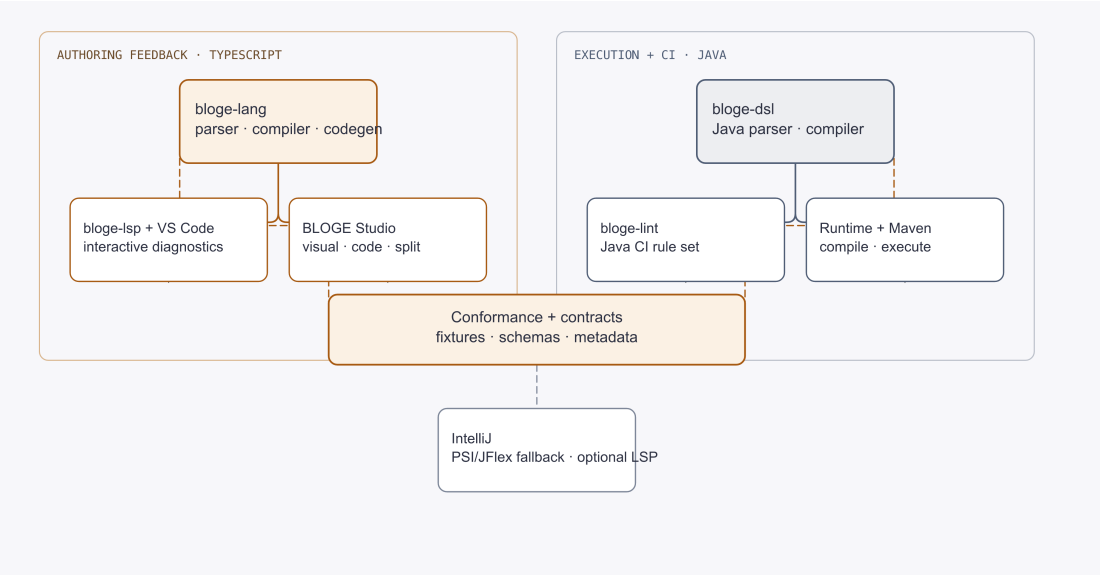
You can write syntactically correct `.bloge` files in a plain text editor. But without tooling:

- **Typos in node IDs** are discovered only at graph-engine start time.
- **Unused nodes** and **missing timeouts** slip past review unnoticed.
- **Formatting drift** makes PRs noisy and obscures real changes.
- **Operator schema mismatches** between the Java code and the DSL are invisible until a runtime `ClassCastException`.

The BLOGE tooling chain is designed so that **the fastest feedback loop is also the cheapest**: red squiggles appear as you type, formatting normalises on save, and the CLI linter runs in CI as a final gate. These surfaces do not share one implementation: TypeScript powers the LSP and Studio, while Java powers runtime parsing and CLI lint. Conformance fixtures and exported contracts keep the paths aligned.

9.5 Mental Model

Think of the tooling stack as two feedback paths joined by explicit compatibility evidence:



The critical insight: **shared contracts are the alignment boundary, not shared code.** `bloge-lang` supplies the TypeScript parser/compiler used by `bloge-lsp` and Studio. Runtime and Maven flows use Java `bloge-dsl`; `bloge-lint` adds Java CI rules; IntelliJ Community keeps a PSI/JFlex path and can add an LSP overlay where supported. Because implementations can drift, compatibility must be checked rather than assumed.

9.6 First Working Example

Open the `ticket-routing.bloge` example in VS Code with the `bloge-vscode` extension installed. You should see immediately:

1. **Syntax highlighting** — keywords like `graph`, `node`, `branch`, `depends_on` are coloured; doc comments (`///`) get documentation-comment scopes; operator references like `FetchCustomerOperator` are highlighted as types.
2. **Diagnostics** — if you deliberately misspell a `depends_on` target (say, change `fetchCustomer` to `fetchCustmer`), a red underline appears before you save.
3. **Hover** — hover over `timeout` and you see: *"Sets the maximum execution time for a node."* Hover over `3s` and you see: *"Duration: 3s = 3000ms"*. Hover over `analyzeSentiment` and you see its operator ref, doc comment, dependencies, and timeout.
4. **Completion** — inside an `input { }` block, type `=` and the LSP suggests `ctx`, all node IDs, and built-in functions like `concat`, `size`, and `coalesce`.
5. **Formatting** — trigger "Format Document" and the entire file is re-emitted through `DslCodeGenerator.generate(ast)`, normalising indentation and spacing.

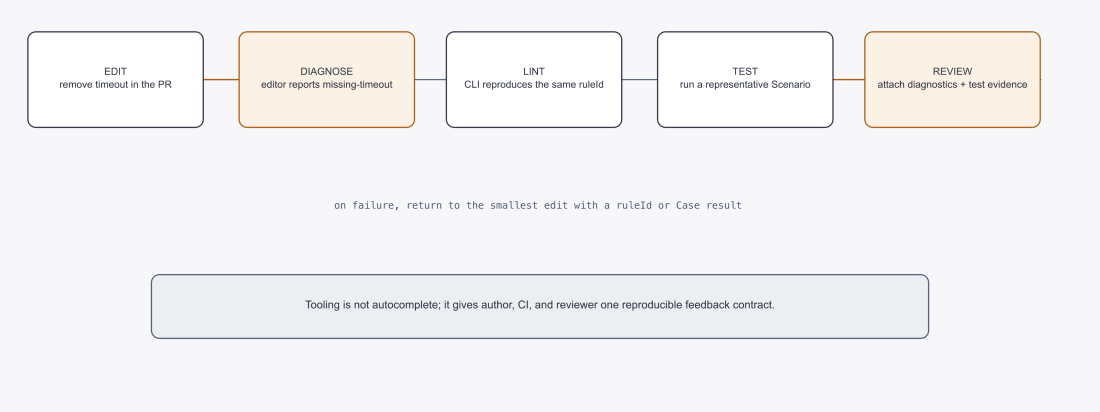
All five features are powered by a single pipeline defined in `document-manager.ts`:

```
// Phase 1: Lexing   → tokens
// Phase 2: Parsing  → AST (GraphDef)
// Phase 3: Compile  → CompiledGraph + diagnostics
// Phase 4: Lint     → LintRunner.run(ast, compiled)
// Phase 5: Index    → buildSymbolIndex(ast, uri, text)
```

This five-stage pipeline runs on **every keystroke** (via `documents.onDidChangeContent` in `server.ts`), and the cached `ParseState` is reused by hover, completion, definition, references, and rename handlers. After those five phases, the `DocumentManager` also refreshes import-graph bookkeeping so cross-file features stay in sync.

9.7 Follow One Pull Request Through the Feedback Loop

A toolchain earns its place when it shortens the distance from an edit to a reviewable fact. Consider a pull request that removes `timeout` from an external call in `missing-timeout.bloge`. The useful story is not a tour of editor features; it is one defect travelling through four feedback surfaces.



9.7.1 One defect, one stable identity

Surface	Reader sees	Required hand-off
Editor	Inline diagnostic at the node	`ruleId=missing-timeout` and source location
CLI lint	Reproducible non-zero result	Same rule id and severity as CI
Scenario test	Runtime behaviour for a representative case	Case verdict plus node/effect evidence
Pull-request review	Why the edit is safe	Diagnostic cleared and focused test result

Autocomplete can help write syntax, but it cannot close this loop. The shared contract is the stable diagnostic identity plus a command that another person can rerun.

9.7.2 Diagnose before testing everything

The shortest useful sequence is:

```
EDIT one declaration
→ DIAGNOSE at the source location
→ LINT with the same rule set as CI
→ TEST the smallest Scenario that exercises the changed path
→ ATTACH both results to review
```

If lint still fails, return to the declaration; a full integration suite adds noise. If lint passes but the Scenario fails, inspect runtime evidence; changing the lint configuration cannot repair business behaviour. Only after both focused signals pass should the broader build run.

This keeps claims bounded: lint proves a declared authoring rule, the Scenario proves behaviour under its fixtures, and neither alone proves production correctness.

9.8 Break It Apart

9.8.1 1 — The shared language core: `bloge-lang`

`bloge-lang` is a pure TypeScript library with **zero dependencies**. It exports:

- **`Lexer`** — hand-written scanner producing tokens from `.bloge` source.

- **`Parser`** — recursive descent with precedence-climbing expression parsing. Handles imports, graph-level `input`/`output` schemas, `signal_schema`, `oneOf(...)`, union types, `?.` safe navigation, and more.
- **`compile(ast, options?)`** — frontend semantic analysis: dependency resolution, cycle detection, schema validation, union narrowing, branch exhaustiveness checking. When a `CompileOptions.importResolver` is provided, imports are resolved recursively.
- **`DslCodeGenerator.generate(ast)`** — AST-to-source formatter. This is the same code path used by both the LSP formatting feature and Studio's export.

Usage in three lines:

```
import { Lexer, Parser, compile, DslCodeGenerator } from 'bloge-lang';

const tokens = new Lexer(source).tokenize();
const ast    = new Parser(tokens).parse();
const result = compile(ast);
// result.diagnostics → errors, warnings
// DslCodeGenerator.generate(ast) → formatted source
```

9.8.2 2 — The LSP: `bloge-lsp`

`bloge-lsp` wraps `bloge-lang` in a Language Server Protocol server that communicates over **stdio**. The `server.ts` entry point registers these capabilities:

Capability	Handler file	What it does
Diagnostics	`document-manager.ts`	Real-time error/warning/info from lexer + parser + compiler + lint
Completion	`features/completion.ts`	Context-aware: keywords, node IDs, schema types, built-in functions, `oneOf(...)` and snippet templates
Hover	`features/hover.ts`	Docs for keywords, ~70 built-in functions, node/schema/transform declarations, durations
Go to Definition	`features/definition.ts`	Symbol-index-backed jump to declaration's `nameRange`; cross-file via `resolveSymbolCrossFile`
Find References	`features/references.ts`	All usages of a node/schema/transform, including transitive importers
Rename	`features/rename.ts`	Renames declaration + all usages; propagates to transitive importers; conflict-checked
Formatting	`features/formatting.ts`	Whole-document via `DslCodeGenerator.generate(ast)`
Code Actions	`features/code-actions.ts`	Quick fixes for `missing-timeout`, `missing-doc-comment`, `unused-node`, `no-duplicate-node-id`; plus a **Source Fix All** action for safe bulk fixes
Semantic Tokens	`features/semantic-tokens.ts`	Full-document semantic highlighting: graph names, operator refs, node refs, path segments, literals, keywords — refines TextMate scopes with AST-aware token classes

Import resolution is automatic for `file://` documents — the `DocumentManager` wires a file-backed `ImportResolver` that resolves paths relative to the open file and appends `.bloge` when needed. Errors in imported files surface as diagnostics in the importing document.

9.8.3 3 — Lint rules

BLOGE lint is **tri-layered**. Knowing which layer a rule lives in tells you where it runs, what context it can see, and which artifact you must rebuild to add a new rule.

Layer	Where it runs	What it sees	Examples
Core (15)	`bloge-lint` CLI + LSP (translated to TS)	A single `.bloge` file's AST after parse + compile	`no-duplicate-node-id`, `no-duplicate-schema-name`, `unused-node`, `missing-timeout`, `excessive-fan-out`, `missing-doc-comment`, `no-cycle`, `no-unresolved-dependency`, `no-unresolved-branch-target`, `max-script-nodes`, `script-timeout-required`, `script-line-limit`, `loop-exit-completeness`, `cron-expression-valid`, `deadline-in-past`
Compiler-driven (3)	Inside the compiler itself	Whole-graph IR after symbol resolution and type checking	`schema-validation-strictness-mismatch`, `dynamic-subgraph-output-untyped`, `ambiguous-timer`
Extension (2 today, more via SPI)	Discovered when the matching `*-ext` JAR is on the classpath	The extension's own DSL surface	`session-backward-transition-guard` (from `bloge-session-ext`), `state-machine-structure` (from `bloge-state-ext`)

9.8.3.1 Core rules — LSP subset

The LSP bundles the six interactive rules (`bloge-lsp/src/lint/rules/`):

Rule ID	Severity	What it catches
`no-duplicate-node-id`	Error	Node ID declared more than once
`no-duplicate-schema-name`	Error	Schema name declared more than once
`unused-node`	Info	Node never referenced downstream
`missing-timeout`	Info	Node without a timeout
`excessive-fan-out`	Info	Node with >5 outgoing edges
`missing-doc-comment`	Info	Node missing `///` documentation

The Java-side CLI linter (`bloge-lint`) includes these same core rules for batch checks and adds the graph-level and script-focused rules listed above, plus the rules contributed by extensions via the `LintExtensionContributor` SPI.

9.8.3.2 Compiler-driven rules

These three rules are emitted by the **compiler**, not the rule runner. You cannot disable them in `.blogerc.json` — they reflect structural checks the compiler must run to produce a valid `Graph`. They surface as ordinary diagnostics in the LSP and CLI just like core rules.

9.8.3.3 Extension rules

Adding `bloge-session-ext` or `bloge-state-ext` to your project also adds that extension's lint contributions. You can register your own rules via `LintRuleProvider` (Java SPI) for the CLI and `BLOGE_LSP_EXTRA_RULES` for the LSP. Severity levels and rule parameters are configurable through `.blogerc.json`:

```
{
  "rules": {
    "missing-timeout": "warning",
    "missing-doc-comment": "off"
  }
}
```

9.8.3.4 Decision-table rules The bloge LSP and CLI both ship two lint rules specifically for `decision_table` nodes. Use these recipes to keep tables exhaustive and intentional.

Rule ID	Severity	Triggers when
`decision-table/missing-otherwise`	WARNING	A `first`, `unique`, or `any` table has no `otherwise` clause
`decision-table/collect-otherwise`	INFO	A `collect` table has an `otherwise` clause (always appends — may be unintentional)

Recipe 1 — Silence `decision-table/missing-otherwise` by adding `otherwise`:

```
// [WARN] decision-table/missing-otherwise
decision_table credit_tier(score = ctx.score) hit=first -> String {
  rule (score: score >= 750) -> "platinum"
  rule (score: 680 <= score < 750) -> "gold"
  // no otherwise — what happens if score < 680?
}

// [OK] Fixed
decision_table credit_tier(score = ctx.score) hit=first -> String {
  rule (score: score >= 750) -> "platinum"
  rule (score: 680 <= score < 750) -> "gold"
  otherwise -> "rejected"
}
```

Recipe 2 — Acknowledge `decision-table/collect-otherwise` if it is intentional:

```
// [INFO] decision-table/collect-otherwise
// "baseline" is ALWAYS appended, even when other rules also matched.
decision_table discounts(score = ctx.score) hit=collect -> String {
  rule (score: score >= 700) -> "loyalty"
  otherwise -> "baseline"   // unconditional — is this intentional?
}

// [OK] If intentional, suppress with a .blogerc.json override:
// { "rules": { "decision-table/collect-otherwise": "off" } }
// If not intentional, remove the otherwise clause.
```

For a full list of compiler and runtime error codes, see Appendix F — Decision Table.

9.8.4 4 — Editor extensions

VS Code (`bloge-vscode`):

- TextMate grammar at `syntaxes/bloge.tmLanguage.json` provides syntax highlighting scopes for keywords, doc comments, strings, operators, types, and built-in functions.
- Language configuration at `language-configuration.json` provides bracket matching, auto-closing pairs, comment toggling (`//`, `/* */`), and block folding.
- The extension entry (`extension.ts`) launches `bloge-lsp` as a child process over stdio and wires it as the LSP client for the `bloge` language ID.

IntelliJ IDEA (`bloge-intellij`):

- **Community edition** — PSI-only mode: hand-written JFlex lexer, lightweight `PsiBuilder` parser, `ExternalAnnotator` delegates to `bloge-dsl` Parser and `bloge-lint` `LintRunner`, structure view, code folding, semantic highlighting. No external process required.
- **Ultimate / commercial editions** — optional LSP overlay: `BlogeLspServerSupportProvider` launches the bundled `bloge-lsp` Node server; overlapping PSI providers (diagnostics, completion, formatting) stand down to avoid

duplicate results. Requires Node.js on `PATH` or `BLOGE_NODE_PATH`.

9.8.5 5 — BLOGE Studio

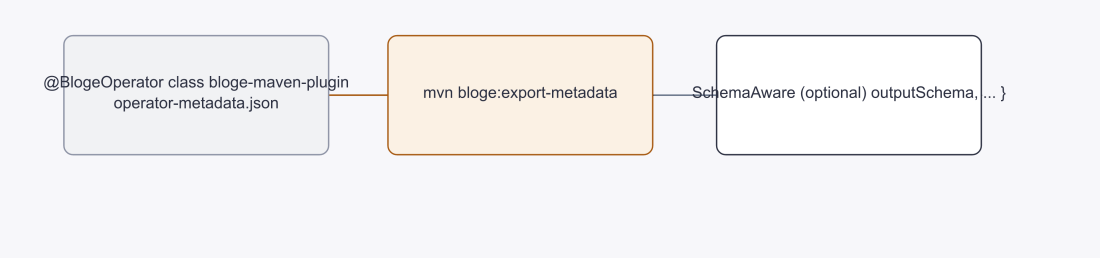
BLOGE Studio is a browser-based visual DAG editor built with React 19 + React Flow 12. It depends on `bloge-lang` as a local `file:` dependency and reuses the same parser, compiler, and code generator.

Key features relevant to your authoring workflow:

- **Three view modes**: Visual (canvas only), Code (Monaco editor only), Split (canvas + editor side by side). The Zustand store synchronises between them.
- **Operator Palette**: loaded from `public/operator-metadata.json`. Operators are tagged by layer (infra / capability / domain) and searchable. Drag to canvas to create a node.
- **Diagnostics Panel**: `DiagnosticsPanel.tsx` — displays errors, warnings, and info from the compiler and lint rules. Click a diagnostic to select the offending node on the canvas.
- **Undo/Redo**: DSL-snapshot-based history (up to 50 steps).
- **Export**: validates → runs code generator → downloads `.bloge` file.

9.8.6 6 — The operator metadata pipeline

The link between your Java operator code and the visual tooling is `bloge-maven-plugin`:



The plugin scans `@BlogeOperator`-annotated classes at `process-classes` phase, reflects `Operator<I,O>` generic parameters, calls `SchemaIntrospector.introspect()` for each type (with Record and POJO support), and writes a JSON file. Studio loads this file to populate the operator palette and provide I/O schema information for field completion.

9.9 The Embedded Ops Console — `/bloge-console`

Studio is great for authoring graphs. Production needs a different view: "which executions are running right now, which timer fires next, which operator is throwing." That view is **`/bloge-console`** — a small, read-only operator UI that ships with `bloge-spring-web` and mounts at the configured base path.

```yaml
spring:
  bloge:
    console:
      enabled: true
      base-path: /bloge-console        # default
      require-role: BLOGE_OPS          # Spring Security role gate
```

What it shows:

- **Executions** — running, waiting, completed, failed; click into the audit journal for a single run.
- **Timers** — currently armed durable timers with their next fire time; manual *fire now* (gated by `require-role`).
- **Leases** — current owners with TTL remaining.
- **Schema** — Flyway `currentVersion`, pending/applied migrations, and `UP_TO_DATE`, `PENDING`, or `FAILED` status.
- **Operator registry** — every registered operator, its module, and its `@BlogeOperator` metadata for quick discovery.

The console is **observational** by design. It does not let you edit graphs (use Studio) or rewrite checkpoints (use a migration). The "fire timer" and "release lease" actions are explicit operator-recovery tools and every action is recorded

in the audit journal under the calling principal.

> ***Brain check:*** *when should you reach for* `/bloge-console` *vs Studio? Hint: Studio writes the graph; the console reads the runtime.*

9.10 Common Trap

9.10.1 [X] Assuming the editor and CI check the same rules

The LSP lint rules (TypeScript, in `bloge-lsp`) and the CLI lint rules (Java, in `bloge-lint`) are **two separate implementations** with overlapping but not identical rule sets. The LSP has `unused-node` and `excessive-fan-out`; the CLI has `no-unresolved-dependency`, `no-unresolved-branch-target`, and `no-cycle`. If you rely only on green squiggles in your editor, CI may still fail on rules the LSP does not check.

```
# Run the Java linter locally before pushing:
java -jar bloge-lint.jar check src/main/resources/bloge/

# Or configure it in your build:
# .blogerc.json controls severity overrides for both editor and CLI usage.
```

Best practice: run `bloge-lint check` as a pre-commit hook or CI step so that the full rule set is applied before merge.

9.11 Guided Rewrite

Take the `missing-timeout.bloge` antipattern file and use the tooling to fix it:

1. **Open it in VS Code** with the `bloge-vscode` extension. You should see an info diagnostic on `loadReport` from the `missing-timeout` lint rule.
2. **Hover** over `loadReport` to confirm there is no timeout listed in the hover popup.
3. **Add a timeout**:

```
node loadReport : LoadReportOperator {
  input {
    reportId = ctx.reportId
  }
  timeout = 5s
}
```

4. **Format the document** (Shift+Alt+F in VS Code). The `DslCodeGenerator` normalises the indentation.
5. **Add a `///` doc comment** above `loadReport`. Confirm the `missing-doc-comment` diagnostic disappears.
6. **Run the CLI linter** against the file:
   `java -jar bloge-lint.jar check src/main/resources/bloge/antipatterns/missing-timeout.bloge`
   With the timeout and doc comment added, the output should report zero errors.
7. **Open the same file in BLOGE Studio** (`npm run dev` in `bloge-studio/`, then use "Open" in the toolbar). Switch to Split view and observe the canvas and DSL code in sync. Check the Diagnostics panel at the bottom — it should be empty.

9.12 Brain Check

1. Which module does the LSP's formatting feature actually delegate to? *(The `DslCodeGenerator` from `bloge-lang`. See `formatting.ts` — it calls `DslCodeGenerator.generate(state.ast)`.)*

2. How does the LSP detect that a `depends_on` target is misspelled? *(`compile(ast)` in `bloge-lang` runs dependency resolution. An unresolvable dependency produces an `error`-level diagnostic, which the `DocumentManager` converts to an LSP `DiagnosticSeverity.Error`.)*
3. What happens in the IntelliJ plugin when the IDE is Community edition and Node.js is not installed? *(The plugin operates in PSI-only mode — all features are provided by the built-in JFlex lexer, PsiBuilder parser, and ExternalAnnotator. The LSP overlay never activates because `com.intellij.modules.lsp` is absent.)*
4. How does BLOGE Studio know which operators to show in its palette? *(It loads `public/operator-metadata.json` on startup. This file can be generated by `bloge-maven-plugin`'s `export-metadata` goal, or hand-authored.)*
5. Can you disable a lint rule in the CLI? How? *(Yes — create a `.blogerc.json` file with a `rules` map. Set the rule ID to `"off"`. For example: `{ "rules": { "missing-doc-comment": "off" } }`. See `LintConfig.java`.)*

9.13 Lab

1. **Editor setup**: Install the `bloge-vscode` extension (or run the IntelliJ plugin sandbox with `./gradlew runIde` in `bloge-intellij/`). Open `ticket-routing.bloge`. Verify that:
   - Syntax highlighting covers all keywords and doc comments.
   - Hovering over `analyzeSentiment` shows its operator, description, dependencies, and timeout.
   - Ctrl+Click on `fetchCustomer` in the `depends_on` list jumps to the node declaration.
   - Renaming `classifyPriority` updates the branch condition and all `depends_on` references.
2. **Break a graph on purpose**: In `ticket-routing.bloge`, change `depends_on = [analyzeSentiment]` on `classifyPriority` to `depends_on = [nonExistentNode]`. Observe the diagnostics that appear. Then run `bloge-lint check` on the file and compare the CLI output to the editor squiggles.
3. **Operator metadata round-trip**: If you have a Maven project with `@BlogeOperator` classes, run `mvn bloge:export-metadata` and copy the resulting `operator-metadata.json` into `bloge-studio/public/`. Start Studio and confirm your operators appear in the palette with correct I/O schemas.
4. **Studio split-view**: Open `ticket-routing.bloge` in BLOGE Studio. Switch to Split view. Add a new node in the code editor and watch it appear on the canvas. Delete it and use Undo (Ctrl+Z) to bring it back.

9.14 Bridge: from tooling to a verification gate

Editor diagnostics and `bloge-lint` shorten authoring feedback; they do not execute an approved business Scenario. In a normal customer module, enter the `0.9.8-RC1` verification path by compiling the bootstrap and running the Preview goal against `.blogeverify.yaml` in one Maven invocation:

```
mvn test-compile \
  com.leanowtech.bloge:bloge-maven-plugin:0.9.8-RC1:verify
```

The generated artifact still needs source binding and a receipt retained outside its mutable directory. In RC1, this book's loan starter fails closed because its `com.leanowtech.bloge.starter.*` bootstrap hits the plugin classloader boundary; run its JUnit Scenario tests instead, and do not interpret `TEST_CLASSPATH_UNAVAILABLE` as a business failure. Chapter 32 — Change Attribution, Maven, and Release Decisions explains source installation, the known limitation, the command ladder, and the claim boundary.

9.15 Experiment acceptance card

- **Expected and observed:** A missing-timeout defect crosses edit, diagnose, lint, Scenario, and review as one loop.
- **Failure and recovery:** Remove timeout to fail the gate; restore it and rerun the smallest check set.
- **Proof boundary:** Proves the toolchain blocks a declared defect, not an unencoded rule.
- **Exercise contract:** One PR; change one defect; deliver diagnostic, test, and review receipts; stop when all point to one rule.

9.16 Coding practice

9.16.1 BP-04 — Prove a public function gap

A reusable project helper does not automatically belong in core. A public proposal needs independent consumers sharing semantics, a gap in existing compositions, complete edge contracts, bounded cost, ownership, and compatibility.

Try one change: Compare an application policy calculation with a proposed key-based distinct operation. Write the null-key and ordering contract before proposing any syntax.

Reasoning, counterexamples, and limits

9.17 Recap

- **`bloge-lang`** is the TypeScript authoring core used by the LSP and Studio; it is not the Java runtime parser.
- **`bloge-dsl`** is the Java runtime/Maven parsing path; conformance fixtures and exported contracts are the compatibility boundary between the two implementations.
- **`bloge-lsp`** wraps `bloge-lang` in an LSP server that provides diagnostics, completion, hover, go-to-definition, find references, rename, and formatting over stdio.
- **`bloge-vscode`** provides TextMate syntax highlighting and language configuration, then delegates all intelligent features to `bloge-lsp`.
- **`bloge-intellij`** works in PSI-only mode on Community edition and adds an optional LSP overlay on Ultimate/commercial editions.
- **BLOGE Studio** is a browser-based visual editor with a three-way view (visual / code / split), a diagnostics panel, and an operator palette driven by `operator-metadata.json`.
- **`bloge-maven-plugin`** bridges Java operator code to the visual tooling by generating `operator-metadata.json` from `@BlogeOperator` annotations.
- **`bloge-lint`** (Java CLI) runs the same style of lint rules in CI; its rule severity is configurable via `.blogerc.json`.
- The LSP and CLI lint rule sets **overlap but are not identical** — always run the CLI linter in CI as a final gate.

9.18 Next Step

In Chapter 10 — Reuse with Subgraphs, you will use this feedback loop while extracting reusable graph behavior behind an explicit parent/child contract.

9.19 Reference Links

- `bloge-lang/README.md` — TypeScript authoring-core documentation
- `bloge-lsp/README.md` — LSP server documentation and architecture diagram
- `bloge-lsp/src/document-manager.ts` — five-stage analysis pipeline source
- `bloge-lsp/src/lint/lint-rule.ts` — `LintRule` interface and built-in rule set
- `bloge-vscode/README.md` — VS Code extension documentation
- `bloge-vscode/src/extension.ts` — LSP client wiring
- `bloge-vscode/syntaxes/bloge.tmLanguage.json` — TextMate grammar
- `bloge-intellij/README.md` — IntelliJ plugin documentation and dual-mode architecture

- `bloge-studio/README.md` — Visual DAG editor documentation
- `bloge-studio/src/components/panels/DiagnosticsPanel.tsx` — Studio diagnostics panel source
- `bloge-studio/public/operator-metadata.json` — built-in operator catalog
- `bloge-maven-plugin/README.md` — Maven plugin for operator metadata generation
- `bloge-lint/README.md` — Java CLI linter documentation
- `bloge-lint/src/main/java/.../LintConfig.java` — `.blogerc.json` configuration parser
- `bloge-lint/src/main/java/.../BlogeLintCli.java` — CLI entry point
- `ticket-routing.bloge` — primary example
- `missing-timeout.bloge` — antipattern fixture

Arc 2 Recap — Maintainable Authoring (Chapters 5–9)

CHAPTER

—

Arc 2 Recap — Maintainable Authoring (Chapters 5–9)

You can now author branching, resilient workflows with well-designed operators and clean, reviewable DSL files.

0.1 What You Learned

Chapter 5 — Branches That Decide introduced `branch on` for control-plane routing. You learned how the engine marks un-chosen paths as SKIPPED, why `otherwise` matters, and how to distinguish `branch on` (routing) from `when` (value selection).

Chapter 6 — Resilience by Design showed that retry, timeout, fallback, and compensate are graph-level declarations, not inline try/catch. You learned to attach policies to nodes, layer them (retry inside timeout inside fallback), and predict the engine's behavior when policies interact.

Chapter 7 — Designing Good Operators turned attention to the operator boundary. You learned what makes an operator testable (single responsibility, no hidden state, declared schema), how to handle failure idiomatically, and when to return a rich result type vs. throw.

Chapter 8 — Turning a Personal DSL Draft into a Team Asset pulled everything together into maintainable `.bloge` files. You practiced graph-level schemas, doc comments, named schemas, `let` bindings, and consistent file layout — the skills that turn a personal draft into a reviewable team asset.

Chapter 9 — Tooling Workflow closed the authoring loop. You followed one change from editor diagnostics to lint, tests, rendered graph, and pull-request evidence, so a readable DSL file becomes a change another person can review.

0.2 What You Should Be Able to Do Now

- Route execution through different paths with `branch on` and provide a safe default with `otherwise`.
- Attach `retry`, `timeout`, and `fallback` to nodes that need them — and explain why the other nodes do not.
- Design an operator with a clear input/output contract and testable boundaries.
- Write a complete `.bloge` file with schemas, doc comments, branches, resilience policies, and transforms.
- Review someone else's `.bloge` file and spot structural issues.
- Run the edit → lint → test → visualize loop and attach the smallest useful evidence to a review.

0.3 Common Mistakes at This Stage

Mistake	Why It Happens	Fix
Branching on the wrong expression type	Using a string where the engine expects an enum or boolean	Match the `branch on` expression type to the declared cases exactly.
Adding retry to every node	Defensive instinct	Retry only the nodes that call unreliable external systems; pure-logic nodes should not retry.
Forgetting `otherwise`	Trusting that all cases are covered	Always include `otherwise` — even if it routes to an error node — so the graph never silently stalls.
Operators that know about graph structure	Passing graph context internals into operator logic	Operators receive typed input and context; they should not navigate the graph model.
Undocumented schemas	"The types are obvious"	Add `///` doc comments and explicit schema blocks; they are free and save the next reader minutes.
Treating editor diagnostics as proof	Red underlines feel authoritative	Let the editor shorten feedback; let parser and behavior tests decide correctness.

0.4 Key Vocabulary Additions

Term	Meaning
`branch on`	Control-plane routing: directs execution to one of several target nodes based on a runtime value.
`otherwise`	The default branch target when no explicit case matches.
SKIPPED	The status given to nodes on branch paths that were not taken.
`retry`	Resilience policy: re-execute a node up to N times on failure.
`timeout`	Resilience policy: cancel a node if it exceeds a duration.
`fallback`	Resilience policy: substitute a default result when the primary execution fails.
`compensate`	Resilience policy: run a cleanup action when a previously successful node must be undone.
Schema	A named or inline type declaration for a graph or node's input/output contract.
Authoring loop	Edit, lint, test, visualize, and review the same change through increasingly strong feedback.

0.5 Try It: Prepare a Reviewable Change

A teammate adds retry to every node in a payment graph and submits only a DSL screenshot. Mark each node **keep retry** or **remove retry**, then write the three artifacts you require before review. A strong answer distinguishes unreliable I/O from deterministic logic and asks for lint output, a focused behavior test, and a rendered graph showing the changed path.

0.6 Where to Go Next

Arc 3 teaches composition: sub-graphs, batch iteration, waiting for external events, and durable execution. Start with Chapter 10 — Reuse with Subgraphs.

Arc 3 — Composition and Time

Subgraphs, iteration, suspension, and crash recovery

Goal: compose work across collections, external time, and process restarts.

CHAPTER
10

Arc 3 — Composition and Time

Chapter 10 — Reuse with Subgraphs

Subgraphs, iteration, suspension, and crash recovery

Goal: compose work across collections, external time, and process restarts.

Promise: *By the end of this chapter you will know how to extract a pipeline into a named subgraph, embed it inside a parent graph with `subgraph("name")`, control scope isolation, and recognise when a dynamic subgraph is the right tool instead.*

10.1 Learning Goals

1. Explain what a subgraph is in BLOGE and why it exists.
2. Register a pre-built `Graph` with `DslCompiler.registerSubGraph()` and reference it in DSL using the `node x : subgraph("name") {}` syntax.
3. Distinguish **isolated** scope (the default for subgraphs) from **parent** scope, and predict what each mode makes visible inside the child graph.
4. Read subgraph output — understand that terminal-node outputs are aggregated into the parent node's result map.
5. Recognise when to use the `dynamicSubGraph` operator for runtime-generated DSL instead of compile-time subgraph registration.

10.2 Prerequisites

- Chapter 8 — Turning a Personal DSL Draft into a Team Asset

10.3 Source Examples

File	What it shows
`loan-approval-subgraph.bloge`	Two parallel subgraphs (credit + compliance) feeding an underwriting decision with a branch
`international-shipment.bloge`	Parallel customs-clearance and route-optimisation subgraphs with fan-in
`smart-ticket-handling.bloge`	Sentiment-analysis subgraph feeding a priority branch into an escalation subgraph

File	What it shows
`order-full-pipeline.bloge`	Payment and inventory subgraphs running in parallel
`dynamic-agent.bloge`	Runtime DSL generation + `dynamicSubGraph` execution
`LoanApprovalSubGraphDslExample.java`	Java wiring: builds sub-graphs via `Graph.builder()`, registers them with `DslCompiler.registerSubGraph()`
`SubGraphOperator.java`	Engine-level implementation that executes a child graph and aggregates terminal outputs

10.4 Why This Matters

By Chapter 8 you can write a complete graph in DSL. But real systems have pipelines that recur across domains:

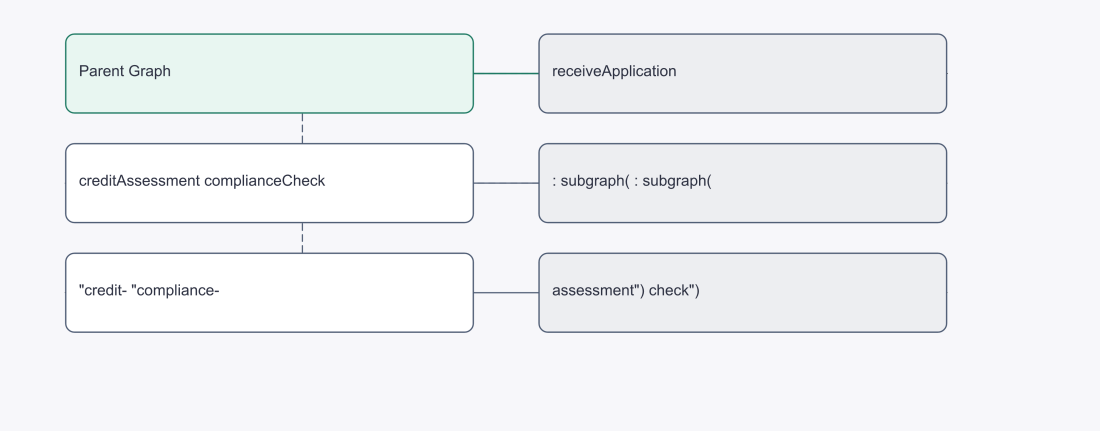
- A **credit-assessment** pipeline (credit query → income verification → debt ratio → risk scoring) appears in loan approval, mortgage processing, and credit-card issuance.
- A **compliance-check** pipeline (AML screening → KYC → sanctions → compliance determination) is required by multiple regulatory workflows.
- A **kitchen dispatch** or **delivery coordination** pipeline is shared across restaurant order types.

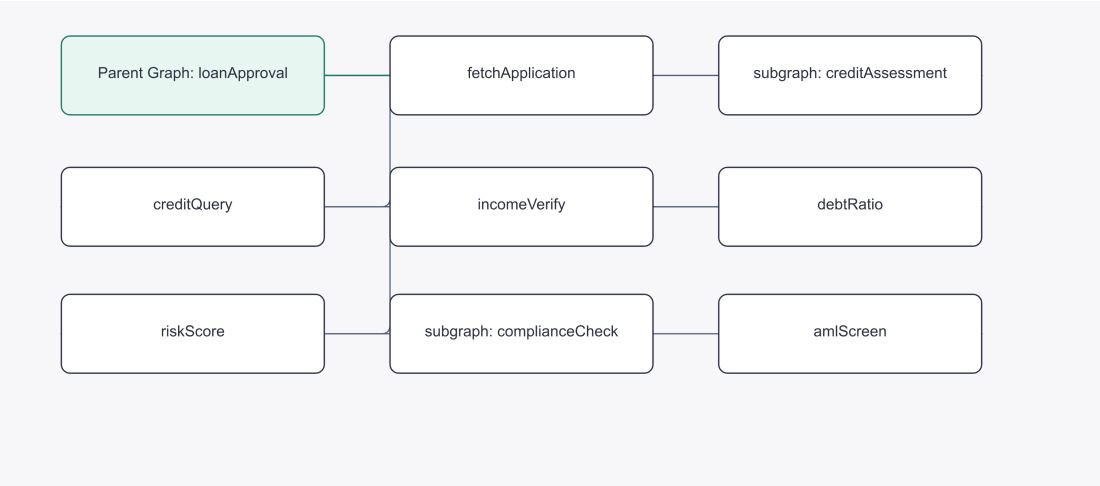
Without subgraphs, you would copy-paste entire node chains into every parent graph, duplicating resilience config, dependencies, and operator wiring. When the compliance rules change, you update *N* copies — and inevitably miss one.

BLOGE solves this with **named subgraphs**: you build a pipeline once as a `Graph`, register it under a name, and reference it from any parent graph with `node x : subgraph("name") {}`. The engine runs the child graph as a nested execution, injects the parent's `input {}` bindings as the child's context, and aggregates the child's terminal-node outputs back into the parent node's result.

One pipeline. One registration. Many call sites.

10.5 Mental Model





Think of each subgraph box as a **function call**: the parent passes arguments via `input {}`, the child runs its internal DAG, and the parent receives the terminal nodes' outputs as a map keyed by node ID.

Key rules:

1. **Input flows in through `input {}`.** The bindings are injected as the child's `GraphContext`. The child graph reads them with `ctx.*` expressions.
2. **Output flows out through terminal nodes.** The `SubGraphOperator` iterates `subGraph.terminalNodes()` and collects each terminal's output into a `Map<String, Object>`. The parent references them as `subgraphNode.output.terminalNodeId`.
3. **Scope defaults to `isolated`.** The child does *not* see parent node outputs or inherit the parent `GraphContext` unless you explicitly set `scope = parent`.

10.6 First Working Example

Here is `loan-approval-subgraph.bloge`, trimmed to the subgraph wiring:

The 71-line wiring is kept in one block because both parallel child contracts, their parent fan-in, and the two terminal routes must be visible together; a shorter fragment would conceal the parent-visible output boundary.

```
graph loanApprovalPipeline {

  node receiveApplication : ReceiveApplicationOperator {
    input {
      applicationId   = ctx.applicationId
      applicantName   = ctx.applicantName
      requestedAmount = ctx.requestedAmount
      termMonths      = ctx.termMonths
      employerId      = ctx.employerId
    }
    timeout = 3s
  }

  /// Runs credit assessment sub-graph:
  /// creditQuery → incomeVerification → debtRatioCalc → riskScoring
  node creditAssessment : subgraph("credit-assessment") {
    depends_on = [receiveApplication]
    input {
      applicationId   = receiveApplication.output.applicationId
      applicantName   = receiveApplication.output.applicantName
```

```
      requestedAmount = receiveApplication.output.requestedAmount
      employerId      = receiveApplication.output.employerId
    }
    timeout = 30s
  }

  /// Runs compliance check sub-graph:
  /// amlScreening → kycVerification → sanctionListCheck → complianceDetermination
  node complianceCheck : subgraph("compliance-check") {
    depends_on = [receiveApplication]
    input {
      applicationId = receiveApplication.output.applicationId
      applicantName = receiveApplication.output.applicantName
    }
    timeout = 30s
  }

  node underwritingDecision : UnderwritingDecisionOperator {
    depends_on = [creditAssessment, complianceCheck]
    input {
      applicationId = receiveApplication.output.applicationId
      credit        = creditAssessment.output.riskScoring
      compliance    = complianceCheck.output.complianceDetermination
    }
  }

  branch on underwritingDecision.output.decision {
    "approved" -> generateApprovalLetter
    otherwise  -> generateRejectionNotice
  }

  node generateApprovalLetter : GenerateApprovalLetterOperator {
    depends_on = [underwritingDecision]
    input {
      applicationId   = underwritingDecision.output.applicationId
      applicantName   = ctx.applicantName
      requestedAmount = ctx.requestedAmount
      approvedRate    = underwritingDecision.output.approvedRate
      termMonths      = ctx.termMonths
    }
  }

  node generateRejectionNotice : GenerateRejectionNoticeOperator {
    depends_on = [underwritingDecision]
    input {
      applicationId = underwritingDecision.output.applicationId
      applicantName = ctx.applicantName
      reason        = underwritingDecision.output.reason
    }
  }
}
```

Read it as a story:

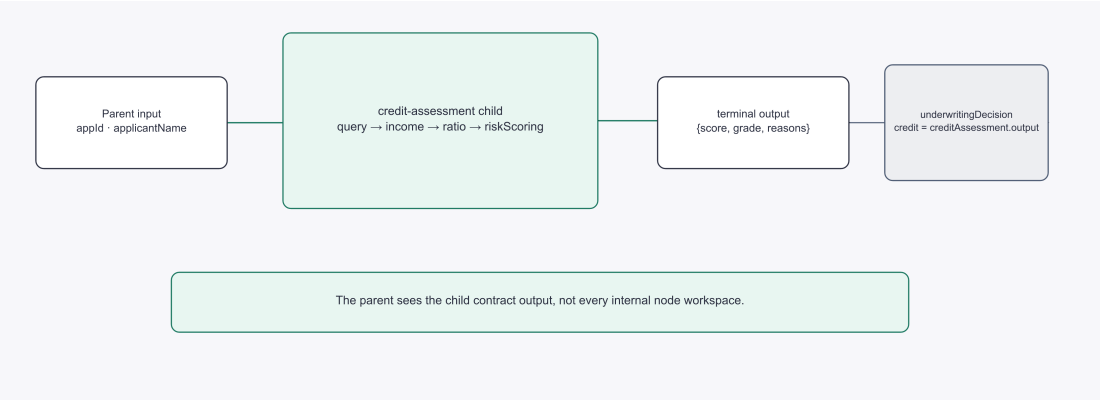
1. **`receiveApplication`** validates the incoming loan request.
2. **Two subgraphs run in parallel** — `creditAssessment` and `complianceCheck` — each wrapping a four-node pipeline. Both depend on `receiveApplication`, so the engine schedules them concurrently.
3. **`underwritingDecision`** fans in from both subgraphs. It reads `creditAssessment.output.riskScoring` — the

terminal node's output from the credit subgraph — and `complianceCheck.output.complianceDetermination`.
4. **The branch** routes to approval or rejection.

Key insight: *Each `subgraph("name")` node looks like a regular node in the parent's DAG. The engine doesn't care that it contains four internal steps — it sees one node with dependencies and an output map.*

10.7 One Child Graph, One Parent-Visible Result

In the loan pipeline, `creditAssessment` contains several internal steps: credit query, income verification, debt-ratio calculation, and risk scoring. The parent graph should not bind to all four workspaces. It needs one stable credit-assessment result.



10.7.1 Read the boundary from the outside in

Viewpoint	May depend on	Should not depend on
Child internals	Intermediate outputs such as income and debt ratio	Parent node names or parent scheduling
Child terminal contract	`{score, grade, reasons}`	Internal node layout
Parent `underwritingDecision`	`creditAssessment.output`	`creditQuery.output` inside the child

The companion smoke scenario observes the public result: both `creditAssessment` and `complianceCheck` execute, underwriting consumes their outputs, and the approval letter executes while the rejection notice is `SKIPPED`. The parent never needs to know which child node produced the score.

10.7.2 Refactor without leaking the workshop

Rename `riskScoring` to `calculateRisk`, or insert a fraud check before it. If the terminal output still satisfies `{score, grade, reasons}`, the parent graph does not change. If the parent references an internal child node, reuse has failed: a private refactor becomes a cross-graph migration.

Keep dynamic subgraphs as an advanced deployment technique. They change how a definition is selected at runtime; they do not change this contract rule. Learn the static parent/child output boundary first, then add dynamic selection only when the business truly chooses among definitions.

10.8 Break It Apart

10.8.1 The `subgraph("name")` syntax

```
node <id> : subgraph("<registered-name>") {
  depends_on = [...]
  input { ... }
  timeout = ...
  scope = isolated | parent    // optional; defaults to isolated
}
```

The parser recognises the `subgraph(` token sequence and rewrites the operator reference to an internal prefix (`subgraph:<name>`). The `StandardNodeCompiler` then looks up the name in the registered subgraph map and wires a `SubGraphOperator`.

10.8.2 Registering subgraphs (Java side)

Before compiling the parent DSL, you build each child graph and register it:

```
// Build sub-graphs via Java API
Graph creditGraph    = buildCreditAssessmentSubGraph();   // Graph.builder("credit-assessment")...
Graph complianceGraph = buildComplianceCheckSubGraph();  // Graph.builder("compliance-check")...

// Register before compile()
var compiler = new DslCompiler(registry);
compiler.registerSubGraph("credit-assessment", creditGraph);
compiler.registerSubGraph("compliance-check", complianceGraph);

Graph mainGraph = compiler.compile(ast);
```

(From `LoanApprovalSubGraphDslExample.java`, lines 274–284.)

Each child graph is built with the standard `Graph.builder()` API — operators, dependencies, inputs, retry, timeout — exactly like a top-level graph. The only difference is that it reads its initial data from `ctx.*` rather than a parent node's output.

10.8.3 How output aggregation works

`SubGraphOperator.execute()` runs the child graph and then:

```
Map<String, Object> output = new LinkedHashMap<>();
for (String terminalId : subGraph.terminalNodes()) {
    Object terminalOutput = result.results().getRaw(terminalId);
    if (terminalOutput != null) {
        output.put(terminalId, terminalOutput);
    }
}
return output;
```

The parent references each terminal by name: `creditAssessment.output.riskScoring` navigates to the `riskScoring` key in that map.

10.8.4 Scope: `isolated` vs `parent`

Scope	Default for	Behaviour
`isolated`	`subgraph("name")`	Child receives only explicit `input {}` bindings. No parent `GraphContext` leakage.
`parent`	`loop`, `foreach`	Child inherits the parent `GraphContext` and can reference parent node outputs via `__parentOutput_*` keys injected at runtime.

You can override the default:

```
node nested : subgraph("childGraph") {
```

```
  scope = parent
  input { key = parentNode.output.value }
}
```

With `scope = parent`, the `SubGraphOperator` merges the parent's `GraphContext` into the child context before overlaying explicit inputs. This is convenient for prototyping but weakens encapsulation — the child now implicitly depends on the parent's context shape.

(See DSL Specification and `StandardNodeCompiler.java` for the scope rules and compilation path.)

10.8.5 Advanced: dynamic subgraphs

For cases where the child graph is not known at compile time, BLOGE provides the `dynamicSubGraph` operator. An upstream node generates DSL text; `dynamicSubGraph` sandbox- validates, compiles, and executes it at runtime:

```
node plan : generateDynamicDsl {
  input { task = ctx.task, history = ctx.history }
  timeout = 3s
}

node executePlan : dynamicSubGraph {
  depends_on = [plan]
  input {
    dslSource = plan.output.dslSource
    context = { task: ctx.task, history: ctx.history, planKind: plan.output.planKind }
  }
  timeout = 15s
  retry = { attempts: 2, backoff: 100ms, strategy: exponential }
}
```

(From `dynamic-agent.bloge`.)

`dynamicSubGraph` is a capability-layer operator registered via `CommonOperators.builder().dynamic().build()`. It validates the generated DSL with `DslSandbox`, enforces complexity limits via `GraphComplexityValidator`, and runs the child through `GraphEngine.executeNestedGraph()`.

Use `subgraph("name")` for compile-time composition. Use `dynamicSubGraph` only when the graph structure must be decided at runtime (e.g. LLM-generated plans, plugin architectures).

10.9 Common Trap

10.9.1 [X] Treating a subgraph node's output as a flat value

```
// WRONG — creditAssessment.output is a Map keyed by terminal node IDs
node decision : UnderwritingOp {
  input {
    riskGrade = creditAssessment.output.riskGrade   // ← WRONG path
  }
}
```

The subgraph node's output is a `Map<String, Object>` where each key is a **terminal node ID** from the child graph. If the child's terminal node is `riskScoring`, the correct path is:

```
// CORRECT — navigate through the terminal node ID
node decision : UnderwritingOp {
  input {
    credit = creditAssessment.output.riskScoring   // ← terminal node ID
  }
```

```
}
```

If the child graph has multiple terminal nodes, each one appears as a separate key. Check the child graph's structure to know which terminal IDs to reference.

10.9.2 [X] Using `scope = parent` as the default

It is tempting to add `scope = parent` everywhere to avoid threading every value through `input {}`. Resist. Isolated scope forces you to declare the contract — which inputs the child actually needs. This makes the subgraph reusable across different parent graphs. Reserve `scope = parent` for genuinely shared infrastructure context (e.g. tracing IDs, tenant keys).

10.10 Guided Rewrite

Look at `international-shipment.bloge`. It uses two parallel subgraphs — `customsClearance` and `routeOptimization` — followed by a fan-in at `bookingConfirmation`.

Questions to work through:

1. **How does the parent pass data to `customsClearance`?** *(Through explicit `input {}` bindings. The child reads `ctx.shipmentId`, `ctx.commodityType`, etc., because those are injected from the parent's input block into the child's `GraphContext`.)*
2. **What is `routeOptimization.output.optimalRouteSelection`?** *(It is the output of the terminal node named `optimalRouteSelection` inside the route-optimization subgraph. The `SubGraphOperator` aggregated it.)*
3. **What happens if you remove the `timeout = 30s` from `customsClearance`?** *(The subgraph node has no time limit. If the child graph hangs — say, an HS-code classification service is down — the parent graph stalls indefinitely at that node. Always timeout subgraph nodes.)*
4. **Could you reuse the `customsClearance` subgraph in a different parent graph?** *(Yes — as long as you register the same child graph under `"customs-clearance"` and provide the expected inputs: `shipmentId`, `commodityType`, `originCountry`, `destinationCountry`, `weightKg`.)*

Now try modifying the graph:

- Extract `trackingSetup` and `sendNotification` into a new subgraph called `"post-booking"`. Register it. Replace the two nodes with a single `node postBooking : subgraph("post-booking") {}` node.
- What inputs must you pass? What terminal node ID will the parent reference?

10.11 Brain Check

1. What syntax references a pre-registered subgraph in DSL? *(`node <id> : subgraph("<name>") { ... }`.)*
2. How do you register a subgraph before compilation? *(`compiler.registerSubGraph("name", graph)` on the `DslCompiler` instance.)*
3. What is the default scope for a `subgraph("name")` node? *(`isolated` — the child sees only explicit `input {}` bindings.)*
4. How does a parent node read a subgraph's result? *(Via `subgraphNode.output.<terminalNodeId>` — the `SubGraphOperator` collects each terminal node's output into a map keyed by node ID.)*
5. When would you use `dynamicSubGraph` instead of `subgraph("name")`? *(When the child graph's structure is determined at runtime — e.g. an LLM-generated plan or plugin-provided workflow. `subgraph("name")` is for compile-time composition.)*
6. *Design question:* Your compliance-check subgraph is used by three parent graphs. The loan-approval parent needs extra AML detail that the other two don't. Should you parameterise the subgraph, create a variant, or push the detail into the parent? *(Prefer parameterisation via subgraph input. If the extra detail fundamentally changes*

the subgraph's pipeline, create a variant. Avoid pushing domain logic into the parent — that defeats the purpose of extraction.)

10.12 Lab

1. Open `loan-approval-subgraph.bloge` and `LoanApprovalSubGraphDslExample.java`.
   - Trace the data flow: what does `creditAssessment.output.riskScoring` resolve to at runtime? *(A map with keys `riskGrade`, `compositeScore`, `summary` — the output of the `riskScoring` node in the credit-assessment child graph.)*
2. Open `smart-ticket-handling.bloge`.
   - The graph has two subgraphs: `sentimentAnalysis` and `escalationWorkflow`. Only one of them always runs. Which one, and why? *(Only `sentimentAnalysis` always runs. `escalationWorkflow` is behind a `branch on determinePriority.output.priority` — it executes only when priority is `"high"`.)*
3. Design a new graph for an e-commerce returns pipeline:
   - Create a `"fraud-screening"` subgraph with three nodes: `orderLookup → returnHistoryCheck → fraudScore`.
   - Create a `"refund-processing"` subgraph with two nodes: `calculateRefund → initiateRefund`.
   - Write a parent graph that runs fraud screening first, branches on the fraud score, and routes clean returns to refund processing.
   - What is the minimal `input {}` contract each subgraph needs?
4. **Bonus:** Open `dynamic-agent.bloge`.
   - What happens if the `plan` node generates invalid DSL? *(The `DslSandbox.parseAndValidate()` call inside `DynamicSubGraphOperator` throws. If `executePlan` has retry configured, the engine retries. If all attempts fail and there is no fallback, the graph fails.)*

10.13 Experiment acceptance card

- **Expected and observed:** The parent consumes terminal outputs only and does not depend on subgraph internals.
- **Failure and recovery:** Bind to a missing internal output; restore the terminal node ID and rerun.
- **Proof boundary:** Proves the subgraph output contract and isolation, not dynamic-subgraph safety.
- **Exercise contract:** Loan subgraph; change one output binding; deliver visible keys and failure reason; stop when only terminal keys are referenced.

10.14 Coding practice

10.14.1 BP-12 — Reuse a contract with one lifecycle owner

Extract a subgraph around stable business input/output, not visual similarity. Callers should not depend on internal node IDs; choose one authoritative outer lifecycle owner.

Try one change: Replace a credit-assessment internal node without changing callers. For a Session/state-machine composition, name the owner of identity, signals, and recovery.

Reasoning, counterexamples, and limits

10.15 Recap

- **`subgraph("name")`** lets you embed a named, pre-built graph inside a parent graph as a single node. The child is registered with `DslCompiler.registerSubGraph()` before compilation.
- **Input flows through `input {}` bindings**, which become the child's `GraphContext`. **Output flows back via**

terminal nodes — the parent reads them as `subgraphNode.output.<terminalNodeId>`.
- Subgraphs default to **`isolated` scope**: no parent context leakage. Use `scope = parent` sparingly — it weakens encapsulation.
- Subgraphs are **regular nodes** in the parent DAG. They support `depends_on`, `timeout`, `retry`, and `fallback` like any other node. Multiple subgraph nodes can run **in parallel** when their dependencies allow.
- For **runtime-generated** graphs, use the `dynamicSubGraph` operator, which sandbox-validates and compiles DSL text at execution time.
- The child graph runs via `GraphEngine.executeNestedGraph()` — inheriting listeners, durability services, and execution-ID chaining from the parent.

10.16 Next Step

In Chapter 11 — Batch and Iteration you will learn how `foreach` and `loop` let you repeat work over collections and polling cycles — both of which compile their body as a nested subgraph under the hood.

10.17 Reference Links

- DSL Specification §5.9 — Scope Mode Compilation — `scope`, default matrix, and compilation behaviour
- Core Architecture — engine internals and nested execution
- Operator Design Specification — operator layer taxonomy (`SubGraphOperator` is a capability-layer operator)
- `loan-approval-subgraph.bloge` — primary example
- `LoanApprovalSubGraphDslExample.java` — Java wiring for subgraph registration
- `LoanApprovalSubGraphExample.java` — Pure Java API version with typed records
- `international-shipment.bloge` — parallel dual-subgraph example
- `smart-ticket-handling.bloge` — subgraph + branch composition
- `order-full-pipeline.bloge` — payment + inventory subgraphs
- `dynamic-agent.bloge` — dynamic subgraph example
- `SubGraphOperator.java` — engine-level nested execution
- `DynamicSubGraphOperator.java` — sandboxed runtime DSL compilation
- `DslCompiler.java` — `registerSubGraph()` and import resolution
- `StandardNodeCompiler.java` — subgraph node compilation logic

CHAPTER
11

Arc 3 — Composition and Time

Chapter 11 — Batch and Iteration

Subgraphs, iteration, suspension, and crash recovery

Goal: compose work across collections, external time, and process restarts.

Promise: *By the end of this chapter you will know how to process every item in a collection with `foreach`, repeat work until a condition is met with `loop`, and combine both constructs in a single graph — all without writing a single `for`/`while` loop in your operator code.*

11.1 Learning Goals

1. Write a `foreach` block that fans out over a collection — in parallel (default) or sequentially — and reference the current item and index inside the body.
2. Write a `loop` block with `until`, `max_iterations`, `delay`, and `carry` to express bounded iteration with state forwarded between rounds.
3. Use the three loop-specific implicit variables — `carry.<path>`, `prev.<nodeId>.<path>`, and `loopIteration` — and explain how each one is resolved at runtime.
4. Consume the aggregate output of a `foreach` or `loop` from a downstream node using `depends_on`.
5. Recognise when `foreach` and `loop` can compose inside the same graph — and when `stream foreach` / `stream loop` are the better choice.

11.2 Prerequisites

- Chapter 10 — Reuse with Subgraphs — you need to understand that `foreach` and `loop` bodies are compiled as nested sub-graphs.

11.3 Source Examples

File	What it shows
`batch-order-processing.bloge`	Simplest parallel `foreach` — fetch orders, validate, process
`batch-order-parallel.bloge`	Parallel `foreach` with a downstream `summarize` node consuming the aggregate output

File	What it shows
`sequential-transfer.bloge`	`foreach … sequential` — one-at-a-time bank transfers
`status-polling.bloge`	Basic `loop` with `until` and `loopIteration` — poll until READY
`cursor-pagination.bloge`	`loop` with multi-field `carry` — paginate an API
`retry-with-backoff.bloge`	`loop`-based custom retry with `carry` and `loopIteration`
`logistics-batch-dispatch.bloge`	`foreach` + `loop` in the same graph — dispatch parcels, then poll status
`basic.bloge` (foreach)	Conformance fixture: minimal `foreach`
`sequential-index.bloge`	Conformance fixture: `foreach … sequential` with index binding
`with-carry.bloge`	Conformance fixture: `loop` with `carry`, `prev`, and `loopIteration`

11.4 Why This Matters

Until now, every node in your graphs ran at most once. Real systems are rarely that simple:

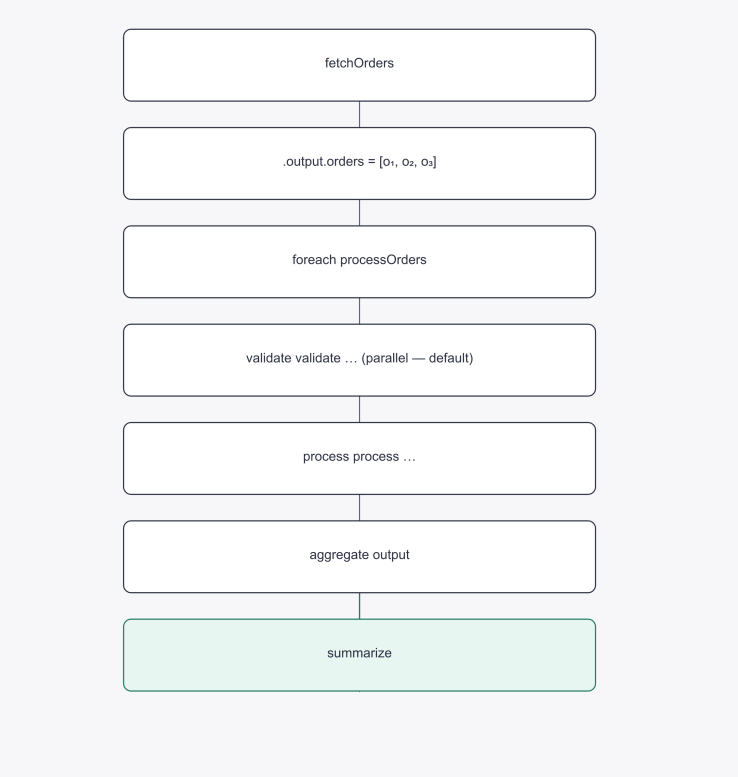
- An e-commerce checkout must validate **every** order line — not just one.
- A data-sync job must paginate through **all** pages of an API, carrying a cursor forward.
- A deployment pipeline must poll an external system until a status flips to READY.

In procedural code you would reach for `for` or `while`. In BLOGE, the equivalent constructs are `foreach` (process each item in a collection) and `loop` (repeat until a condition holds). Both are **hyper-nodes**: they appear as a single node in the outer graph, but internally they compile to a nested sub-graph that the engine schedules, retries, and observes just like any other graph.

The key benefit: iteration semantics — parallelism, ordering, bounded retries, carry state — are declared in the DSL, not buried inside operator code. The engine owns the loop; your operators stay single-item or single-iteration.

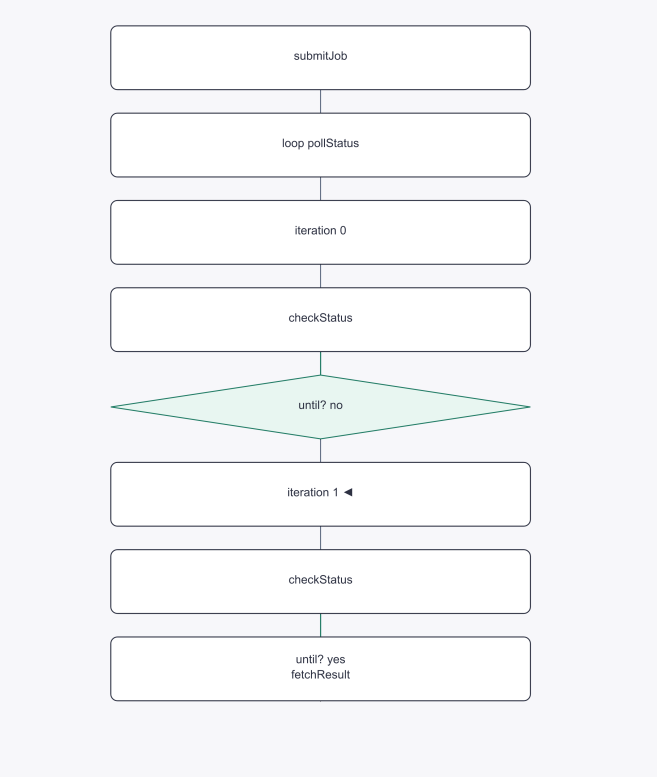
11.5 Mental Model

11.5.1 `foreach` — fan-out over a collection



Think of `foreach` as a **stamp press**: the engine takes the collection, stamps out one copy of the nested sub-graph per item, runs them (in parallel by default, or sequentially), and collects the results into an aggregate output.

11.5.2 `loop` — repeat until done



Think of `loop` as a **turnstile with a counter**: the engine re-executes the nested sub-graph, evaluates `until` after each round, and stops when the predicate is `true` **or** `max_iterations` is reached. Between iterations the engine waits `delay` and threads `carry` state into the next round.

11.6 First Working Example

Here is `batch-order-processing.bloge` — the simplest parallel `foreach`:

```
graph batchOrderProcessing {
  node fetchOrders : OrderFetcher {
    input {
      customerId = ctx.customerId
    }
  }

  foreach processOrders : (order, idx) in fetchOrders.output.orders {
    node validate : OrderValidator {
      input {
        order = order
        index = idx
      }
    }
    node process : OrderProcessor {
      depends_on = [validate]
      input {
        validated = validate.output
      }
    }
```

```
  }
}
```

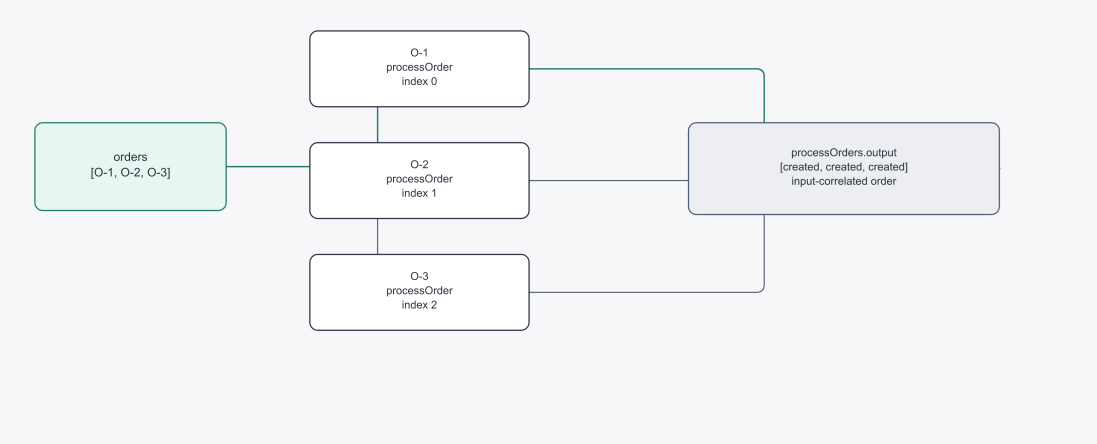
Walk through it:

1. `fetchOrders` runs and produces `output.orders` — a list.
2. The engine enters `foreach processOrders`. For **every** item in the list it creates a copy of the body sub-graph with `order` bound to the current element and `idx` bound to its 0-based position.
3. By default, all copies run **in parallel**.
4. Within each copy, `validate` runs first; `process` depends on it.
5. When all copies finish, `processOrders` itself completes with an aggregate output.

> ***Key insight:*** *You never write a loop. The engine fans out, schedules, and collects. Your operators — `OrderValidator`, `OrderProcessor` — deal with a single item.*

11.7 Complete Story A: `foreach` Produces a Batch Receipt

`orderBatch` loads three orders, runs the same `processOrder` capability for each item, then gives `generateReport` one correlated collection. The result is not "three background jobs happened"; it is a batch receipt the parent can inspect.



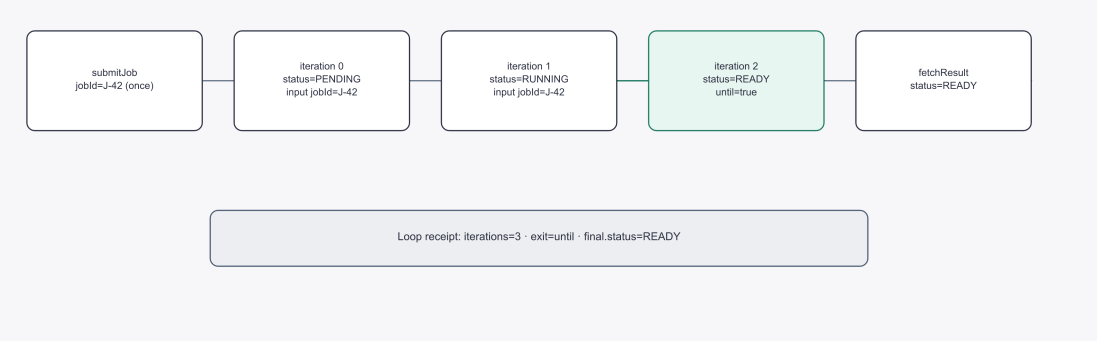
11.7.1 Observe input, item result, and aggregate separately

Level	Example observation	Failure question
Collection	input contains `O-1`, `O-2`, `O-3`	Was an item omitted before dispatch?
Item	index `1` belongs to `O-2`	Did this specific item execute or fail?
Aggregate	`totalProcessed=3`	Did the parent receive a complete receipt?

The companion scenario asserts `generateReport` executed and `totalProcessed=3`. That proves the three-item fixture reached the report under that controlled run. It does not prove unlimited parallelism, ordering under every failure policy, or safe external effects.

11.8 Complete Story B: `loop` Produces an Exit Receipt

A polling loop answers a different question. `submitJob` runs once; the loop reuses `jobId=J-42` and checks status until it sees `READY` or reaches `max_iterations`.



11.8.1 A loop needs an explicit memory and stop rule

Receipt field	Example	Why the parent needs it
`iterations`	`3`	Distinguishes immediate success from repeated polling
`exit`	`until`	Distinguishes success condition from protection limit
final output	`status=READY`	Supplies `fetchResult` without exposing every round

Previous-round outputs — plus `carry` fields when the DSL declares them — are the loop's memory. In this polling story, `jobId` stays a parent input rather than becoming carry state. `until` is the business stop; `max_iterations` is the safety stop. If the loop reaches the safety stop with `PENDING`, do not call it a successful poll merely because the engine terminated normally.

`foreach` expands one collection into many sibling executions. `loop` evolves one logical activity across rounds. Combining them is useful later, but only after each receipt is independently understandable.

11.9 Break It Apart

11.9.1 `foreach` grammar

The formal grammar (DSL Specification §3.1):

```
ForEachDef = "foreach" IDENT ":" ForEachBinding "in" Expression
             "sequential"? "{" body "}"

ForEachBinding = "(" IDENT ("," IDENT)? ")"   // tuple with optional index
               | IDENT                         // bare item variable
```

Part	Example	Meaning
ID	`processOrders`	Name of the hyper-node in the outer graph
Binding	`(order, idx)`	`order` = current item, `idx` = 0-based index. Both are optional — `(order)` or bare `order` are valid.
`in` expression	`fetchOrders.output.orders`	The collection to iterate over
`sequential`	(keyword, optional)	Process items one-at-a-time instead of in parallel
body	`{ node … }`	A nested sub-graph compiled as `<foreachId>__subgraph__`

11.9.1.1 Implicit variables inside `foreach`

Variable	AST type	Runtime context key	Meaning
`<itemVar>` (e.g. `order`)	`ItemPath`	`__item__`	Current collection element
`<itemVar>.<field>` (e.g. `order.amount`)	`ItemPath`	`__item__`	Field on the current element
`<indexVar>` (e.g. `idx`)	`ItemIndex`	`__itemIndex__`	0-based integer position

(See DSL Specification Rules 7–8.)

11.9.1.2 Parallel vs. sequential

In `sequential-transfer.bloge` bank transfers **must** be processed in order so each transfer sees the updated balance from the previous one:

```
foreach processTransfers : (transfer, idx) in fetchTransfers.output.transfers sequential {
  node riskCheck : RiskCheckOperator {
    input {
      amount      = transfer.amount
      fromAccount = transfer.fromAccount
      toAccount   = transfer.toAccount
      index       = idx
    }
  }
  node executeTransfer : TransferExecutionOperator {
    depends_on = [riskCheck]
    input {
      transfer   = transfer
      riskResult = riskCheck.output
    }
  }
  node recordLedger : LedgerRecordOperator {
    depends_on = [executeTransfer]
    input {
      transfer  = transfer
      execution = executeTransfer.output
      index     = idx
    }
  }
}
```

The only syntactic difference is the `sequential` keyword after the `in` expression. At runtime the engine executes one item at a time, in list order.

11.9.1.3 Consuming foreach output downstream

A `foreach` hyper-node exposes an aggregate output. Any downstream node can `depends_on` the foreach and read it. From `batch-order-parallel.bloge`:

```
node summarize : BatchSummaryOperator {
  depends_on = [processOrders]
  input {
    results = processOrders.output
  }
}
```

11.9.2 `loop` grammar

```
LoopDef  = "loop" IDENT "{" LoopBody "}"
```

```
LoopBody = ( "max_iterations" "=" NUMBER
           | "delay" "=" DURATION
           | DependsOn
           | Member
           | CarryBlock
           | UntilClause )*

CarryBlock  = "carry" "{" ( IDENT ":" Expression ","? )* "}"
UntilClause = "until" Expression
```

Part	Example	Meaning
ID	`pollStatus`	Name of the hyper-node in the outer graph
`max_iterations`	`20`	Hard upper bound — the engine stops even if `until` never fires
`delay`	`2s`	Pause between iterations
`until`	`checkStatus.output.status == "READY"`	Early-exit predicate — evaluated **after** each iteration
`carry { … }`	`cursor: fetchPage.output.nextCursor`	Named state forwarded to the next iteration
body	`node …`	A nested sub-graph compiled as `<loopId>__subgraph__`

11.9.2.1 Implicit variables inside `loop`

Variable	AST type	Runtime context key	Meaning
`carry.<path>`	`LoopCarryPath`	`__carry__`	State map from the previous iteration (or initial input)
`prev.<nodeId>.<path>`	`LoopPrevPath`	`__prev__`	Previous iteration's node outputs
`loopIteration`	`LoopIterationRef`	`__loopIteration__`	0-based iteration counter

(See DSL Specification Rules 9–11.)

11.9.2.2 Basic loop — status polling

From `status-polling.bloge`:

```
loop pollStatus {
  max_iterations = 20
  delay = 2s
  depends_on = [submitJob]
  node checkStatus : StatusCheckerOperator {
    input {
      jobId     = submitJob.output.jobId
      iteration = loopIteration
    }
  }
  until checkStatus.output.status == "READY"
}
```

The engine:

1. Runs `checkStatus` (iteration 0).
2. Evaluates `until`. If `status == "READY"` → exit.
3. Otherwise, waits **2 s**, increments `loopIteration`, and repeats.
4. After 20 iterations with no match, the loop stops — `max_iterations` is the safety net.

11.9.2.3 Loop with carry — cursor pagination

From `cursor-pagination.bloge`:

```
loop fetchAllPages {
  max_iterations = 100
  delay = 500ms
  depends_on = [initPagination]

  node fetchPage : PageFetcherOperator {
    input {
      endpoint = ctx.endpoint
      pageSize = ctx.pageSize
      cursor   = carry.cursor
    }
  }
  node transformPage : PageTransformerOperator {
    depends_on = [fetchPage]
    input {
      records      = fetchPage.output.records
      currentTotal = carry.totalRecords
    }
  }

  carry {
    cursor:       fetchPage.output.nextCursor
    totalRecords: transformPage.output.runningTotal
  }
  until fetchPage.output.hasMore == false
}
```

Key points:

- `carry.cursor` and `carry.totalRecords` are read inside the body — they come from the previous iteration's `carry { … }` block.
- On the **first** iteration `carry` fields are unset (null), so the operator should handle the initial case.
- The `carry` block defines what state is **written** at the end of each iteration for the next one to read.

11.9.2.4 Consuming loop output downstream

After the loop completes, its output is the **last iteration's node outputs** keyed by node ID:

```
node finalizeData : DataFinalizerOperator {
  depends_on = [fetchAllPages]
  input {
    totalRecords = fetchAllPages.output.transformPage.runningTotal
    lastCursor   = fetchAllPages.output.fetchPage.nextCursor
  }
}
```

11.9.3 Composing `foreach` and `loop` in one graph

`logistics-batch-dispatch.bloge` shows both constructs working together:

1. A `foreach` fans out over parcels (route planning → dispatch).
2. A `loop` polls aggregate delivery status **after** the foreach completes, connected by `depends_on = [assignRoutes]`.
3. A final `dispatchReport` node depends on the loop.

This demonstrates the key composition rule: `foreach` and `loop` are hyper-nodes — you connect them with `depends_on` just like any other node.

11.9.4 A note on streaming variants

The DSL also supports `stream foreach` and `stream loop`. These emit each item's or iteration's result downstream **as it completes** rather than waiting for the entire batch. For example, from `streaming-batch.bloge`:

```
stream foreach processOrders : item in loadOrders.output.orders {
  buffer = 16
  node processItem : OrderProcessor {
    input { order = item }
  }
}
```

Streaming is an advanced topic beyond the scope of this chapter. The key thing to know is that it exists and that `buffer = N` controls the ring-buffer capacity of the stream edge.

11.10 Throttling and Per-Item Resilience

Parallel `foreach` defaults to "**spawn all items at once**", which is fast on a small list but punishing when you fan out 10,000 items against a downstream service that can only handle 50 concurrent calls. Two `foreach` attributes solve this without rewriting the body:

```
foreach order in orders {
  batch_size      = 25            // at most 25 items running concurrently
  on_item_failure = continue       // one bad item does not kill the batch

  node process : ProcessOrderOperator {
    input { order = item }
  }
}
```

11.10.1 `batch_size`

Value	Meaning
omitted	unbounded — engine spawns one task per item
`batch_size = N` ($N \geq 1$)	engine processes at most N items at a time; the next item starts as one finishes

`batch_size` is local throttling. For **global** caps across the whole engine, see `maxGlobalForeachBatches` below.

11.10.2 `on_item_failure`

Value	Behaviour when an item fails
`abort_all` (default)	propagate the exception; the whole `foreach` fails — matches "fail-fast" semantics
`abort_batch`	finish the items currently in flight, then mark the `foreach` failed; later items are not started
`continue`	record the item failure in the aggregate output and keep processing the rest; the `foreach` itself completes successfully

With `on_item_failure = continue`, the aggregate output preserves the **input order** and contains a sentinel for failed items so downstream nodes can branch on `item.status`:

```
node summarize : SummarizeBatchOperator {
  depends_on = [process]
  input {
    results = process.output       // List with success entries and failures
  }
}
```

11.10.3 Global ceilings — `maxGlobalForeachBatches` and `defaultForeachProtection`

`batch_size` lives on a single `foreach`. In a busy service you also want a **system-wide** ceiling so a hot tenant cannot starve the rest. These two engine-level settings apply across every graph:

```
GraphEngine engine = GraphEngine.builder()
    .maxGlobalForeachBatches(200)         // hard ceiling — across all graphs
    .defaultForeachProtection(            // fallback for foreach nodes
        ForeachProtection.builder()        //   that did NOT set batch_size
            .batchSize(50)
            .onItemFailure(OnItemFailure.ABORT_BATCH)
            .build())
    .build();
```

- `maxGlobalForeachBatches`: cluster-wide upper bound. Once reached, new batches queue until existing ones finish.
- `defaultForeachProtection`: the configuration applied to any `foreach` whose `.bloge` did not specify `batch_size` / `on_item_failure`. Set this in production to avoid a forgotten knob taking the cluster down.

> ***Brain check:*** *when does `on_item_failure = continue` deserve a manual alert? Hint: if 49 of 50 items fail and the batch completes "OK", you have hidden a real outage. Always pair `continue` with a downstream node that checks the failure ratio.*

11.11 Common Trap

11.11.1 [X] Forgetting that `carry` fields are null on the first iteration

```
// WRONG — cursor will be null on iteration 0; the operator might blow up
loop fetchPages {
  max_iterations = 50
  node fetch : PageFetcher {
    input {
      cursor = carry.cursor    // null on first iteration!
    }
  }
  carry { cursor: fetch.output.nextCursor }
  until fetch.output.done == true
}
```

On the **first** iteration there is no previous `carry` — every `carry.<field>` resolves to `null`. Your operator must handle this:

```
// Inside PageFetcher.execute(...)
String cursor = (String) input.get("cursor");
if (cursor == null) {
    // first page — start from the beginning
}
```

This is not a DSL bug. It is the same principle as a loop variable being uninitialized before the first iteration. Design

your operators to treat `null` carry values as ”start fresh.”

11.11.2 [X] Omitting `max_iterations` on a `loop`

`max_iterations` is the **hard safety net**. Without it, a flaky `until` condition could cause the loop to run indefinitely. Always declare a reasonable upper bound, even if you expect `until` to fire well before it.

11.12 What Goes Wrong

11.12.1 `loop` without `max_iterations`

```
loop pollStatus {
  until = checkStatus.output.status == "READY"
  delay = 5s
  // No max_iterations — the compiler rejects this
  body {
    node checkStatus : StatusCheckerOperator { … }
  }
}
```

The parser enforces a safety bound:

```
GraphDefinitionException: Loop 'pollStatus' is missing required 'max_iterations'
```

Fix: Always declare `max_iterations`. Choose a value that covers the realistic worst case with margin:

```
loop pollStatus {
  max_iterations = 60   // 60 × 5s = 5 minute ceiling
  until = checkStatus.output.status == "READY"
  delay = 5s
  body { … }
}
```

11.13 Guided Rewrite

Start from the conformance fixture `with-carry.bloge`:

```
graph g {
  loop processBatch {
    max_iterations = 100
    delay = 5s
    node fetchBatch : BatchFetcher {
      input {
        cursor     = carry.cursor
        prevResult = prev.fetchBatch.result
        iteration  = loopIteration
      }
    }
    carry { cursor: fetchBatch.output.nextCursor }
    until fetchBatch.output.done == true
  }
}
```

Questions to work through:

1. **What are the three implicit variables in use here?** *(`carry.cursor` — carry state from the previous iteration.*

`prev.fetchBatch.result` — the `result` field from the previous iteration's `fetchBatch` output. `loopIteration` — the 0-based iteration counter.)

2. **On iteration 0, what is the value of `carry.cursor`?** *(Null — there is no previous carry yet. The operator must handle this.)*
3. **On iteration 0, what is the value of `prev.fetchBatch.result`?** *(Null — there is no previous iteration. Same rule as carry.)*
4. **If `fetchBatch.output.done` never becomes `true`, what happens?** *(The loop runs 100 times — `max_iterations` is the hard stop — and then exits.)*

Now extend this graph yourself:

- Add a `node transformBatch` inside the loop body that depends on `fetchBatch` and transforms the raw records.
- Update the `carry` block to also carry `totalRecords: transformBatch.output.runningTotal`.
- Add a downstream `node finalize` outside the loop that consumes `processBatch.output.transformBatch.runningTotal`.

Compare your result with `cursor-pagination.bloge` — it follows the same pattern.

11.14 Brain Check

1. What is the default execution mode for `foreach` — parallel or sequential? *(Parallel. Add the `sequential` keyword to switch to one-at-a-time.)*
2. What two implicit variables are available inside a `foreach` body? *(The item variable — e.g. `order` — resolved as `ItemPath`, and the optional index variable — e.g. `idx` — resolved as `ItemIndex`.)*
3. Name the three implicit variables available inside a `loop` body. *(`carry.<path>` — `LoopCarryPath`. `prev.<node Id>.<path>` — `LoopPrevPath`. `loopIteration` — `LoopIterationRef`.)*
4. What happens when `until` never evaluates to `true`? *(The loop runs until `max_iterations` is reached, then exits.)*
5. How does a downstream node consume the output of a `foreach` or `loop`? *(By declaring `depends_on = [<foreach OrLoopId>]` and reading `<id>.output`.)*
6. What is the difference between `loop`-based retry and node-level `retry`? *(`loop`-based retry lets you run arbitrary sub-graphs per attempt, carry custom state, and compute dynamic backoff. Node-level `retry` is a declarative policy that the engine applies automatically to a single operator invocation.)*
7. *Design question:* You have 1 000 orders to process. A parallel `foreach` fans out to 1 000 virtual threads. Your downstream payment gateway allows only 50 concurrent requests. How do you handle this? *(The engine does not have built-in concurrency limiting for `foreach`. Options: batch the orders into groups of 50 in a pre-processing step, use the operator's own rate-limiting or semaphore, or use `foreach sequential` if ordering matters. The design choice depends on whether you need throughput or ordering.)*

11.15 Lab

1. **Parallel foreach — batch validation** Open `batch-order-parallel.bloge`.
   - Add a third node `notifyCustomer` inside the `foreach` body that depends on `deductStock`.
   - Add an `input` binding that reads `order.orderId` and `deductStock.output.remaining`.
   - Predict: does adding this node change the parallelism across items? *(No — each item's sub-graph is still independent. The new node only adds a step within each item's copy.)*
2. **Sequential foreach — ordering matters** Open `sequential-transfer.bloge`.
   - Remove the `sequential` keyword.
   - Predict: what changes? *(All transfers now execute in parallel — risk checks and balance deductions interleave. This is **wrong** for bank transfers where order matters.)*

3. **Loop with carry — build a paginator** Write a new graph from scratch:
   - `node initSearch` provides an initial query.
   - `loop fetchPages` with `max_iterations = 50`, `delay = 1s`.
   - Inside the loop: `node fetchPage` reads `carry.pageToken`.
   - `carry { pageToken: fetchPage.output.nextPageToken }`.
   - `until fetchPage.output.hasMore == false`.
   - After the loop: `node aggregate` consumes `fetchPages.output`.
   - Compare with `cursor-pagination.bloge`.
4. **Compose foreach + loop** Open `logistics-batch-dispatch.bloge`.
   - Trace the data flow: which node's output feeds into `foreach`? Which `depends_on` links `foreach` output to the `loop`? Which links the `loop` to the final report?
   - Add `retry = { attempts: 2, backoff: 500ms, strategy: fixed }` to `planRoute` inside the foreach. Does this affect the loop? *(No — retry is per-node, within each foreach item's sub-graph.)*

11.16 Experiment acceptance card

- **Expected and observed:** foreach produces item receipts; loop produces an exit receipt.
- **Failure and recovery:** Make until unreachable; recover with a cap or corrected carry.
- **Proof boundary:** Proves batching and iteration differ, not that an unbounded stream converges.
- **Exercise contract:** Three-item batch and one loop; change only the exit condition; deliver both receipts; stop with complete correlation and one exit reason.

11.17 Coding practice

11.17.1 BP-13 — Define stop and resource boundaries

State input limits, business termination, capacity, and budgets separately. A method returning or cancellation being requested does not establish actual worker exit or resource release.

Try one change: Test empty input, the limit, over-limit input, and a condition that never becomes true. Distinguish an Agent budget exit from a valid business result and process-local recovery permits from cluster-wide limits.

Reasoning, counterexamples, and limits

11.18 Recap

- **`foreach`** fans out over a collection. Each item gets its own copy of the body sub-graph. Parallel is the default; add `sequential` for ordered execution.
- Inside `foreach`, the **item variable** (e.g. `order`) and optional **index variable** (e.g. `idx`) are implicit — you declare them in the binding and use them in input expressions.
- **`loop`** repeats a nested sub-graph until `until` evaluates to `true` or `max_iterations` is reached. `delay` pauses between rounds.
- Inside `loop`, three implicit variables provide iteration context: `carry.<path>` (forwarded state), `prev.<nodeId>.<path>` (previous iteration outputs), and `loopIteration` (0-based counter).
- Both constructs compile to **hyper-nodes** — they appear as single nodes in the outer graph and connect via `depends_on` like anything else.
- The `carry` block defines state written at the end of each iteration. On the first iteration, carry fields are `null` — operators must handle this.
- Streaming variants (`stream foreach`, `stream loop`) emit results downstream as they complete instead of waiting for the full batch.

11.19 Next Step

In Chapter 12 — Waiting for the World you will learn how BLOGE graphs can pause and wait for external events — timers, human approvals, and webhook callbacks — without blocking a thread.

11.20 Reference Links

- DSL Specification — ForEach and Loop grammar — formal grammar, binding rules, and execution semantics
- DSL Specification — Expression Rules 7–11 — `ItemPath`, `ItemIndex`, `LoopCarryPath`, `LoopPrevPath`, `LoopIterationRef`
- Implicit Declarations Registry — complete list of reserved names and runtime lifecycle
- `batch-order-processing.bloge` — basic parallel foreach
- `batch-order-parallel.bloge` — foreach with downstream summarize
- `sequential-transfer.bloge` — foreach sequential
- `status-polling.bloge` — basic loop with until
- `cursor-pagination.bloge` — loop with carry
- `retry-with-backoff.bloge` — loop-based custom retry
- `logistics-batch-dispatch.bloge` — foreach + loop composition
- `streaming-batch.bloge` — stream foreach example
- `basic.bloge` (foreach) — conformance fixture
- `sequential-index.bloge` — conformance fixture
- `with-carry.bloge` — conformance fixture

CHAPTER
12

Arc 3 — Composition and Time

Chapter 12 — Waiting for the World

Subgraphs, iteration, suspension, and crash recovery

Goal: compose work across collections, external time, and process restarts.

Promise: *You will let a graph release execution resources while it waits for a human approval or webhook, read a real suspended-and-resumed result, and recognize bad correlation keys, duplicate events, and the current `0.9.8-RC1` semantic boundary.*

12.1 Learning goals

By the end of this chapter, you can:

1. Explain the resource difference between blocking, polling, and suspension in business terms.
2. Read `GraphResult` to tell failure, completion, and waiting apart.
3. Choose `wait` or `await` from the shape of the external response.
4. Run a one-variable experiment for a wrong correlation key and a duplicate event.

12.2 This chapter adds one variable

The loop from the previous chapter still runs inside one active execution. This chapter adds one fact: **the prerequisite for the next step is temporarily outside the system.**

A support manager may approve a refund two hours later. A payment gateway may send a callback in ten minutes. A warehouse may report stock tomorrow. The graph should neither keep a worker occupied nor ask a database every few seconds. It must remember what it is waiting for and return control to the runtime.

12.3 Start with the problem: a ticket awaiting manager approval

Customer `C-17` reports a delayed refund. The system creates ticket `T-2048`, classifies it, and notifies a manager. It may send a confirmation only after approval; no response within 48 hours enters timeout handling.

Predict the three implementation shapes before reading code:

Approach	Resources held while waiting	Pressure on other systems	Recovery problem
Block a thread for 48 hours	Worker and in-memory state	None	A process restart erases the wait

Approach	Resources held while waiting	Pressure on other systems	Recovery problem
Poll every 5 seconds	Schedules, connections, and queries	High	Poll progress and deduplication must recover
Suspend and await a signal	One waiting record	Low	Identity, state, and correlation must persist

Suspension is not smarter sleeping. It changes the control model: execution reaches a stable boundary and returns; an external response starts the continuation later.



12.4 Run it first: observe the real suspension boundary

The book probe loads the remote examples file `ticket-approval-wait.bloge`, supplies minimal stand-ins for documentation Operators, and executes twice. The first run stops at the boundary; the second sends an approval signal and continues.

Install the pinned RC1 core and DSL once if they are not in your local Maven repository:

```
cd submodule/bloge
mvn -pl bloge-core,bloge-dsl -am -DskipTests install
```

Then run from the book repository:

```
cd probes/ch11-waiting
mvn test
```

On 2026-09-14, BLOGE commit `cc38fbe5` produced these key lines:

```
boundary.isSuccess=true
boundary.isSuspended=true
boundary.suspendedNodes={waitApproval=wait}
boundary.waitApproval=SUSPENDED
boundary.sendConfirmation=CANCELLED
```

```
resume.isSuccess=true
resume.isSuspended=false
resume.waitApproval=COMPLETED
resume.sendConfirmation=COMPLETED
```

Read the first three lines carefully:

- `isSuccess=true` means no node has failed at this boundary. It does not mean the business process is finished.
- `isSuspended=true` means at least one node is waiting for a signal.
- `suspendedNodes` identifies both the node, `waitApproval`, and the observed suspend key, `wait`.

`sendConfirmation=CANCELLED` is not a rejected ticket. Scheduling converged at the suspension boundary, so dependent work has not run yet. It becomes `COMPLETED` only after the second execution sends the signal.

> ***Observation:*** *"successful and suspended" is a valid intermediate result. Checking only `isSuccess()` reports unfinished work as complete.*

The executable observation is in `WaitingForWorldProbeTest.java`. It is evidence for this chapter, not production durability or concurrency acceptance.

12.5 What does suspension remember?

Think of a restaurant pickup token. A customer need not occupy the counter; the restaurant remembers which order is waiting and where the result belongs.

Real-world object	BLOGE identity	Purpose
One process instance	`executionId`	Finds the same graph execution
One waiting point	`nodeId`	Finds the resume entry
Pickup token	suspend/correlation key	Prevents delivery to the wrong process
Completed work	checkpoint/status	Avoids blindly repeating finished work
External answer	signal payload	Becomes the waiting node's output

Payload without identity cannot find an execution. Execution identity without a node cannot find the continuation. Missing completed state can repeat an effect that already happened.

12.6 Two entry points: `wait` and `await`

Both use suspend-and-resume, but answer different questions.

12.6.1 `wait`: time or a direct signal decides the next step

The ticket example introduces this block:

```
wait waitApproval = 48h after notifyManager {
  signal_key = ctx.ticketId
  on_timeout {
    decision = "auto-approved"
    approver = "system"
  }
}
```

Its intended meaning is: suspend after `notifyManager`; a human may signal the known wait point, while the timer may produce a timeout result. This shape fits a known waiting node governed by a delay or deadline.

12.6.2 `await`: an event name and business key decide the next step

A payment provider does not know BLOGE's `executionId`. It reports "payment confirmed for order `0-42`." `payment-wait.bloge` expresses that correlation:

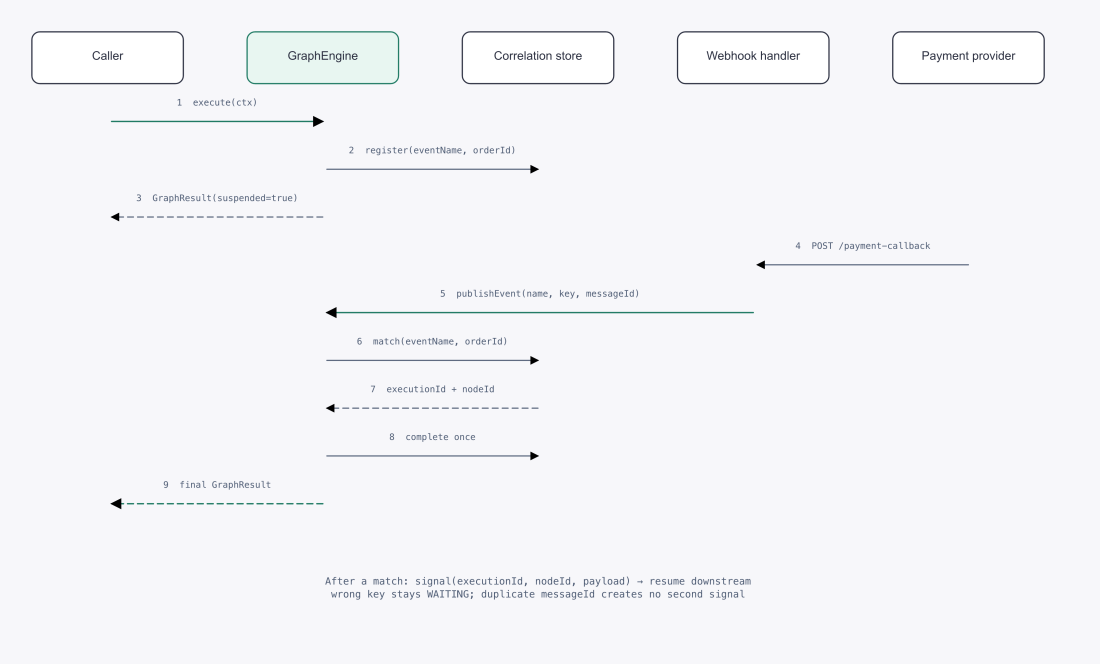
```
await awaitPayment {
  event "payment.confirmed" where orderId = createOrder.output.orderId
  timeout = 15m
  on_timeout { status = "timeout" }
}
```

The webhook handler translates its external protocol into a runtime event:

```
engine.publishEvent(
    "payment.confirmed",
    orderId,
    Map.of("status", "confirmed", "transactionId", transactionId),
    webhookMessageId
);
```

`eventName` says what happened, `orderId` says which business instance it belongs to, and `webhookMessageId` identifies a duplicate delivery. Do not collapse all three identities into one string.

12.7 One picture: suspend, webhook, correlation, resume



Look for two transfers of control:

1. `execute` reaches `await`, records what it is waiting for, and returns that waiting fact to the caller.
2. A webhook calls `publishEvent`; the runtime matches event name and business key, then signals the original `executionId + nodeId` and continues downstream.

The webhook endpoint is not a public "continue this graph function." It authenticates the message, extracts a stable business key, supplies an idempotent message ID, and publishes an event. Correlation state owns the resume identity.

12.8 Why resume does not mean start over

A correct signal path maintains three invariants:

1. Completed nodes stay completed instead of running again unconditionally.
2. Signal payload becomes the waiting node's output and enters the normal downstream data flow.
3. One wait can complete only once; late or duplicate messages cannot replace the first accepted result.

In the probe, `Map.of("decision", "approved", "approver", "manager-7")` becomes `waitApproval` output. `send Confirmation` can then read that answer. This is the exact point where the outside world's response re-enters the graph.

Durable recovery also needs execution checkpoints, a wait store, and a reconstructable graph definition. That is the single new variable in the next chapter; this chapter observes the in-process protocol.

12.9 Break one variable: wrong correlation key

Keep the event name and payload, but change `orderId` from `O-42` to `O-24`:

```
var matched = engine.publishEvent(
    "payment.confirmed", "O-24", payload, "msg-901");
```

The expected observation is neither an exception nor the wrong order resuming:

```
matched=[]
awaitPayment=WAITING
downstream.started=false
```

No match must leave the original correlation waiting. Fix the webhook's key extraction and replay the correct event with a **new** message ID; do not mutate the waiting record to fit a bad message.

This counterexample prevents cross-order wakeups, which are usually more dangerous than a short delay in payments, logistics, and approvals.

12.10 Break one variable: duplicate the event

Now keep the correct key but deliver the same `webhookMessageId=msg-902` twice. RC1's `EventDeduplicationTest` verifies that an identical idempotency key neither reapplies a partial event nor creates a second signal.

Focused results on 2026-09-14:

```
EventDeduplicationTest                Tests run: 5, Failures: 0
EventCorrelationAndOrTest            Tests run: 3, Failures: 0
SuspendSignalTest                    Tests run: 1, Failures: 0
GraphResultContractTest              Tests run: 4, Failures: 0
Total                                Tests run: 13, Failures: 0
```

Duplicates are a normal webhook delivery shape. Use the sender's stable event identity. Generating a random UUID or receive timestamp at ingestion effectively disables deduplication.

12.11 RC1 boundary: stop when source intent and runtime disagree

The probe exposes a gap that the chapter must not hide. The latest remote example declares `signal_key = ctx.ticketId`, but runtime output is `{waitApproval=wait}`. At pinned RC1, `WaitAwaitCompiler` parses `signal_key` and `on_timeout`; its current `compileWait` path constructs `WaitOperator` only from the duration and does not bind those fields.

This chapter therefore verifies:

- The ticket graph reaches `SUSPENDED` on RC1 and resumes through a node signal.
- `GraphResult` separates success, suspension, and downstream work not yet run.
- Core tests cover event correlation, AND/OR aggregation, and duplicate events.

It does not verify:

- The companion DSL's dynamic `signal_key=T-2048` is active in RC1.
- Its `on_timeout` business payload is injected after 48 hours.
- A different process can resume after restart; that requires durable-store evidence in Chapter 13.

The remote examples POM also remains pinned to `0.3.1` and references the renamed `bloge-core-ext` artifact. A direct build fails dependency resolution; overriding the version to RC1 still cannot resolve that old artifact name. File presence is not proof that the entire companion repository passes on RC1.

> ***Engineering rule:*** *When DSL, compiler, and runtime observation disagree, narrow the claim to the observed behavior at a fixed commit and record the gap.*

12.12 Transfer the model to your system

Choose one action whose prerequisite lives outside your system: patient consent, customs clearance, or a candidate's signed offer. Fill in four lines:

Item	Your answer
Waiting identity	How `executionId` and waiting node survive
Business correlation key	Stable field that routes the answer correctly
Duplicate identity	Sender-provided message or event ID
Stop condition	Timeout and the owner authorized to decide its result

Do not implement yet. Let the business and platform owners confirm these four lines. If nobody authorized "auto-approve on timeout," leave it undefined instead of letting the framework invent policy.

12.13 Lab: replace polling with an event wait

Choose one polling flow and change one variable:

1. Input: one stable business key, one external event, and one acceptable timeout.
2. Scope: replace only the polling node with `await`; keep downstream business nodes intact.
3. Run: send a wrong key, a correct key, then the same message ID again.
4. Artifact: keep all three observations, the key's origin, and the timeout owner.
5. Stop if the business cannot define timeout behavior or the sender cannot provide a stable duplicate identity.

12.14 Coding practice

12.14.1 BP-11 — Make time and recovery identity explicit

Model long external waits with suspension and deadlines. Use controlled time, then recover persisted state through a new runtime; lease fencing and external-effect guarantees remain separate.

Try one change: Test before, at, and after the deadline plus a wrong correlation key. Show why an old worker losing commit authority does not establish that a remote effect never happened.

Reasoning, counterexamples, and limits

12.15 Recap

- Suspension releases active execution resources while preserving resume identity and state.
- `wait` is shaped around time or direct signals; `await` around event name and business correlation.
- `isSuccess=true` and `isSuspended=true` may both be true.
- A wrong key stays unmatched; a duplicate event is absorbed by idempotency.
- Example intent, compiler behavior, and runtime output must agree before the claim expands.

The next chapter adds one variable: Chapter 13 — Durable Execution persists the facts needed for an eligible process to continue the same execution after restart.

12.16 Experiment acceptance card

- **Expected and observed:** wait returns suspended; the correct signal resumes the same execution.
- **Failure and recovery:** Send a wrong key or duplicate event; restore the key and handle duplicates idempotently.
- **Proof boundary:** Proves suspend/signal/resume, not a configured durable store.
- **Exercise contract:** Ticket event; change one correlation key; deliver suspended and resumed results; stop when the wrong key does nothing and the right key advances once.

12.17 Exact fact entry points

- `WaitingForWorldProbeTest.java` — this chapter's suspend/resume observation.
- `ticket-approval-wait.bloge` — ticket story DSL.
- `payment-wait.bloge` — webhook correlation DSL.
- `GraphResult.java` — success, suspension, and node-status contract.
- `EventCorrelationAndOrTest.java` — AND/OR resume facts.
- `EventDeduplicationTest.java` — duplicate and early-event boundaries.

CHAPTER
13

Arc 3 — Composition and Time

Chapter 13 — Durable Execution

Subgraphs, iteration, suspension, and crash recovery

Goal: compose work across collections, external time, and process restarts.

Promise: *By the end of this chapter you will understand how BLOGE persists execution state so that a graph can survive process restarts, restore its accepted checkpoints, and continue on another eligible worker when the graph definition and durable stores can be reconstructed.*

13.1 Learning Goals

1. Explain what ”durable execution” means in BLOGE — checkpoint-based persistence of node outputs, suspend contexts, loop snapshots, and execution identity so a graph can resume after a crash.
2. Identify the runtime stores that make durability possible: `ExecutionStore`, `ExecutionCheckpointStore`, `WaitStore`, and `WorkItemStore`.
3. Wire in-memory durable stores into a `GraphEngine` and run a suspend → signal → resume flow across two engine instances.
4. Describe how cold recovery works: execution identity → graph registry lookup → checkpoint reload → continue.
5. Recognise the checkpoint types (`NODE_OUTPUT`, `LOOP_SNAPSHOT`, `FOREACH_PROGRESS`, `SUSPEND_CONTEXT`, `PARTIAL_RESULT`) and when each is written.

13.2 Prerequisites

- Chapter 6 — Resilience by Design — retry and timeout fundamentals.
- Chapter 12 — Waiting for the World — suspend, signal, and event correlation.
- Familiarity with `SuspendableOperator` and `OperatorResult.suspend()`.

13.3 Source Examples

File	What it shows
`RuntimeGraphEngineDurabilityTest.java`	Cross-process signal using runtime stores; two engine instances share state

File	What it shows
`RuntimeGraphRegistryRecoveryTest.java`	Cold recovery from a published graph definition; hash-mismatch rejection; automatic expired-claim recovery
`PaymentWaitExample.java`	End-to-end suspend → publishEvent → resume lifecycle with runtime stores
`payment-wait.bloge`	DSL `await` block with event correlation, timeout, and branching
`RuntimeLoopOperatorDurabilityTest.java`	Loop snapshot checkpoint and resume-from-iteration

13.4 Why This Matters

Everything you've learned so far — nodes, dependencies, branches, resilience, waiting — assumes the JVM stays alive. In production that assumption fails:

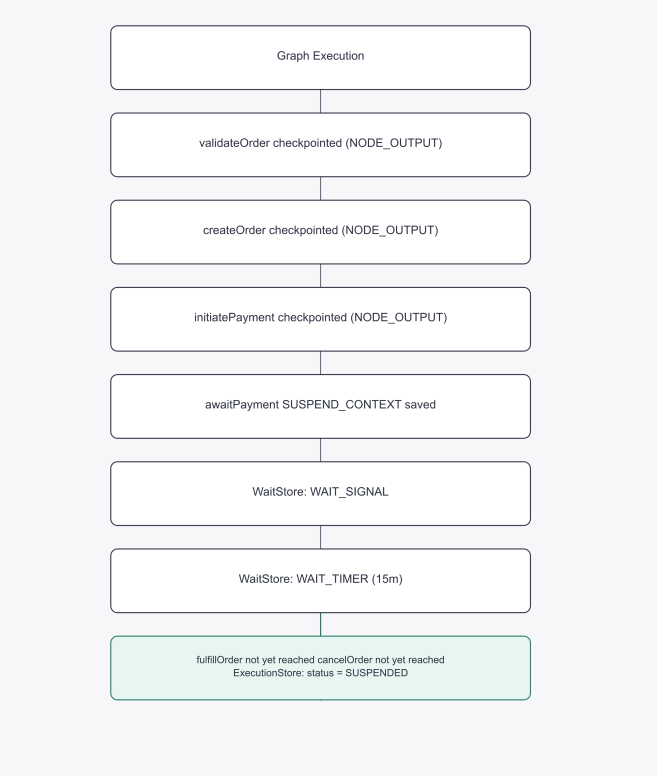
- A **process crashes** mid-execution. Without durability, every in-flight node output is lost and the workflow must restart from scratch.
- A **suspend** can last minutes, hours, or days (waiting for a payment callback, a manager approval). Keeping a thread blocked that long wastes resources and is destroyed by any deploy.
- **Scaling out** means multiple worker processes. A graph that suspended on Worker A must be resumable on Worker B without Worker A ever coming back.

Durable execution solves all three. The engine writes **checkpoints** to a persistent store as nodes complete, and writes **wait records** when a node suspends. On resume — whether triggered by a signal, a timer, or crash recovery — any engine instance can reload those checkpoints and continue from the exact frontier where execution paused.

> ***Key insight:*** *Durability in BLOGE is not "event sourcing" or "replay". The engine **skips** nodes whose checkpoints it accepts and schedules the remaining frontier. An external effect that happened before its checkpoint was committed can still be uncertain; idempotency and reconciliation remain application responsibilities.*

13.5 Mental Model

Think of durable execution as a **bookmark in a recipe book**:



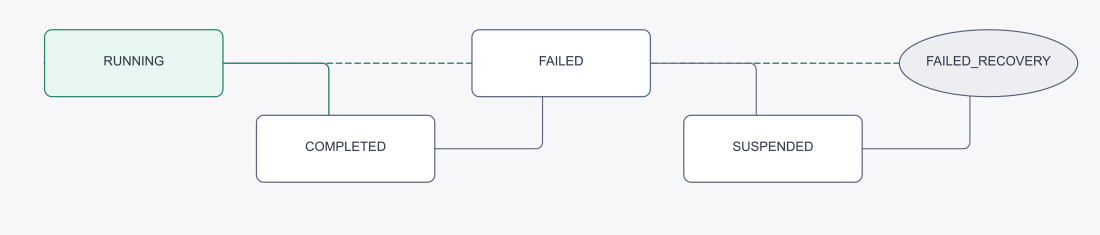
When the payment event arrives, any engine instance can:

1. **Load** the `ExecutionInstance` from the `ExecutionStore`.
2. **Reload** the `NODE_OUTPUT` checkpoints for the three completed nodes.
3. **Resolve** the `WAIT_SIGNAL` wait record.
4. **Resume** from `awaitPayment` → branch → `fulfillOrder`.

The four pillars of this model:

Pillar	Store	What it records
Identity	`ExecutionStore`	Execution lifecycle, status, lease, recovery attempts
Checkpoints	`ExecutionCheckpointStore`	Node outputs, loop snapshots, suspend contexts, foreach progress
Waits	`WaitStore`	Why the execution is paused (signal, timer, task, event)
Work items	`WorkItemStore`	Pending actions: timer-due, event-matched, task-resume

The execution lifecycle as a state diagram:



13.6 The bloge-runtime-spi Module

Before any code, name the seam. The durable execution layer is split into two modules:

- **bloge-runtime-spi** — *interfaces only*. ExecutionStore, ExecutionCheckpointStore, WaitStore, TimerService, GraphRegistryStore, LeaseStore, AuditJournalStore, TaskInboxStore, plus the typed events those stores emit.
- **bloge-durable** — *engine plumbing* that consumes those interfaces: the recovery loop, hash check, checkpoint codec, and the in-memory implementations used in tests.

Database-backed implementations live in their own modules (bloge-durable-mybatis for MyBatis/JDBC). The split means: if you only need to plug in a custom backing store, depend on bloge-runtime-spi alone — you do not have to bring in the engine artifact.

```
<dependency>
  <groupId>com.leanowtech</groupId>
  <artifactId>bloge-runtime-spi</artifactId>
</dependency>
```

13.7 First Working Example

This example is extracted from RuntimeGraphEngineDurabilityTest. It demonstrates a graph that suspends on one engine instance and completes on another — exactly the ”cross-process signal” pattern you need in production.

13.7.1 Step 1 — Create the stores and the graph

```
// Shared durable stores (in production these would be database-backed)
var executionStore    = new InMemoryExecutionStore();
var checkpointStore   = new InMemoryExecutionCheckpointStore();
var waitStore         = new InMemoryWaitStore();
var timerService      = new InMemoryTimerService();

// A suspendable operator that parks execution
SuspendableOperator<Void, String> waitOp = (in, ctx) ->
        OperatorResult.suspend("ck", null, Duration.ofMinutes(10));
Operator<Object, String> completeOp = (in, ctx) -> "C";

var graphBuilder = new GraphBuilder("runtime-cross-process");
var graph = graphBuilder
        .suspendNode("w", waitOp)
        .node("c", completeOp).dependsOn("w")
        .build();

CountDownLatch suspended = new CountDownLatch(1);
AtomicReference<String> executionId = new AtomicReference<>();

ExecutionListener listener = new ExecutionListener() {
    @Override
    public void onGraphStart(String graphName, GraphContext ctx) {
        executionId.set((String) ctx.get(ReservedKeys.EXECUTION_ID));
    }

    @Override
    public void onNodeSuspended(String graphName, String nodeId, String suspendKey) {
        if ("w".equals(nodeId)) {
```

```
            suspended.countDown();
        }
    }
};
```

13.7.2 Step 2 — Execute until suspension on Engine 1

```
var engine1 = GraphEngine.builder()
        .registry(new DefaultOperatorRegistry())
        .executionStore(executionStore)
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .timerService(timerService)
        .listeners(List.of(listener))
        .build();

// The source test runs executeWithOperators() on a background thread so it can
// observe suspension and then resume from a second engine instance.
Thread runThread = Thread.ofVirtual().start(() -> {
    try {
        engine1.executeWithOperators(graph, null, graphBuilder.operators());
    } catch (Exception ignored) {
    }
});

// The execution is now SUSPENDED in the store
assertTrue(suspended.await(5, TimeUnit.SECONDS));
assertNotNull(executionId.get());
assertEquals(ExecutionStatus.SUSPENDED,
    executionStore.get(executionId.get()).orElseThrow().status());
```

13.7.3 Step 3 — Resume on a different Engine 2

```
var engine2 = GraphEngine.builder()
        .registry(new DefaultOperatorRegistry())
        .executionStore(executionStore)          // same stores
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .timerService(timerService)
        .build();

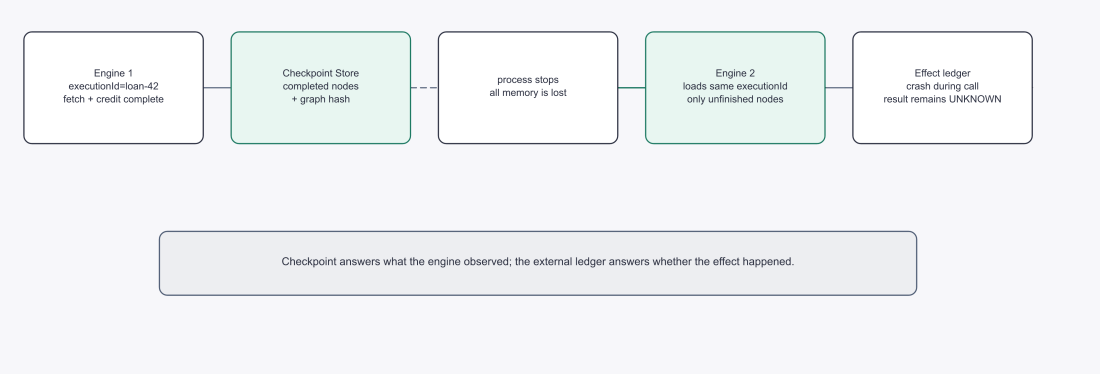
// Signal resumes the suspended node and continues to "c"
engine2.signal(graph, executionId.get(), "w", "resume-data");

// Execution completes
assertEquals(ExecutionStatus.COMPLETED,
    executionStore.get(executionId.get()).orElseThrow().status());
assertEquals("C",
    CheckpointCodec.DEFAULT.deserialize(
        checkpointStore.load(executionId.get(), CheckpointType.NODE_OUTPUT, "c")
            .orElseThrow().payload()));
runThread.join(1_000);
```

Engine 1 and Engine 2 are completely independent objects. They share nothing except the durable stores. This is the fundamental production pattern: one process suspends, another resumes.

13.8 Cold Recovery: What Survives Process Death

Warm resume is easy to overestimate because the original engine still holds objects in memory. A real durable claim needs Engine 1 to disappear, Engine 2 to load the same `executionId`, and completed work to remain completed.



13.8.1 Follow `loan-42`, not an engine object

Moment	Durable fact	Runtime consequence
Engine 1 checkpoints	`fetchApplication` and `checkCredit` completed	Their outputs can be restored
Process stops	Heap objects disappear	In-memory references prove nothing
Engine 2 claims `loan-42`	Graph hash and checkpoint match	Only unfinished nodes become schedulable
Recovery completes	New checkpoint records later work	Same logical execution continues

The identity belongs to the execution record, not to either engine instance. The graph definition is loaded again and checked against retained state before unfinished work is dispatched.

13.8.2 The effect can still be `UNKNOWN`

Suppose Engine 1 called `approveLoan`, the lender committed the approval, and the process died before the completion checkpoint. Engine 2 sees an unfinished node. The checkpoint cannot tell whether the external effect happened.

Recovery therefore needs the Chapter 6 contract: stable idempotency key, reconciliation read, and an explicit policy for retry or manual repair. Durable execution prevents completed **engine work** from being scheduled again when a completion was retained; it does not manufacture exactly-once effects across an uncoordinated external system.

13.9 Break It Apart

13.9.1 Checkpoint types

The engine writes different checkpoint types at different lifecycle moments. These are defined in `CheckpointType`:

Type	When written	Purpose
`NODE_OUTPUT`	Each node completes	Serialized output — skipped on resume
`SUSPEND_CONTEXT`	An operator returns `OperatorResult.suspend()`	Suspend key, partial output, graph context snapshot
`PARTIAL_RESULT`	Alongside `SUSPEND_CONTEXT`	Intermediate output visible while suspended

Type	When written	Purpose
`LOOP_SNAPSHOT`	After each loop iteration	`completedIterations` + `carryStateJson` so loops resume mid-way
`FOREACH_PROGRESS`	After each sequential foreach item	Per-item progress so foreach can skip finished items
`EVENT_CORRELATION`	When an `await` registers a matcher	Correlation key so events route to the right execution
`SESSION_PHASE_SNAPSHOT` / `SESSION_ROUND_SNAPSHOT`	Session phase/round transitions	Session-specific state (Chapter 14 — Multi-Turn Sessions)

13.9.2 Execution lifecycle states

`ExecutionStatus` tracks where an execution is in its lifecycle:

- `RUNNING` → nodes are actively executing
- `SUSPENDED` → at least one node returned `OperatorResult.suspend()`
- `COMPLETED` → all terminal nodes finished successfully
- `FAILED` → a node failed after all resilience attempts
- `FAILED_RECOVERY` → automatic crash recovery exceeded `maxRecoveryAttempts`
- `CANCELLED` / `TERMINATED` → explicitly stopped

13.9.3 Wait types

`WaitType` records **why** an execution is suspended:

- `WAIT_SIGNAL` — waiting for `engine.signal()` with a matching key
- `WAIT_TIMER` — a timeout-based auto-resume
- `WAIT_EVENT` — waiting for `engine.publishEvent()` with a correlation match
- `WAIT_TASK` — waiting for a user-task completion
- `WAIT_PHASE_TIMEOUT` — a session phase-level timeout fired
- `WAIT_ROUND_TIMEOUT` — a session round-level timeout fired
- `WAIT_RETRY_BACKOFF` — a delayed retry scheduled as a durable work item

If sessions are new to you, Chapter 14 — Multi-Turn Sessions unpacks how phase/round timeouts and session snapshots fit on top of the same durable machinery. State-machine durability takes a different path: `StateMachineCheckpoint` bridges through the generic execution-checkpoint SPI instead of introducing a new core checkpoint type, which you will meet in Chapter 15 — State Machines.

13.9.4 The DurableManager bridge

`DurableManager` is the internal bridge that the `GraphEngine` uses to coordinate writes across all stores. You don't call it directly — the engine calls it when:

- A graph starts → `initExecution()`
- A node completes → `saveNodeCompletion()`
- A node suspends → writes suspend context plus the corresponding wait/timer records
- A loop iteration finishes → `saveLoopSnapshot()`
- A resume completes → cleanup suspend context + wait records

---

13.10 Common Trap

13.10.1 [X] Assuming checkpoint data keeps its original Java type

When you suspend and later resume, checkpoint payloads are serialized to JSON and deserialized back. If your operator output was a `PaymentEvent` record, after deserialization it will be a `Map<String, Object>` unless you use a typed

`CheckpointCodec`.

```
// FRAGILE — will ClassCastException after resume
PaymentEvent event = results.get("awaitPayment", PaymentEvent.class);

// SAFE — handle both fresh-run and checkpoint-restored shapes
Object raw = results.getRaw("awaitPayment");
String txnId;
if (raw instanceof PaymentEvent pe) {
    txnId = pe.transactionId();
} else if (raw instanceof Map<?,?> m) {
    txnId = (String) m.get("transactionId");
}
```

This pattern appears throughout the `PaymentWaitExample`.

Fix: Either use `getRaw()` with pattern matching, or wire a `JacksonCheckpointCodec` with typed deserialization into `GraphEngine.builder().checkpointCodec(...)`.

13.11 What Goes Wrong

13.11.1 Checkpoint deserialization failure after schema change

You deploy version 2 of an operator that changes the output type from `record OrderOutput(String orderId, int total)` to `record OrderOutput(String orderId, BigDecimal total, String currency)`.

An in-flight execution was checkpointed with the v1 schema. When recovery loads the checkpoint:

```
Recovery error: cannot deserialize the checkpoint for node 'createOrder'
  because required field 'currency' is missing from the stored payload.
```

Fix: Make schema changes backward-compatible (add optional fields, don't remove required ones). For breaking changes, drain in-flight executions before deploying, or implement a `CheckpointCodec` that handles version migration.

13.12 Guided Rewrite

Open `RuntimeGraphRegistryRecoveryTest`. The test `publishesDslGraphAndRecoversColdSignalWithoutCallerGraph` demonstrates **cold recovery** — resuming without the caller passing a `Graph` object:

```
// Engine 1: execute with a DSL graph — the engine publishes the
// GraphDefinition to graphRegistryStore automatically
GraphEngine engine1 = GraphEngine.builder()
        .registry(registry)
        .executionStore(executionStore)
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .graphRegistryStore(graphRegistryStore)        // ← enables cold recovery
        .addGraphDefinitionCodec(codec)                // ← how to decode DSL
        .timerService(timerService)
        .build();

engine1.execute(graph, new GraphContext(Map.of("sessionId", "SESSION-001")));
// Graph suspends at "wait" node

// Engine 2: signal WITHOUT passing the graph object
GraphEngine engine2 = GraphEngine.builder()
        .registry(recoveryRegistry)
```

```
        .executionStore(executionStore)
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .graphRegistryStore(graphRegistryStore)
        .addGraphDefinitionCodec(codec)
        .timerService(timerService)
        .build();

// The engine loads the graph from graphRegistryStore using the
// execution's graphVersion + graphHash binding
engine2.signal(executionId, "wait", Map.of("message", "resume"));
```

Questions to consider:

1. **What happens if the graph definition changes between suspend and resume?** The engine compares the `graphHash` stored in `ExecutionIdentity` with the hash of the published `GraphDefinition`. If they don't match, recovery fails with a `"hash mismatch"` error. This is tested in the `rejectsColdRecoveryWhenPublishedGraphHashChanges` test case.
2. **What about automatic crash recovery?** When `RecoveryConfig` is enabled and an execution's lease expires (the worker crashed), another engine instance can automatically claim the expired execution and re-execute it. The `recoveryAttempts` counter is incremented each time. If the budget is exhausted, the status transitions to `FAILED_RECOVERY`.
3. **Where are graph definitions stored?** In the `GraphRegistryStore`, which persists `graphName + graphVersion + graphHash + source`. In production this is backed by the `bd_graph_definition` database table.

13.13 Graph Definition Source

When the engine publishes a graph to the `GraphRegistryStore`, it needs to know **where the definition came from** — a DSL file on disk, a Java builder call, or a registry import. The `GraphDefinitionSource` interface (`com.leanowtech.bloge.core.runtime.registry.GraphDefinitionSource`) captures that provenance:

```
public interface GraphDefinitionSource {
    String sourceId();          // unique identifier for this source
    String sourceType();        // e.g. "dsl-file", "java-builder", "registry"
    Optional<String> content(); // raw source content if available
}
```

Attach a source when building a graph for registry publication:

```
var source = new FileBasedGraphDefinitionSource(
        "file:///graphs/order-process.bloge", dslContent);

Graph graph = Graph.builder("orderProcess")
    .definitionSource(source)
    // ... nodes and edges ...
    .build();

engine.publishGraphDefinition(graph);
```

The three fields serve different recovery needs:

Field	Purpose
`sourceId()`	Uniquely identifies the origin — a file path, a classpath resource, or a registry URI
`sourceType()`	Tells the registry *how* the definition was produced (`"dsl-file"`, `"java-builder"`, `"registry"`)

Field	Purpose
`content()`	Optionally stores the raw DSL text so the definition can be reconstructed without the original file

Without a `GraphDefinitionSource`, the registry stores the compiled graph structure but loses the trail back to the original artifact. For production systems that need audit trails or multi-cluster replication, always attach a source.

13.14 Checkpoint Codec SPI

By default the engine serializes checkpoint payloads as JSON. For many workflows this is fine, but production systems sometimes need:

- **Custom compression** to reduce storage costs for high-volume workflows
- **Encryption** for checkpoints that contain sensitive data (PII, financial records)
- **Compatibility layers** that handle schema migration across deployments

The `CheckpointCodec` SPI (`com.leanowtech.bloge.core.checkpoint.CheckpointCodec`) lets you replace the default serialization:

```
public interface CheckpointCodec {
    byte[] encode(ExecutionCheckpoint checkpoint);
    ExecutionCheckpoint decode(byte[] data);
}
```

Wire it into the engine builder:

```
GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .checkpointCodec(new JsonCheckpointCodec())  // custom serialization
    .build();
```

A typed codec also eliminates the `Map<String, Object>` problem described in the Common Trap section above — if your codec knows the target types, restored checkpoints come back as proper Java records instead of raw maps.

13.15 Operator Fingerprint Mismatch Policy

Cold recovery still rejects a graph-definition hash mismatch. A separate question appears when a completed node has a saved Operator fingerprint but the registered Operator implementation now has another fingerprint. `VersionMismatchPolicy` controls only that checkpoint decision:

Policy	Behaviour
`WARN`	Restore the checkpoint and log the Operator mismatch; this is the default.
`RERUN`	Skip that checkpoint so the node is scheduled again.
`FAIL`	Abort resume with `OperatorVersionMismatchException`.

Wire an explicit choice into the engine builder:

```
GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .versionMismatchPolicy(VersionMismatchPolicy.FAIL)
    .build();
```

Choose by effect semantics, not environment labels. `RERUN` is safe only when the node can repeat without duplicating an irreversible effect. `WARN` accepts old output under new code and therefore needs an explicit compatibility decision. `FAIL` stops automatic recovery and leaves migration or operator intervention outside the engine.

Production warning — schema migration is a separate version axis
Treat the dialect-specific Flyway migrations packaged in `bloge-durable-mybatis` as authority. `/actuator/bloge/schema` reports `UP_TO_DATE`, `PENDING`, or `FAILED`; apply required schema migrations before deploying binaries that depend on them. In RC1, V25 only migrates legacy session enum values. Recovery leases were introduced by V17, the audit journal table by V9, and lease fencing by V26. Appendix H keeps the operational migration checklist outside this narrative chapter.

13.16 Brain Check

1. What are the four runtime stores that enable durable execution? *(ExecutionStore, ExecutionCheckpointStore, WaitStore, WorkItemStore.)*
2. What happens to already-completed nodes when an engine resumes a suspended execution? *(They are skipped. The engine loads their NODE_OUTPUT checkpoints and populates the NodeResults without re-executing the operators.)*
3. Why does the engine reject a cold recovery when the graph hash changes? *(Because the graph structure may have changed — new nodes, removed edges, different branching — so the checkpointed node outputs may not correspond to the current graph. Resuming would produce undefined behaviour.)*
4. What `ExecutionStatus` does an execution transition to when automatic recovery exceeds `maxRecoveryAttempts`? *(`FAILED_RECOVERY` — a terminal status.)*
5. If a `SuspendableOperator` returns `OperatorResult.suspend("key", partial, Duration.ofMinutes(5))`, how many wait records are created? *(Two: one `WAIT_SIGNAL` for the manual signal resume path, and one `WAIT_TIMER` for the 5-minute timeout auto-resume path.)*
6. *Design question:* Your durable workflow processes insurance claims that take 2–4 weeks. During that time, you will deploy multiple code updates. What is your strategy for ensuring in-flight executions survive deployments? *(Keep operator output schemas backward-compatible. Use `GraphRegistryStore` to version graphs so recovery can find the correct graph definition. Design compensation operators to be idempotent so that a recovered execution re-running a node is safe. Test recovery in staging with long-running executions before shipping.)*

13.17 Lab

Goal: Build a two-phase durable payment flow and simulate a crash between phases.

1. **Create a graph** with three nodes:
   - `createOrder` — a normal operator that returns an order ID.
   - `awaitPayment` — a `SuspendableOperator` that suspends with `OperatorResult.suspend("payment:" + orderId, null, Duration.ofSeconds(30))`.
   - `fulfillOrder` — depends on `awaitPayment`, returns a shipment ID.
2. **Wire runtime stores** (use the in-memory implementations):
```
var executionStore    = new InMemoryExecutionStore();
var checkpointStore   = new InMemoryExecutionCheckpointStore();
var waitStore         = new InMemoryWaitStore();
```
3. **Execute** the graph on `engine1`. Verify that:
   - The execution status is `SUSPENDED`.
   - A `NODE_OUTPUT` checkpoint exists for `createOrder`.
   - A `SUSPEND_CONTEXT` checkpoint exists for `awaitPayment`.
   - Wait records contain both `WAIT_SIGNAL` and `WAIT_TIMER`.
4. **Create `engine2`** with the same stores but a fresh `DefaultOperatorRegistry`. Signal the suspended node:
```
engine2.signal(graph, executionId, "awaitPayment", paymentData);
```

5. **Assert** that:
   - The execution completes.
   - `fulfillOrder` has a `NODE_OUTPUT` checkpoint.
   - All wait records have been cleaned up.

Stretch: Enable `GraphRegistryStore` and try calling `engine2.signal()` without passing the `graph` object — use only the execution ID.

13.18 Experiment acceptance card

- **Expected and observed:** A new process takes over using identity, definition, and checkpoint.
- **Failure and recovery:** Change the definition version; recover with a match or explicit migration.
- **Proof boundary:** Proves engine cold recovery, not known external-effect state.
- **Exercise contract:** Interrupted checkpoint; change one version; deliver accepted/rejected receipts; stop when completed nodes do not rerun and UNKNOWN stays separate.

13.19 Coding practice

13.19.1 BP-11 — Make time and recovery identity explicit

Model long external waits with suspension and deadlines. Use controlled time, then recover persisted state through a new runtime; lease fencing and external-effect guarantees remain separate.

Try one change: Test before, at, and after the deadline plus a wrong correlation key. Show why an old worker losing commit authority does not establish that a remote effect never happened.

Reasoning, counterexamples, and limits

13.20 Recap

- **Durable execution** means persisting node outputs, suspend contexts, wait records, and execution identity so a graph survives crashes and can be resumed by any engine instance.
- The **four runtime stores** — `ExecutionStore`, `ExecutionCheckpointStore`, `WaitStore`, and `WorkItemStore` — are the foundation. Use in-memory implementations for development; MyBatis-backed implementations for production.
- On resume, the engine **loads checkpoints and skips completed nodes**. It does not replay already-finished work.
- **Cold recovery** uses `GraphRegistryStore` to reload the graph definition from its `graphVersion + graphHash` binding — no caller-supplied `Graph` needed.
- The engine rejects recovery when the **graph hash changes**, preventing checkpoint/graph mismatches.
- **Automatic crash recovery** uses lease expiry and `RecoveryConfig` to let healthy workers claim expired executions, with a `recoveryAttempts` budget.
- Checkpoint payloads are **JSON-serialized**. Use `getRaw()` with pattern matching or a custom `CheckpointCodec` to handle type differences between fresh runs and restored executions.

13.21 Next Step

In Chapter 14 — Multi-Turn Sessions you will move from durable one-shot graphs to longer-lived interactions. The same checkpointing machinery you learned here becomes the substrate for phase/round sessions, and then for explicit state machines in Chapter 15.

13.22 Reference Links

- `bloge-durable` README — module overview, store table, quick start, schema migrations
- `bloge-durable-codec` README — `JacksonCheckpointCodec` and custom serialization
- `CheckpointCodec` SPI — the codec interface
- `ExecutionCheckpointStore` — checkpoint store SPI
- `ExecutionCheckpoint` — checkpoint record with identity, type, payload, and operator fingerprint
- `CheckpointType` — `NODE_OUTPUT`, `LOOP_SNAPSHOT`, `FOREACH_PROGRESS`, `SUSPEND_CONTEXT`, etc.
- `ExecutionIdentity` — tenant, namespace, business key, graph binding, routing
- `ExecutionInstance` — lifecycle record with lease, version, recovery attempts
- `ExecutionStatus` — `RUNNING`, `SUSPENDED`, `COMPLETED`, `FAILED`, `FAILED_RECOVERY`
- `WaitType` — `WAIT_SIGNAL`, `WAIT_TIMER`, `WAIT_EVENT`, `WAIT_TASK`, `WAIT_PHASE_TIMEOUT`, `WAIT_ROUND_TIMEOUT`, `WAIT_RETRY_BACKOFF`
- `OperatorResult` — `Completed` vs `Suspended` sealed result type
- `SuspendableOperator` — operator contract for nodes that may suspend
- `DurableManager` — internal bridge between engine and durable stores
- `DurableStoreFactory` — production store wiring with MyBatis + Flyway
- Core Architecture — engine internals and package structure

Arc 3 Recap — Composition and Time (Chapters 10–13)

CHAPTER

—

Arc 3 Recap — Composition and Time (Chapters 10–13)

You can now compose large workflows out of reusable pieces, iterate over collections, suspend for external events, and survive process restarts.

0.1 What You Learned

Chapter 10 — Reuse with Subgraphs introduced the `subgraph("name")` syntax. You learned to register a pre-built `Graph` with `DslCompiler.registerSubGraph()`, embed it in a parent graph, and control scope isolation (isolated vs. parent). You also saw when a `dynamicSubGraph` — runtime-generated DSL — is the better choice.

Chapter 11 — Batch and Iteration showed two loop primitives: `foreach` (iterate over a collection with one subgraph execution per item) and `loop` (repeat with `max_iterations`, `delay`, and `until`). You learned how batch results are collected and how `loopIteration` gives each pass a zero-based counter.

Chapter 12 — Waiting for the World added suspension. With `await` a graph pauses without keeping a worker thread blocked, and resumes only when an external signal arrives or a timeout elapses. You practiced correlating events by key and branching on whether the signal arrived or timed out.

Chapter 13 — Durable Execution added the stores and identities needed for restart recovery. Accepted checkpoints let an eligible worker skip completed nodes and continue the remaining frontier; effects outside a committed checkpoint still need idempotency or reconciliation.

0.2 What You Should Be Able to Do Now

- Extract a reusable pipeline into a named subgraph and embed it in multiple parent graphs.
- Process a collection with `foreach` and a polling loop with `loop`, choosing the right primitive for each use case.
- Suspend a graph with `await`, correlate an external event, and handle timeout gracefully.
- Wire durable stores into a `GraphEngine` and demonstrate suspend → crash → cold-recovery → resume.
- Explain the relationship between checkpoints, execution identity, and the graph registry.

0.3 Common Mistakes at This Stage

Mistake	Why It Happens	Fix
Using parent scope when you mean isolated	"The subgraph needs one value from the parent"	Default to isolated scope and pass the needed value explicitly through the subgraph input binding. Parent scope leaks the entire context.
Infinite loops without `until` or `max_iterations`	Forgetting that `loop` defaults to unbounded	Always set `max_iterations` as a safety net, even if `until` normally exits first.
Treating `await` like `Thread.sleep()`	Confusing suspension with blocking	`await` leaves no worker thread blocked while waiting; the persisted wait, timer, and correlation state still consume system resources.
Skipping durable stores in development	"I'll add durability later"	Use in-memory store implementations to practice the API and recovery path, while remembering that their state is lost with the process.
Not testing the resume path	Only testing the happy path where nothing crashes	Write at least one test that executes halfway, creates a new engine, and resumes from checkpoint.

0.4 Key Vocabulary Additions

Term	Meaning
Subgraph	A pre-built `Graph` registered by name and embedded inside a parent graph as a single node.
`foreach`	Iterates over a collection, executing the body once per item and collecting results.
`loop`	Repeats the body up to `max_iterations` times, with optional `delay` and `until` exit condition.
`loopIteration`	Built-in zero-based counter available inside every loop body.
`await`	Suspends the graph and releases its thread until an external signal or timeout.
Checkpoint	A persisted snapshot of node outputs and execution state after each node completes.
Cold recovery	Resuming a graph on a different engine instance by loading its checkpoint from durable storage.

0.5 Try It: Choose the Right Time Model

An order flow processes 200 items, polls inventory up to five times, and may wait two days for customer confirmation. Assign **foreach**, **loop**, or **await** to each part, then mark which state must survive process restart. Explain why replacing `await` with a sleeping loop would hold the wrong resource and hide the external signal boundary.

0.6 Where to Go Next

Arc 4 shifts from durable one-shot execution to longer-lived orchestration patterns: sessions, state machines, and nested lifecycle ownership. Start with Chapter 14 — Multi-Turn Sessions.

Learning Arc

Arc 4 — Lifecycle Models

Sessions, state machines, nested ownership, and bounded agents

Goal: choose one explicit owner for every long-lived decision.

CHAPTER
14

Arc 4 — Lifecycle Models

Chapter 14 — Multi-Turn Sessions

Sessions, state machines, nested ownership, and bounded agents

Goal: choose one explicit owner for every long-lived decision.

Promise: *By the end of this chapter you will know when to move from a one-shot graph to a long-lived `session`, how to model `ONCE` and `ROUND` phases, and how to keep a multi-turn interaction recoverable instead of rewriting the same suspend/resume glue over and over.*

14.1 Learning Goals

1. Distinguish a graph that suspends once from a `session` that spans multiple signals, phase boundaries, or both.
2. Model a `session` as ordered `phase` steps with `ONCE` and `ROUND` behavior.
3. Use `ctx.round.input`, `yield_on`, `until`, and `then` transitions to shape a real conversation.
4. Configure idle timeout, history retention, ownership, and recovery without losing track of the runtime model.
5. Decide when `session` is the right abstraction — and when a plain graph from Chapter 12 or Chapter 13 is still simpler.

14.2 Prerequisites

- Chapter 12 — Waiting for the World — you need the suspend/signal mental model first
- Chapter 13 — Durable Execution — sessions build on the same checkpoint and recovery foundation
- Chapter 7 — Designing Good Operators — multi-turn flows stay readable only when operators are small and explicit

14.3 Source Examples

File	What it shows
`ch13/customer-service-session.bloge`	Canonical `session` DSL: greeting → triage rounds → solve → wrap-up
`ch13/interactive-review-session.bloge`	A smaller collect → review rounds → finalize flow used in the guided rewrite

File	What it shows
CustomerServiceSessionExample.java	Fluent Java API equivalent with PhaseBuilder.once(...) and PhaseBuilder.round(...)
bloge-session-ext/README.md	In-memory runtime, manual restore boundary, and nested-state-machine note
docs/session-phase-round-specification.md	Exact DSL and runtime contract for session / phase / round behavior

14.4 Why This Matters

In Chapter 12 you learned how one graph can pause and later resume. That solves a lot of real work: wait for approval, wait for payment, wait for an external event.

But some workflows are not just "one graph plus one wait". They are **conversations**:

- greet the user
- ask for more detail
- wait for the reply
- ask again if needed
- hand off or close the interaction

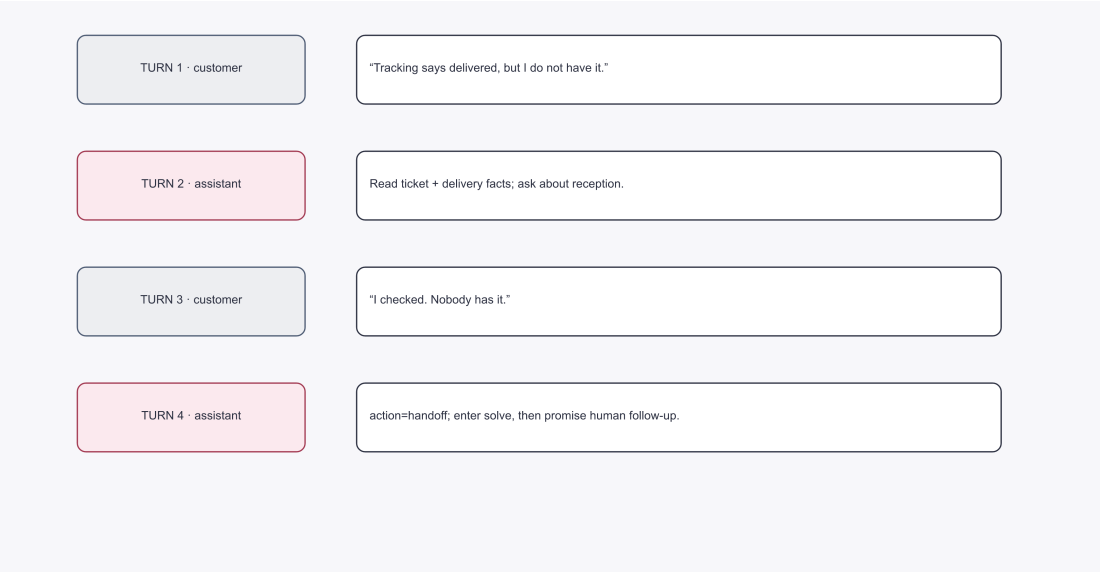
The same behavior can be encoded with `await`, branches, and context plumbing inside one graph. The trade-off is that conversation identity, round boundaries, and recovery state remain implicit in graph structure, making them harder to inspect and explain.

A `session` gives that conversation a first-class runtime model. Instead of forcing every turn into one giant DAG, you say:

- these steps happen once
- this step repeats round by round
- these signals belong to the same business interaction
- this whole interaction should survive restart, timeout, and resume

14.5 Read the Conversation Before the Session Model

A customer says the parcel is marked delivered but missing. The assistant reads facts, asks one narrowing question, receives the answer, and hands the case to a human path. That four-turn exchange is the problem the Session must retain—not an abstract desire to "support chat".



14.5.1 Translate each conversational fact once

Conversation fact	Session owner	Why it is retained
Ticket and customer identity	Session context	Stable across every round
Current customer message	`ctx.round.input.userMessage`	Belongs only to this round
Assistant answer and `done`	`respond.output`	Determines yield and exit
`action=handoff`	Phase transition condition	Moves ownership from triage to solve
Final resolution	wrap-up output	Stable terminal result for the caller

Turn 2 yields after `respond`, so the caller can display the question while the session remains in `triage`. Turn 3 becomes the next round input. Turn 4 returns `done=true` and `action=handoff`, so the phase transitions to `solve`; it does not silently mutate a ticket from inside the language model response.

Now the vocabulary has a reason to exist: a **phase** owns a stage of the conversation, a **round** owns one repeated exchange, `yield_on` marks the observable pause, and `until` says when repetition ends.

14.6 Mental Model

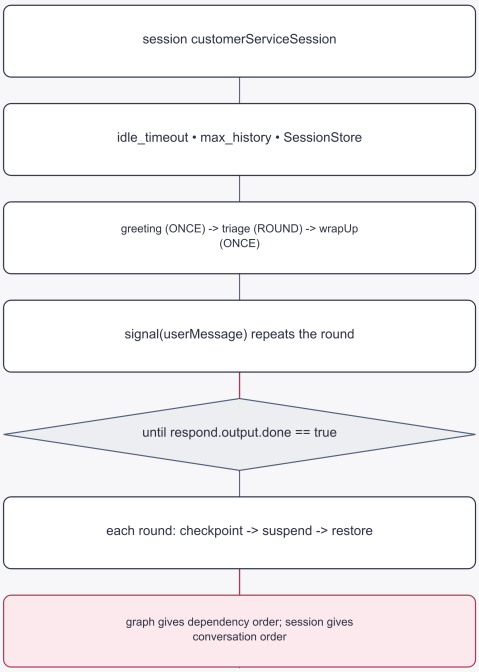
Think of a `session` as a **container for multiple graph executions**.

Piece	Responsibility
`SessionGraph`	Immutable definition of the whole interaction
`phase`	A named step in the interaction lifecycle
`ONCE` phase	Execute the embedded graph one time, then transition
`ROUND` phase	Execute once per external signal payload, suspending between rounds
`SessionExecutor`	Starts, signals, restores caller-provided state, and terminates in-memory sessions
`DurableSessionManager`	Coordinates checkpoint stores, definition lookup, leases, and restart recovery when durability is required

Two phase types do almost all the work:

Phase type	Best for	Runtime behavior
`ONCE`	greeting, enrichment, finalize, wrap-up	Run the graph once, write outputs, resolve `then`
`ROUND`	clarify-until-done, iterative review, menu loops	Wait for a signal, run the round graph, evaluate `until`, either suspend again or transition

The key shift is this: a graph gives you **dependency order** inside one execution, while a `session` gives you **conversation order** across many executions.



14.7 First Working Example

The companion `customerServiceSession` example is the smallest realistic shape to copy:

```
session customerServiceSession {
  idle_timeout = 5m
  timeout_action = "cs_session_timeout_policy"
  max_rounds = 20
  max_history = 50

  phase greeting {
    node greet : CsSessionGreeter {
      input {
        sessionId = ctx.sessionId
      }
    }
    then -> triage
  }

  phase triage {
```

```
    max_rounds = 5
    yield_on = [respond]
    round {
      node respond : CsSessionResponder {
        input {
          userMessage = ctx.round.input.userMessage
        }
      }
    }
    until respond.output.done == true
    then {
      respond.output.action == "handoff" -> solve
      otherwise -> wrapUp
    }
  }
}
```

Read it in plain English:

1. Start the interaction with a one-time greeting phase
2. Enter a repeating triage phase
3. Each signal payload becomes the next `ctx.round.input`
4. Stay in triage until the responder says `done == true`
5. Hand off to a solver or wrap up immediately

That is the whole value proposition of `session`: the business interaction stays readable even though it spans multiple user turns.

14.8 Break It Apart

14.8.1 `ONCE` and `ROUND` phases solve different problems

Use an `ONCE` phase when the step has a single answer:

- greet the customer
- load the current account
- produce the final summary

Use a `ROUND` phase when the step can legitimately repeat:

- ask clarifying questions until the issue is understood
- review a document until the reviewer stops requesting revisions
- collect IVR menu input until the caller reaches a resolved path

The runtime behavior is different enough that the DSL makes the distinction explicit instead of hoping readers infer it from branches.

14.8.2 What a round can see

Inside a `ROUND` phase the runtime injects extra bindings on top of normal `GraphContext` data:

- `ctx.round.input` — the payload that resumed the current round
- `ctx.<phaseId>.output` — outputs from earlier completed phases
- `ctx.session.namespace` / `ctx.session.ownerId` — identity metadata when present

That means you usually combine three data sources in one round:

- initial request context such as `ctx.sessionId`
- accumulated prior phase outputs such as `ctx.collect.output.*`

- the newest user reply through `ctx.round.input.*`

14.8.3 Session controls and defaults

Property	Scope	Default	Why you care
`idle_timeout`	session	`30m`	Maximum idle time allowed between signals
`timeout_action`	session	none	Policy reference resolved by `TimeoutPolicyProvider`
`max_rounds`	session	`100`	Global cap across all round phases
`max_rounds`	`ROUND` phase	`1`	Cap for that specific phase — set this explicitly for real conversations
`max_history`	session	`0`	Retained history entries; `0` means **unlimited**, not disabled
`yield_on`	`ROUND` phase	terminal nodes	Which node outputs are returned after each round
`on_round_failure`	`ROUND` phase	`terminate_session`	How the runtime reacts when a round fails
`max_visits`	phase	unbounded	Hard cap on how many times the runtime may re-enter this phase via `then` rules
`then`	phase	none	Unconditional or rule-based transition to the next phase

Two defaults surprise people a lot:

- `max_history = 0` means **keep everything**
- a `ROUND` phase without its own `max_rounds` loops only **once**

> ***Preventing transition loops with `max_visits`***
> *`max_rounds` caps how many times a `ROUND` phase iterates internally. `max_visits` caps how many times the runtime is allowed to* enter *any phase via a `then` rule from elsewhere. Without it, two phases that route to each other can ping-pong indefinitely:*
>
> ```
> phase classify : ROUND { ... then "needs-more-info" -> gatherInfo }
> phase gatherInfo : ROUND { max_visits = 5  then "complete" -> classify }
> ```
>
> *When `max_visits` is exceeded, the runtime emits `PHASE_VISIT_LIMIT_EXCEEDED` and throws `SessionPhaseVisitExceededException`. The executor then marks the session failed through its normal error-termination path. Set `max_visits` on every phase that can be re-entered.*

14.8.4 Starting, signalling, and restoring in-memory state

The lifecycle API is intentionally small:

```
SessionExecutor executor = SessionExecutor.builder(engine)
    .accessGuard(new OwnerOnlySessionAccessGuard())   // default
    .build();

SessionHandle handle = executor.start(
        sessionGraph,
        new GraphContext(Map.of("sessionId", "SESSION-1001")),
        SessionIdentity.of("support", "user-123")
);

executor.signal(handle.sessionId(), Map.of("userMessage", "I need a refund"), "user-123");
```

`SessionExecutor` deliberately owns only the live, in-memory interaction. Its builder configures access control, listeners, timeout policies, and snapshot callbacks; it does not accept a durable store.

If an application has saved a session snapshot itself, it reconstructs the plain `SessionState` and context, then supplies both explicitly:

```
SessionHandle restored = executor.restore(
        sessionGraph,
        restoredState,
        restoredContext
);
```

That method resumes caller-provided state; it does not discover checkpoints, claim a lease, or recover sessions after a restart. Those responsibilities belong to `DurableSessionManager`, introduced later in this chapter.

14.8.5 Ownership is part of the model

A session is meant to live long enough that **who is allowed to signal it** becomes important.

By default the runtime uses `OwnerOnlySessionAccessGuard`, so the caller who started the session matters on later `signal(...)`, read, and terminate operations. That is a big difference between:

- "a graph that happens to wait once" and
- "a business interaction that might stay alive for hours or days"

The application must derive that caller id from its own authenticated request and supply it to `start(...)` and later operations. The similarly named Spring property below serves a different owner:

```
spring:
  bloge:
    session:
      owner-id: ${HOSTNAME}   # durable recovery/lease owner for this JVM
```

`spring.bloge.session.owner-id` is not a user-expression resolver. It names the JVM that claims durable recovery work; when omitted, the durable manager creates a random UUID. Authentication-to-`SessionIdentity` mapping remains an application boundary.

14.8.6 When *not* to use sessions

Do **not** reach for `session` automatically.

A plain graph is still the better tool when:

- there is only one suspend/resume boundary
- there are no meaningful phase names beyond node names
- the workflow is best described as a single DAG plus durability

And if the main challenge is **named lifecycle states with event-driven transitions**, not conversation rounds, Chapter 15 is usually the better fit.

14.9 Common Trap

14.9.1 [X] Forgetting that a `ROUND` phase defaults to one round

This looks like it should keep looping:

```
phase triage {
  round {
    node respond : CsSessionResponder { }
  }
  until respond.output.done == true
}
```

But it will stop after the first round unless you also configure the phase-level cap:

```
phase triage {
  max_rounds = 5
  round {
    node respond : CsSessionResponder { }
  }
  until respond.output.done == true
}
```

The session-level `max_rounds = 20` is a **global budget**. It does not override the phase-level default of `1`.

14.10 What Goes Wrong

14.10.1 A restarted in-memory executor cannot find the old session

After a service restart, creating a fresh `SessionExecutor` and signalling an old id fails because that executor has no live state for the session:

```
SessionExecutor executor = SessionExecutor.builder(engine).build();
executor.signal(sessionId, payload, "user-123"); // session is not active here
```

For caller-managed persistence, reconstruct `SessionState` and call `restore(sessionGraph, state, context)`. For store-backed restart recovery, use `DurableSessionManager` with an execution store, checkpoint store, and `definitionLookup`. Keeping these two paths separate prevents an in-memory executor from pretending that it owns distributed recovery.

14.11 Guided Rewrite

Open `interactive-review-session.bloge` and compare it with the single-wait patterns from Chapter 12.

The rewrite strategy is:

1. **Keep the one-time setup as an `ONCE` phase.** `collect` gathers the initial review context once.
2. **Move the repeated human step into a `ROUND` phase.** `review` executes once per reviewer response.
3. **Use `until` to express the exit rule directly.** The loop ends when the decision is no longer `needs_revision`.
4. **Finish in a final `ONCE` phase.** `finalize` writes the summary exactly once.

```
session interactiveReview {
  phase collect { ... then -> review }

  phase review {
    max_rounds = 3
    yield_on = [reviewRound]
    round {
      node reviewRound : DecideReviewOperator { ... }
    }
    until reviewRound.output.decision != "needs_revision"
    then -> finalize
  }

  phase finalize { ... }
}
```

The point of the rewrite is not “use more runtime features”.

14.12 Brain Check

1. When should you stay with a plain graph instead of introducing `session`?
   (When there is only one meaningful suspend/resume boundary and no need for multi-phase conversation structure.)
2. What is the difference between session-level `max_rounds` and phase-level `max_rounds`?
   (The session-level value is a global cap across all round phases; the phase-level value limits that one `ROUND` phase, and defaults to `1`.)
3. What does `ctx.round.input` contain?
   (The payload that resumed the current round.)
4. What does `max_history = 0` mean?
   (Unlimited retained history, not "disable history".)
5. What must the caller supply to `SessionExecutor.restore(...)`?
   (The `SessionGraph`, reconstructed `SessionState`, and corresponding `GraphContext`; persistence and lease recovery remain outside the pure executor.)
6. What makes `session` different from a graph with one `await` node?
   (A session is explicitly multi-turn: it owns phase order, round boundaries, session-level timeout/history, access control, and restore semantics across multiple signals.)

14.13 Lab

Goal: turn a one-shot support graph into a real multi-turn support session.

1. Start from a three-step interaction: greet, clarify, resolve.
2. Move `greet` and `resolve` into `ONCE` phases.
3. Put the clarification step in a `ROUND` phase.
4. Configure:
   - session `idle_timeout = 10m`
   - session `max_rounds = 12`
   - phase `max_rounds = 4`
   - `yield_on` so the caller can display the latest bot reply after every round
5. Add a branch that hands off to a specialist when the round output marks `action == "handoff"`.

Stretch: implement a custom timeout policy reference so an idle session sends a reminder before it terminates.

14.14 Durable Sessions

Everything above uses the pure in-memory `SessionExecutor`. In production, sessions that span hours or days need checkpoint persistence so they survive process restarts. The `bloge-session-durable` module provides `DurableSessionManager` for exactly this purpose.

14.14.1 Maven dependency

```
<dependency>
    <groupId>com.leanowtech.bloge</groupId>
    <artifactId>bloge-session-durable</artifactId>
</dependency>
```

14.14.2 Builder and usage

```
Map<String, SessionGraph> sessionRegistry = Map.of(
    sessionGraph.name(), sessionGraph
);
```

```
DurableSessionManager durableSession = DurableSessionManager.builder()
    .executionStore(executionStore)
    .checkpointStore(checkpointStore)
    .graphEngine(engine)
    .definitionLookup(sessionRegistry::get)
    .recoveryConfig(SessionRecoveryConfig.defaultConfig())
    .build();

durableSession.startup();
// If this session is not active locally, signal() claims its durable execution,
// restores the checkpoint, and then delivers the next turn.
durableSession.signal(sessionId, newTurn);
```

`DurableSessionManager` builds a `SessionExecutor` with a durable snapshot callback. Session snapshots are written through the checkpoint store. When `signal(...)` cannot find the session locally, the manager claims its execution, reloads the latest checkpoint, and then delivers the signal.

14.14.3 Content hash and version detection

Each session graph definition has a `contentHash` — a SHA-256 digest of the session graph structure. This hash is stored inside every checkpoint. When `DurableSessionManager` loads a checkpoint, it compares the stored hash with the current definition's hash.

If the hashes don't match, the session definition has changed since the checkpoint was written.

14.14.4 SessionVersionMismatchException

When a hash mismatch is detected, `DurableSessionManager` throws `SessionVersionMismatchException`:

```
try {
    durableSession.signal(sessionId, userTurn);
} catch (SessionVersionMismatchException e) {
    // e.sessionId()   — which session failed
    // e.checkpointHash() — hash from the checkpoint
    // e.currentHash() — hash of the current definition
}
```

Recovery strategies:

Strategy	When to use
Fail fast (default)	During development — forces you to notice definition changes immediately
Discard and restart	When old session state is expendable — catch the exception, start a fresh session
Migrate	When you need continuity — implement application-level migration that maps old phase state to the new definition

The choice depends on your domain. Short-lived support sessions can usually discard safely. Long-running onboarding flows may need migration logic.

14.15 Experiment acceptance card

- **Expected and observed:** Four turns map to context, round input, yield, and transition.
- **Failure and recovery:** Resume without session identity; recover the same identity and checkpoint.
- **Proof boundary:** Proves Session lifecycle behavior, not conversation correctness.
- **Exercise contract:** Four-turn transcript; change one input; deliver a phase table; stop when each owner and next transition are explainable.

14.16 Recap

- A plain graph models **one execution**; a `session` models a **multi-turn interaction**.
- `ONCE` phases handle single-pass work; `ROUND` phases handle repeated signal-driven work.
- `ctx.round.input` is the bridge between an external signal and the graph executed inside the current round.
- Session-level and phase-level limits are different; you usually need both.
- The pure executor restores caller-provided state; store-backed restart recovery belongs to `DurableSessionManager`.
- Use `session` when conversation structure matters more than a single DAG.

14.17 Next Step

In Chapter 15 — State Machines you will switch from conversational turns to **named lifecycle states**. Instead of repeating rounds until a phase is done, you will model explicit state transitions such as `draft -> pendingReview -> processing -> completed`.

14.18 Reference Links

- `bloge-session-ext/README.md` — in-memory session runtime and restore boundary
- `docs/session-phase-round-specification.md` — full session / phase / round contract
- `CustomerServiceSessionExample.java` — Java builder example
- `customer-service-session.bloge` — chapter-facing DSL example
- `interactive-review-session.bloge` — guided rewrite example

CHAPTER
15

Arc 4 — Lifecycle Models

Chapter 15 — State Machines

Sessions, state machines, nested ownership, and bounded agents

Goal: choose one explicit owner for every long-lived decision.

Promise: *By the end of this chapter you will know when a workflow has outgrown a single DAG and needs a `state_machine`, how to model event-driven transitions and timeouts, and how to keep explicit lifecycle state visible instead of hiding it in branches and flags.*

15.1 Learning Goals

1. Recognize when a problem is better modeled as named states and transitions instead of one branching graph.
2. Define `initial`, normal, and `terminal` states with local graphs that do the work inside each state.
3. Use event-driven, guarded, automatic, and timeout transitions without mixing their responsibilities.
4. Apply safety caps such as `max_transitions` and `max_state_visits` before a bad loop becomes a production incident.
5. Decide when a `state_machine` is the right abstraction — and when a graph or `session` is still simpler.

15.2 Prerequisites

- Chapter 5 — Branches That Decide — state transitions solve a different problem than graph branches
- Chapter 12 — Waiting for the World — event-driven transitions still depend on the same external signalling model
- Chapter 13 — Durable Execution — state-machine checkpoints and timers still ride on durable foundations
- Chapter 14 — Multi-Turn Sessions — useful for comparing "conversation turns" with "named lifecycle states"

15.3 Source Examples

File	What it shows
`ch14/order-lifecycle-state-machine.bloge`	Draft → review → processing → completed lifecycle with timeout fallback
`ch14/ticket-state-machine.bloge`	A broader support-ticket lifecycle with assign / escalate / resolve / close paths

File	What it shows
`OrderLifecycleStateMachineExample.java`	Fluent Java builder version using `StateMachineBuilder`
`bloge-state-ext/README.md`	Runtime model, timeout semantics, and nested-session note
`ReviewStateMachineWithSessionExample.java`	A preview of nested composition, expanded in Chapter 16

15.4 Why This Matters

A graph is excellent at answering one question: **what can run now, given the current dependencies?**

A state machine answers a different question: **what state is this business object in, and what event moves it to the next state?**

That distinction matters when the workflow revisits earlier states or must remain legible to humans outside the implementation team.

Examples:

- an order can move from `draft` to `pendingReview`, back to `draft`, then forward again
- a ticket can move from `open` to `triaging`, then either `assigned` or `escalated`
- a paused review state can time out and return to an earlier state automatically

Branches, flags, and loops can encode those rules inside one large graph. The lifecycle then remains implicit in node names and condition expressions. A `state_machine` represents named states and transitions directly.

15.5 Mental Model

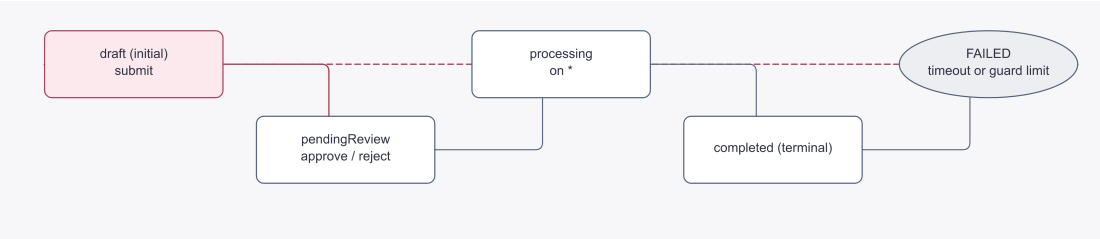
A state machine is an **event-driven shell** around per-state work.

Piece	Responsibility
`StateMachineDef`	Immutable definition of the whole machine
`state`	Named lifecycle step such as `draft`, `review`, `processing`
embedded `graph`	The work that runs while the machine is in that state
transition	Rule that moves to another state on event, timeout, or auto condition
`StateMachineExecutor`	Starts the machine and delivers external events
`StateMachineCheckpoint`	Serializable snapshot for restore and durable storage

Four transition styles matter most:

Transition style	Syntax	Use when
Event-driven	`on approve -> processing`	an external event should move the machine
Guarded	`on * when ... -> approved`	a state's output decides the next state
Automatic	`on * -> completed`	the machine should move immediately after state work finishes
Timeout-driven	`timeout = 24h + on_timeout -> draft`	the machine should fail over or rewind if nobody signals it

The machine's job is to make lifecycle state visible. The embedded graph's job is to do local work inside each state.



15.6 First Working Example

The `orderLifecycle` example is the cleanest first state machine to study:

The 44-line definition is kept whole because the reader must see every state, event, timeout, and terminal transition as one lifecycle; isolated state fragments would recreate the ownership ambiguity this chapter removes.

```
state_machine orderLifecycle {
  max_transitions = 25
  max_state_visits = 5
  timeout = 72h

  state draft [initial] {
    graph {
      node initOrder : InitOrderOperator {
        input {
          orderId = ctx.orderId
          customerId = ctx.customerId
        }
      }
    }
    on submit -> pendingReview
  }

  state pendingReview {
    graph {
      node reviewOrder : ReviewOrderOperator {
        input {
          orderId = ctx.draft.output.initOrder.orderId
        }
      }
    }
    on approve -> processing
    on reject -> draft
    timeout = 24h
    on_timeout -> draft
  }

  state processing {
    graph {
      node fulfillOrder : FulfillmentOperator {
        input {
          orderId = ctx.pendingReview.output.reviewOrder.orderId
        }
      }
    }
    on * -> completed
  }
```

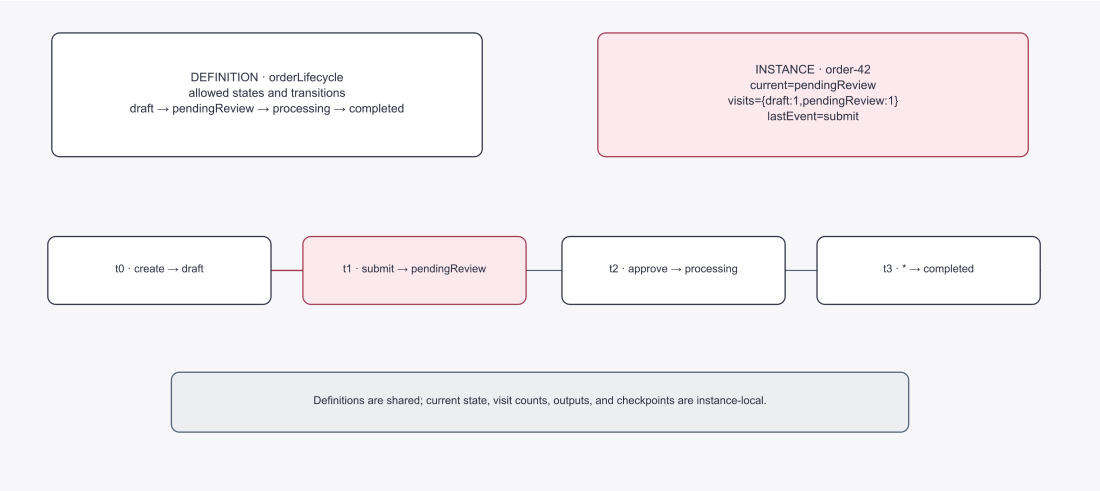
```
  state completed [terminal] { }
}
```

Read it as a lifecycle, not as a DAG:

- start in `draft`
- wait for `submit`
- move to `pendingReview`
- either `approve`, `reject`, or time out back to `draft`
- after `processing` completes, auto-transition to `completed`

15.7 A Shared Definition Is Not a Running Order

`orderLifecycle` defines what every order *may* do. `order-42` is one instance currently in `pendingReview`; another instance can be in `processing` at the same time. Mixing those two ideas is the fastest route to leaked state.



15.7.1 Read one instance as a sequence of accepted events

Time	Event	Instance fact after acceptance
`t0`	create	`currentState=draft`, visits for draft = 1
`t1`	`submit`	`currentState=pendingReview`, review graph output retained
`t2`	`approve`	`currentState=processing`, transition count = 2
`t3`	wildcard completion	`currentState=completed`, terminal = true

The definition contains state ids, graphs, guards, timeouts, and allowed transitions. The instance contains current state, visit/transition counters, state-local outputs, accepted event history, and checkpoint identity. The definition can be cached and shared; the instance must be isolated per order.

15.7.2 Change the definition, then ask what happens to the instance

Adding `manualReview` to the definition does not move `order-42` there. An event must select an allowed transition, and durable restore must decide whether the old checkpoint is compatible with the new definition. This is why definition versioning is a migration decision, not a live-state assignment.

15.8 Break It Apart

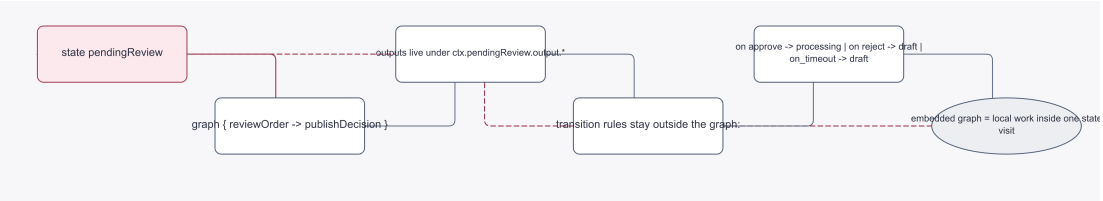
15.8.1 States hold local work; transitions hold lifecycle rules

Keep these responsibilities separate:

- the embedded `graph { ... }` describes what happens **inside** a state
- the `on ... -> target` rules describe **when the machine changes state**

When those responsibilities blur together, the definition becomes hard to reason about. A useful rule of thumb is:

- if the logic is about data dependency inside one visit, keep it in the state's graph
- if the logic is about changing lifecycle step, keep it in the transition list



15.8.2 Event-driven, automatic, guarded, and timeout transitions

`state_machine` supports more than one way to move forward:

- **Event-driven:** `on approve -> processing`
- **Automatic:** `on * -> completed`
- **Guarded:** `on * when ctx.review.output... -> approved`
- **Timeout-driven:** `timeout = 24h` and `on_timeout -> draft`

The timeout path is implemented as a synthetic timeout event behind the scenes. When a dedicated `TimerService` is present, paused instances can expire asynchronously; without one, the wall-clock limit is still enforced on the next `execute(...)` or `signal(...)` step.

15.8.3 Output shape is namespaced by state

State outputs are stored under the state ID, not flattened globally.

That is why the order example reads:

```
orderId = ctx.draft.output.initOrder.orderId
```

not simply:

```
orderId = ctx.initOrder.output.orderId
```

Namespacing keeps the lifecycle visible in downstream expressions. You can tell immediately which state produced the data.

15.8.4 Safety caps are part of the design, not an afterthought

The runtime ships with explicit runaway guards:

Guard	Default	Purpose
`max_transitions`	`100`	Caps total state transitions
`max_state_visits`	`10`	Caps visits to any single state
top-level `timeout`	none	Caps wall-clock lifetime of the entire machine
state `timeout`	none	Caps how long a waiting state can remain paused

These are especially important for backward transitions like `reject -> draft`. Without the caps, a bad event stream

or a bad guard can spin forever.

15.8.5 When a graph is still the better tool

Stay with a graph when:

- the work is a single pass through dependencies
- revisiting earlier named states is not part of the model
- you do not need external events to move between lifecycle stages

Stay with `session` when the core problem is a **conversation across rounds**. Reach for `state_machine` when the core problem is an **explicit lifecycle across states**.

15.9 Global Transitions

So far, every transition in the examples is defined inside a specific state. But some events should be handled **from any state** — cancellation, error recovery, or global timeouts. Repeating the same `on "CANCEL" -> cancelled` inside every non-terminal state is noisy and error-prone.

The `global_transitions` block solves this:

```
state_machine orderLifecycle {

  global_transitions {
    on "CANCEL"  -> cancelled
    on "ERROR"   -> error_state
    on "TIMEOUT" -> timed_out
  }

  initial_state = pending

  state pending {
    on "PAYMENT_RECEIVED" -> processing
    on "EXPIRED"          -> expired
  }

  state processing {
    on "FULFILLED"    -> completed
    on "OUT_OF_STOCK" -> backordered
  }

  state completed { terminal = true }
  state cancelled { terminal = true }
  state error_state { terminal = true }
  state timed_out { terminal = true }
  state expired { terminal = true }
  state backordered {
    on "RESTOCKED" -> processing
  }
}
```

15.9.1 How global transitions work

Global transitions are applied to **every non-terminal state** in the machine. They behave exactly like state-level transitions, but you write them once.

If a state defines its own transition for the same event, the **state-level transition wins**. This lets you override global behaviour when a specific state needs different handling.

15.9.2 When to use global vs state-specific transitions

Use global transitions for	Use state-specific transitions for
Cancellation events that should work from any state	Events that only make sense in a specific lifecycle stage
Error fallback handlers	Guarded transitions based on state-local output
Global timeout escalation	Automatic transitions (`on *`)

The rule of thumb: if removing the transition from one state would be a bug, it belongs in `global_transitions`.

15.10 Common Trap

15.10.1 [X] Using a state machine for a straight DAG

If your whole process is really just:

```
validate -> price -> confirm
```

then wrapping it in states like `validating`, `pricing`, and `confirming` does not make the workflow clearer. It just adds ceremony.

Use a state machine when the state names communicate something the DAG alone does not — especially when external events, retries across states, or backward transitions matter.

15.11 What Goes Wrong

15.11.1 A backward transition burns through `max_state_visits`

Suppose `pendingReview` can reject back to `draft`, and a bad integration keeps publishing `reject` forever.

That is exactly why `max_state_visits` exists. The runtime stops the machine instead of looping silently forever.

The fix is not "raise the cap until the test passes". The fix is to ask why the machine is cycling:

- is the transition guard wrong?
- is the external event wrong?
- is the domain lifecycle missing an intermediate state?

Test the loop intentionally before shipping it.

15.12 Guided Rewrite

Open `ticket-state-machine.bloge`.

Imagine you had started with one big graph plus branch nodes. The refactor into a state machine follows this sequence:

1. **Name the stable lifecycle states.** `open`, `triaging`, `assigned`, `escalated`, `resolved`, `closed`.
2. **Move local work into state graphs.** Receiving the ticket belongs inside `open`; classification belongs inside `triaging`.
3. **Move event boundaries into transitions.** `assign`, `escalate`, `resolve`, and `close` are lifecycle events, not graph branches.
4. **Add timeout behavior where silence has meaning.** `resolved` times out to `closed` after the waiting window.

The refactor pays off when someone new joins the project. They can read the lifecycle from the state names alone, before they care about the operator details.

15.13 Brain Check

1. What problem does `state_machine` solve better than a single branching graph?
   (Explicit lifecycle modeling across named states and external transitions.)
2. What does `on * -> completed` mean?
   (An automatic transition: after the state's graph finishes, move immediately to `completed`.)
3. Why does the order example read from `ctx.draft.output.initOrder.orderId`?
   (Because outputs are namespaced by state ID, making lifecycle provenance explicit.)
4. What is the difference between the machine-level `timeout` and a state-level `timeout`?
   (The machine-level timeout caps the whole run; the state-level timeout caps how long one waiting state can remain paused.)
5. When should you avoid introducing a state machine?
   (When the process is really just a one-pass DAG with no meaningful lifecycle states or external state transitions.)
6. Why is `max_state_visits` important for transitions like `reject -> draft`?
   (Because backward transitions can loop forever if bad events or bad guards keep revisiting the same state.)

15.14 Lab

Goal: model a real approval lifecycle as a state machine.

1. Define states `draft`, `review`, `approved`, and `rejected`.
2. Put the data-loading work in `draft`.
3. Put review enrichment in `review`.
4. Add transitions:
   - `on submit -> review`
   - `on approve -> approved`
   - `on reject -> draft`
5. Add `max_transitions` and `max_state_visits` explicitly.
6. If review can stall, add `timeout` and `on_timeout` on the review state.

Stretch: add one guarded `on * when ... -> approved` transition so the state's output can auto-complete the lifecycle without a separate signal.

15.15 Durable State Machines

In-memory state machines lose their current state when the process restarts. The `bloge-state-durable` module provides `DurableStateMachineManager` to persist every transition as a checkpoint.

15.15.1 Maven dependency

```
<dependency>
    <groupId>com.leanowtech.bloge</groupId>
    <artifactId>bloge-state-durable</artifactId>
</dependency>
```

15.15.2 Builder and usage

```
Map<String, StateMachineDef> definitions = Map.of(
    stateMachineDef.name(), stateMachineDef
);

DurableStateMachineManager durableSm = DurableStateMachineManager.builder()
```

```
    .executionStore(executionStore)
    .checkpointStore(checkpointStore)
    .graphEngine(graphEngine)
    .definitionLookup(definitions::get)
    .recoveryConfig(StateMachineRecoveryConfig.defaultConfig())
    .build();

durableSm.startup();
StateMachineResult waiting = durableSm.start(
    "orderLifecycle",
    Map.of("orderId", "ORD-123")
);
String executionId = waiting.instance().instanceId();

StateMachineResult result = durableSm.signal(
    executionId,
    "PAYMENT_RECEIVED",
    Map.of("providerRef", "PAY-9")
);
```

`start(...)` creates and claims a durable execution. When the machine reaches `WAITING_EVENT`, the manager persists a checkpoint and marks the execution suspended. `signal(...)` claims that execution, loads the checkpoint, resumes the definition selected by `definitionLookup`, and then delivers the event.

After `startup()`, the recovery loop scans expired claims and resumes from the last committed checkpoint. Work after that checkpoint may be retried, so external effects still need idempotency. Checkpoint/status atomicity additionally depends on wiring a transactional persistence coordinator; the direct default executes those writes independently.

This is the same checkpoint foundation introduced in Chapter 13 — Durable Execution, applied to the state-machine abstraction.

15.16 Experiment acceptance card

- **Expected and observed:** A shared definition does not share current state, visits, output, or checkpoint.
- **Failure and recovery:** Reuse one ID for two instances; recover unique identities and replay separately.
- **Proof boundary:** Proves definition/instance isolation, not trustworthy event sources.
- **Exercise contract:** Two ticket instances; change one instance ID; deliver side-by-side traces; stop when neither contaminates the other.

15.17 Recap

- A graph models dependency order; a `state_machine` models lifecycle state.
- Each state can run local graph work, but transitions are the main abstraction.
- Outputs are namespaced by state, which keeps provenance visible.
- Event, auto, guarded, and timeout transitions solve different problems.
- Global transitions handle events from any state — cancellation, errors, timeouts — without repetition.
- Safety caps such as `max_transitions` and `max_state_visits` are essential for backward transitions.
- `DurableStateMachineManager` persists transitions so machines survive restarts.
- Reach for `state_machine` when you need explicit states, not just more branches.

15.18 Next Step

In Chapter 16 — Composing Sessions and State Machines you will combine the two orchestration models. Some flows are conversations with an embedded lifecycle; others are state machines that need a short multi-step interaction inside one state.

15.19 Reference Links

- `bloge-state-ext/README.md` — state-machine runtime overview and timeout semantics
- `OrderLifecycleStateMachineExample.java` — Java builder example
- `order-lifecycle-state-machine.bloge` — chapter-facing lifecycle example
- `ticket-state-machine.bloge` — richer support-ticket lifecycle example
- `ReviewStateMachineWithSessionExample.java` — nested session preview

CHAPTER
16

Arc 4 — Lifecycle Models

Chapter 16 — Composing Sessions and State Machines

Sessions, state machines, nested ownership, and bounded agents

Goal: choose one explicit owner for every long-lived decision.

Promise: *By the end of this chapter you will know how to combine* `session` *and* `state_machine` *without losing track of signal ownership, timeout behavior, or output paths — and you will understand why the two nesting directions are useful but* ***not symmetric****.*

16.1 Learning Goals

1. Compare the two composition patterns: session-outermost and state-machine-outermost.
2. Decide which runtime owns lifecycle, signalling, recovery, and timeout semantics in a nested flow.
3. Trace output paths through nested orchestration without flattening away important structure.
4. Understand the current asymmetric constraints so you do not design a composition the runtime cannot support.
5. Choose the smallest composition boundary that keeps the business model obvious.

16.2 Prerequisites

- Chapter 14 — Multi-Turn Sessions — you need the session lifecycle first
- Chapter 15 — State Machines — you need explicit state transitions first
- Chapter 13 — Durable Execution — nested runtimes still sit on top of the same durability foundation

16.3 Source Examples

File	What it shows
`ch15/order-session-with-state-machine.bloge`	Session-outermost composition: a phase embeds a full order lifecycle state machine
`ch15/review-state-machine-with-session.bloge`	State-machine-outermost composition: a state embeds a short review session

File	What it shows
`ch15/ivr-customer-service.bloge`	A longer-lived session that could later grow nested lifecycle behavior
`OrderSessionWithStateMachineExample.java`	Java builder version of session → nested state machine
`ReviewStateMachineWithSessionExample.java`	Java builder version of state machine → nested session
`OrderSessionWithStateMachineExampleTest.java` / `ReviewStateMachineWithSessionExampleTest.java`	Runtime tests proving the nested shapes compile and execute

16.4 Why This Matters

Real business flows often need both abstractions:

- a customer conversation that contains a multi-step order lifecycle
- a lifecycle state that needs a short, self-contained review interaction before deciding where to go next

If you flatten everything into one graph, the model becomes muddy. If you split everything into separate services, the ownership boundaries become muddy.

Composition lets you keep the **business model** visible:

- the outer runtime says what the *main* lifecycle is
- the nested runtime handles one smaller orchestration problem inside it

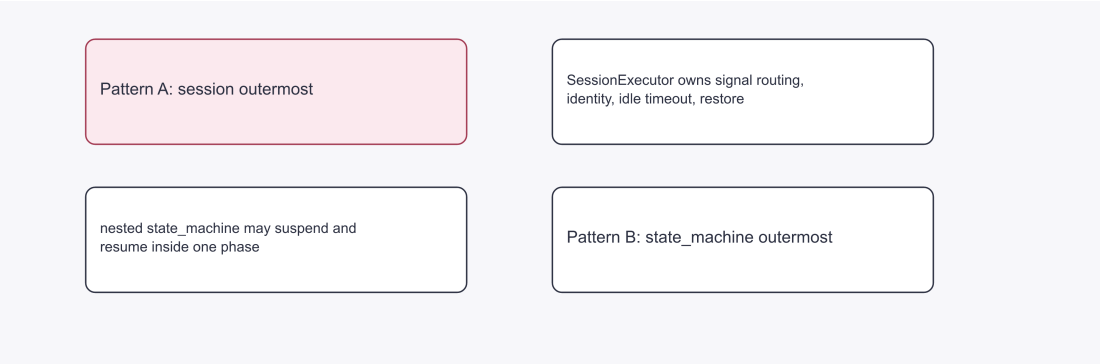
The crucial design question is not "can these be nested?" It is: **which runtime owns the lifecycle that humans or other systems care about most?**

16.5 Mental Model

There are two supported directions:

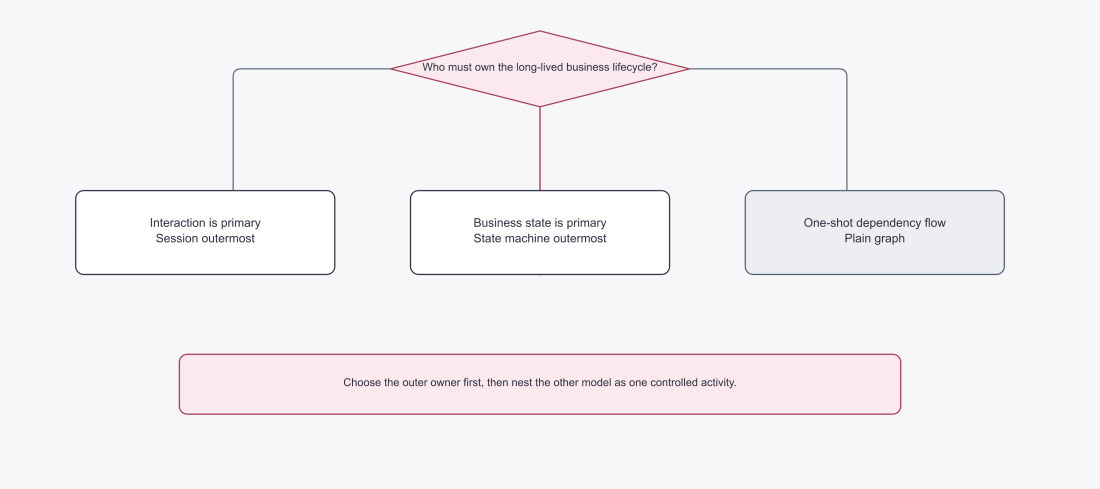
Pattern	Outer owner	Inner owner	Best fit
`session` → nested `state_machine`	`SessionExecutor`	`StateMachineExecutor` logic wrapped as a synthetic node	A long-lived interaction whose one phase needs explicit state transitions
`state_machine` → nested `session`	`StateMachineExecutor`	`SessionExecutor` logic wrapped as a synthetic node	A named lifecycle state that needs a short, self-contained multi-step interaction

The patterns are both useful, but their runtime constraints differ.



16.6 Choose the Outer Owner Before Choosing Syntax

Composition begins with one question: which runtime must still own the case after the inner activity ends?



Dominant fact	Outer owner	Inner activity example
A customer continues a multi-turn conversation	Session	Order state machine inside the ordering phase
An order continues through durable business states	State machine	Review Session inside the review state
Work is one-shot and dependency-driven	Graph	No second lifecycle model needed

For `orderWorkflow`, the caller returns to the conversation after order state reaches `shipped`, so Session is outermost. For `reviewWorkflow`, the case must become `approved` or `rejected` after interactive review, so the state machine is outermost. Nesting direction is an ownership decision, not formatting.

16.7 First Working Example

The more flexible direction is **session outermost, state machine nested inside one phase**:

```
session orderWorkflow {
  idle_timeout = 72h
  max_rounds = 10

  phase ordering {
    state_machine orderFlow {
      max_transitions = 20
      max_state_visits = 5

      state draft [initial] {
        graph {
          node collectInfo : CollectOrderInfoOperator { }
        }
        on submit -> pendingPayment
      }

      state pendingPayment {
        graph {
          node chargePayment : ChargePaymentOperator { }
```

```
        }
        on payment_confirmed -> processing
        on payment_failed -> draft
      }

      state processing {
        graph {
          node shipOrder : ShipOrderOperator { }
        }
        on * -> shipped
      }

      state shipped [terminal] { }
    }
    then -> fulfillment
  }
}
```

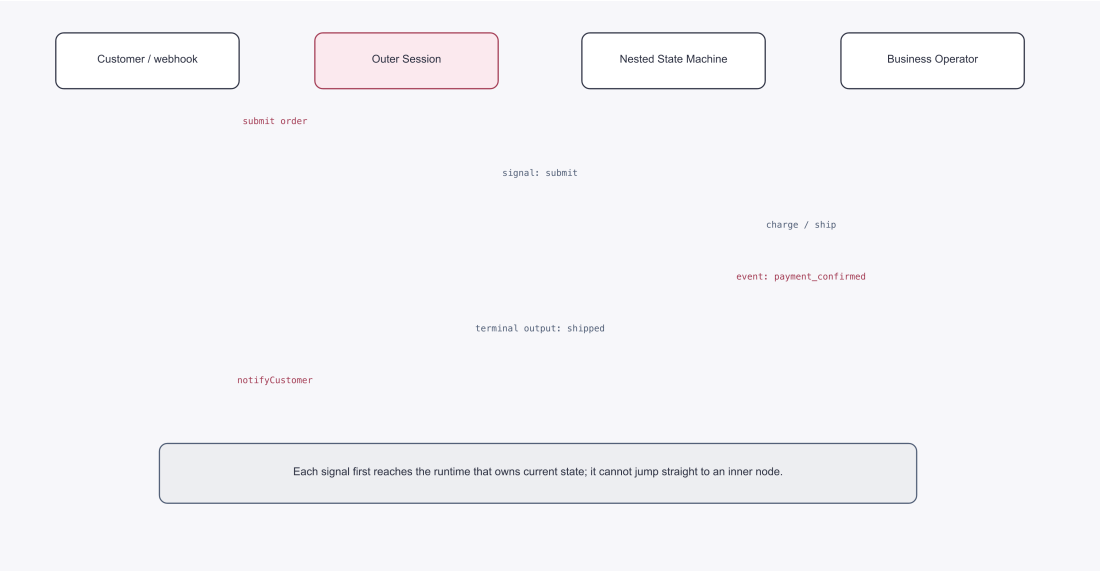
Read it as:

- the outer business interaction is an order session
- one phase of that interaction is a nested order lifecycle
- when the nested machine finishes, the session moves to fulfillment

The outer session still owns persistence, signalling, and the overall interaction identity.

16.8 Signals Travel Inward; Terminal Outputs Travel Outward

The nested runtime owns its current state, so an external event cannot jump directly to `chargePayment` or `shipOrder`. It enters through the outer Session, is translated to a state-machine signal, and only then selects a transition.



16.8.1 Trace one order through both owners

Direction	Contract crossing the boundary	Wrong shortcut
Inward	customer `submit` → Session input → state-machine `submit` signal	Address an inner node id from the webhook
Inward	payment callback → owning runtime → `payment_confirmed` event	Mutate `currentState` directly
Outward	state-machine terminal output contains `shipped` and `shipmentId`	Read arbitrary inner workspace from the caller
Outward	Session `fulfillment` phase calls `notifyCustomer`	Let inner Operator send ungoverned conversation messages

This gives both directions a receipt: accepted signal and state transition on the way in; namespaced terminal output and Session result on the way out. When a signal is rejected, inspect the owner, instance id, and allowed transition before inspecting business node code.

16.9 Break It Apart

16.9.1 Pattern A — session outermost, state machine inside a phase

This is the shape used by `order-session-with-state-machine.bloge`.

Important behaviors:

- the enclosing `SessionExecutor` is still the runtime entry point
- external `signal(...)` calls go to the session handle, not directly to the nested state machine
- while the nested state machine is waiting for an event, the outer session surfaces as `SUSPENDED`
- when the nested machine resumes, the session returns to `ACTIVE`

The output path makes the ownership visible:

```
finalState = ctx.ordering.output.orderFlow.stateMachine.currentStateId
shipmentId = ctx.ordering.output.orderFlow.processing.output.shipOrder.shipmentId
```

That path is long on purpose. It tells you the data came from:

- session phase `ordering`
- nested state machine node `orderFlow`
- nested state `processing`
- nested graph node `shipOrder`

16.9.2 Pattern B — state machine outermost, session inside a state

This is the shape used by `review-state-machine-with-session.bloge`.

The review state contains a nested session:

```
state review [initial] {
  session reviewSession {
    phase collect { ... then -> finalize }
    phase finalize { ... }
  }
  on * when ctx.review.output.reviewSession.finalize.output.decideReview.decision == "approved" ->
    approved
  on * -> rejected
}
```

This is where the asymmetry appears.

A nested session inside a state machine must complete **synchronously**. If that inner session suspends for an external

signal, the `SessionOperator` fails fast instead of leaving the enclosing state machine half-paused.
So this pattern is good for:

- a short two-phase review
- a compact enrich → decide interaction
- an inner lifecycle that stays self-contained inside one outer state visit

It is **not** good for a long-lived human conversation with repeated external turns.

16.9.3 Current composition constraints

Today the nested forms are intentionally narrow:

- a phase may contain at most **one** top-level nested `state_machine`
- that nested `state_machine` must be the only executable member in the phase
- a state may embed a nested `session`, but that session must complete synchronously

These constraints keep lifecycle ownership unambiguous.

> ***Constraint — avoid signal deadlock***
> *A nested `session` inside a `state_machine` cannot wait for an* external *signal, only complete synchronously. The reason is ownership: the outer state-machine executor holds the lifecycle lock and will not release control to another signal recipient. If the nested session tries to suspend on a signal, the compiler raises `SM_NESTED_SESSION_AWAITS_SIGNAL` and refuses to publish the graph.*
> *The reverse — a nested `state_machine` inside a `session` phase — can suspend freely, because the outer session is the signal owner. Reach for **Pattern A** (session outermost) whenever you might need the inner lifecycle to pause for the world.*

16.9.4 Ownership checklist

Ask these questions before you nest anything:

Question	If the answer is "session"	If the answer is "state machine"
What is the main business interaction?	Human or multi-turn interaction	Object lifecycle or status progression
Who receives external signals?	The session	The state machine
Who owns idle timeout / access guard?	The session runtime	The state-machine runtime
What should a dashboard show first?	Current phase / round	Current state
Can the inner flow wait across external turns?	Yes, if the outer owner is the session	No, not for a nested session inside a state machine

16.9.5 Choose the outer owner first

The easiest way to design nested orchestration is to choose the outer owner first:

- if humans think in **conversation turns**, make `session` outermost
- if humans think in **named states**, make `state_machine` outermost

Only after that should you decide whether one inner portion deserves its own orchestration model.

16.10 Common Trap

16.10.1 [X] Assuming the two nesting directions are symmetric

They are not.

- A nested state machine inside a session can wait and resume through the outer session's lifecycle.

- A nested session inside a state machine must finish synchronously today.

If you design as though both inner runtimes can suspend freely, you will discover the constraint late — after you already modeled the business flow the wrong way.

16.11 What Goes Wrong

16.11.1 A signal is sent to the wrong runtime boundary

In the session-outermost pattern, the nested state machine may be the thing logically waiting — but the signal still has to enter through the session runtime.

So this is wrong:

```
// Wrong mental model: trying to signal the nested machine directly
stateMachineExecutor.signal(...)
```

and this is the correct shape:

```
sessionHandle.signal(Map.of("event", "payment_confirmed"));
```

Route the signal to the **outer owner**.

16.12 Guided Rewrite

Compare the two companion examples and practice choosing the outer owner before you write any DSL.

1. **Start with the business question.** Are you mainly tracking a conversation, or a lifecycle state?
2. **Choose the outer runtime.** Session for conversation, state machine for lifecycle.
3. **Nest only the portion that has a different shape.**
   - Order flow: the outer interaction is a customer/order session, but one phase is best expressed as states.
   - Review flow: the outer lifecycle is a review workflow, but one state needs a short collect → finalize interaction.
4. **Write one downstream expression that proves the output path is understandable.**

If the path expression is unreadable, your boundary is probably wrong.

16.13 Brain Check

1. In the session-outermost pattern, who receives external signals?
   (The outer `SessionExecutor` / `SessionHandle`.)
2. In the state-machine-outermost pattern, can the nested session suspend for a later external signal?
   (No. It must complete synchronously today.)
3. Why is `ctx.ordering.output.orderFlow.stateMachine.currentStateId` a good thing, not a bad thing?
   (Because it preserves ownership and provenance across the nested boundary.)
4. What should you decide first when designing a nested flow?
   (Which runtime owns the main business lifecycle.)
5. When is session-outermost usually the safer default?
   (When the flow is long-lived, signal-driven, and centered on a human or multi-turn interaction.)
6. What is the key risk of treating nested composition as symmetric?
   (You may design an inner session that needs to suspend, even though the state-machine-outermost pattern does not support that today.)

16.14 Lab

Goal: choose and implement the right outer owner for two similar-looking workflows.

1. Model a customer order flow where the customer may respond hours later to payment prompts.
2. Model an internal approval flow where the review interaction completes immediately from existing data.
3. For each workflow, write down:
   - outer runtime
   - inner runtime (if any)
   - who receives signals
   - one sample downstream output path
4. Implement one DSL file for each shape.

Stretch: add a test that proves you can signal the session-outermost example through the outer session handle and still reach the nested state-machine terminal state.

16.15 Experiment acceptance card

- **Expected and observed:** The outer owner controls lifecycle; signals move inward and terminal outputs outward.
- **Failure and recovery:** Make Session and state machine co-own the outside; recover one owner.
- **Proof boundary:** Proves composition ownership and signal direction, not deadlock-free arbitrary nesting.
- **Exercise contract:** Support composition; swap only the outer owner; deliver a choice card and sequence; stop with one owner.

16.16 Coding practice

16.16.1 BP-12 — Reuse a contract with one lifecycle owner

Extract a subgraph around stable business input/output, not visual similarity. Callers should not depend on internal node IDs; choose one authoritative outer lifecycle owner.

Try one change: Replace a credit-assessment internal node without changing callers. For a Session/state-machine composition, name the owner of identity, signals, and recovery.

Reasoning, counterexamples, and limits

16.17 Recap

- You can compose `session` and `state_machine`, but the two directions have different constraints.
- The outer runtime owns signalling, recovery, and the business identity that matters most.
- Session-outermost is the right fit for long-lived, externally driven interactions with an inner lifecycle.
- State-machine-outermost is the right fit for named lifecycle states that contain a short synchronous inner interaction.
- Long output paths are a feature: they preserve provenance across nested boundaries.
- Choose the outer owner first, then justify whether any inner slice really needs its own orchestration model.

> ***Durable variants for production.*** *When composing sessions and state machines in production, wire the corresponding durable managers: use* `DurableSessionManager` *(from* `bloge-session-durable`*) for crash-safe session lifecycle, and* `DurableStateMachineManager` *(from* `bloge-state-durable`*) for crash-safe state-machine lifecycle. The composition patterns in this chapter remain the same — the durable wrappers add checkpoint persistence under the same API surface.*

16.18 Next Step

In Chapter 17 — Putting a Nondeterministic Agent Inside a Deterministic Lifecycle, the same ownership discipline contains a nondeterministic Agent inside explicit tool, turn, and lifecycle boundaries. Chapter 18 then returns to testing discipline.

16.19 Reference Links

- `order-session-with-state-machine.bloge` — session-outermost example
- `review-state-machine-with-session.bloge` — state-machine-outermost example
- `ivr-customer-service.bloge` — longer-lived session example
- `OrderSessionWithStateMachineExample.java` — Java builder implementation
- `ReviewStateMachineWithSessionExample.java` — Java builder implementation
- `OrderSessionWithStateMachineExampleTest.java` — nested state-machine runtime test
- `ReviewStateMachineWithSessionExampleTest.java` — nested session runtime test

CHAPTER
17

Arc 4 — Lifecycle Models

Chapter 17 — Putting a Nondeterministic Agent Inside a Deterministic Lifecycle

Sessions, state machines, nested ownership, and bounded agents

Goal: choose one explicit owner for every long-lived decision.

Promise: *You will let a support Agent use only allowlisted read tools, make turns, tool concurrency, and exit reasons observable, and prevent an incomplete answer from marking a ticket resolved through a deterministic gate.*

17.1 Learning goals

By the end of this chapter, you can:

1. Separate Session rounds, ticket-state transitions, and Agent model turns.
2. Treat the tool list as a capability allowlist rather than prompt advice.
3. Separate `finishReason`, loop exit reason, and business resolution.
4. Handle normal stop, tool failure, `max_turns`, and recoverable checkpoints correctly.
5. Place deterministic validation and human handoff after an Agent.

17.2 This chapter adds one variable

The support story already has a stable `ticketId`, a multi-round Session, and a ticket state machine. This chapter does not introduce a fourth lifecycle; it adds one nondeterministic model/tool loop inside `triaging`.

```
Session round: customer and system exchanges
Ticket state: open → triaging → assigned / escalated → resolved
Agent turn: model response → tool call → observation → next response
```

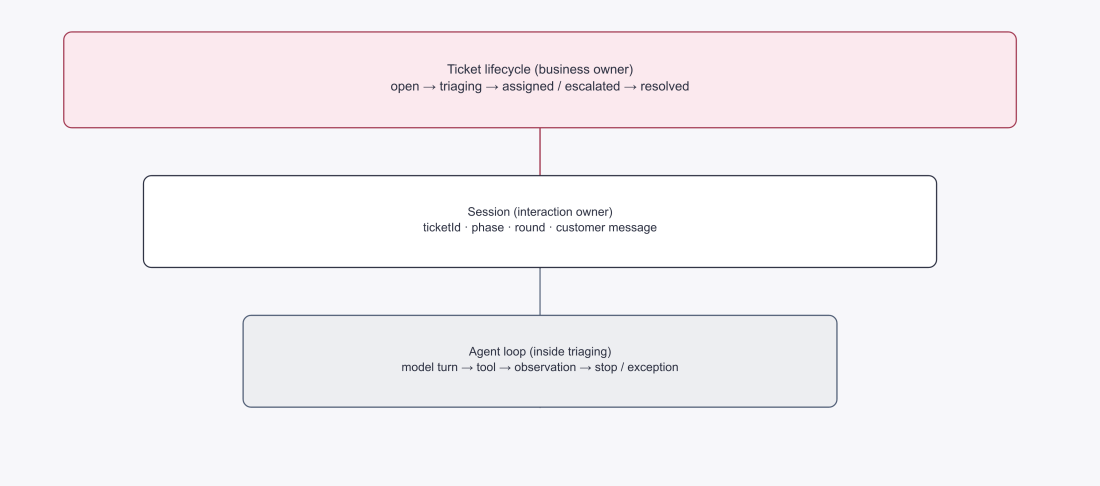
All three counts can change, but the word “turn” must not collapse them.

17.3 Opening problem: can “resolved” close the ticket

A customer says, “My last refund has not arrived.” After searching the knowledge base, the Agent replies, “Refunds usually take three to five business days; this is resolved.”

Should `ticket.status` become `resolved`?

No. Model text is a candidate explanation. The system still does not know whether this refund exists, exceeded SLA, or needs account review. No refund effect occurred.



17.4 Minimum support Agent: two read-only tools

```
agent supportTriage {
  model = "replaceable-support-model"
  system_prompt = "Clarify the issue. Never claim a refund was issued."
  max_turns = 4
  max_tool_concurrency = 2
  memory = sliding_window(6)

  tool searchKnowledgeBase : KBSearchOperator {
    description = "Search approved support articles"
    input { query = tool_args.query }
  }

  tool readTicket : ReadTicketOperator {
    description = "Read the current ticket snapshot"
    input { ticketId = ctx.ticketId }
  }

  exit_condition = finish_reason == "stop"
}
```

The two parameters have different authority. The model supplies `query`; the host injects `ticketId`, so the model cannot switch to someone else's ticket.

There is no `refund`, `closeTicket`, or `changeSla` tool. "Never refund" in a prompt is guidance. The absence of a refund capability from the tool list is the execution boundary.

17.5 Observe the actual RC1 loop semantics

Run compiler and loop tests:

```
cd submodule/bloge
mvn -pl bloge-agent-ext \
  -Dtest=AgentLoopOperatorTest,AgentDslCompileTest test
```

Observed on 2026-09-14 at commit `cc38fbe5`:

```
AgentLoopOperatorTest  Tests run: 6, Failures: 0
```

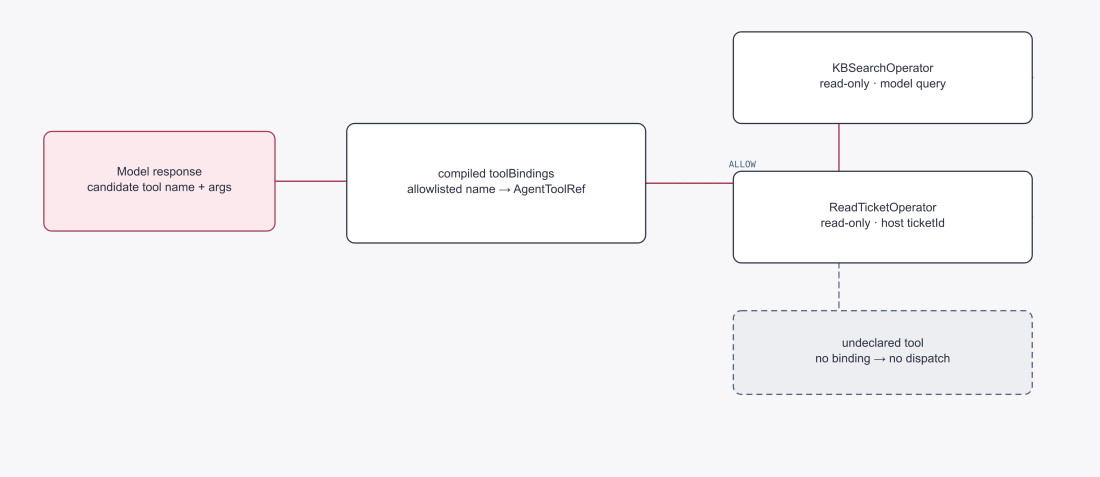
```
AgentDslCompileTest    Tests run: 5, Failures: 0
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0
```

The 11 tests cover direct `stop`, continue after a tool call, multi-tool concurrency, tool-failure feedback, `max_turns`, DSL schema inference, and embedded operators.

17.6 Mechanism: what happens in one turn

The synchronous `AgentLoopOperator` path is:

```
assemble memory
  → call llmChat
    → no tool call or finish_reason=stop: return AgentOutput
    → tool call: dispatch only through compiled toolBindings
      → append tool result to memory
        → evaluate exit_condition
          → next turn or exit
```



`ToolDispatcher` resolves a name to `AgentToolRef`, which points to a compiled subgraph and operator. An undeclared name has no dispatch binding. The host must still register the correct implementation and authority; an allowlist cannot repair an operator that is itself overprivileged.

17.7 Read four outcomes separately

Runtime outcome	RC1 behavior	Downstream handling
model `stop`, no tool call	returns `AgentOutput`	still pass the business gate
`exit_condition` true	returns `AgentOutput`; listener records exit	still pass the business gate
tool failure	error becomes a tool message; loop may continue	inspect degradation and handoff
`max_turns` reached	throws `MaxTurnsExceededException` with `AgentCheckpoint`	node fails; retain/handoff, never treat as final

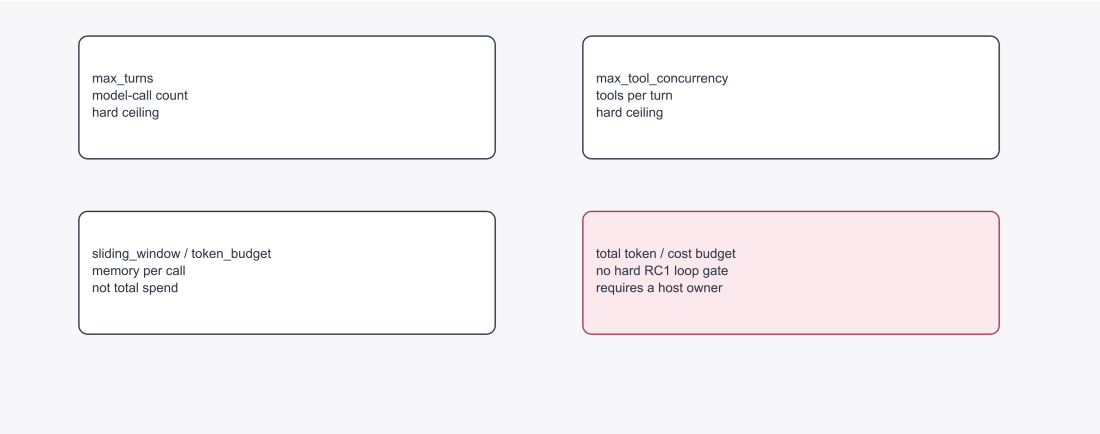
This corrects a common misconception: in RC1, exhausting `max_turns` **does not send partial text downstream as a normal `AgentOutput`**. Partial progress is in the exception's checkpoint: conversation, turn count, and recent tool results.

On a normal return, `AgentOutput.finishReason` is the model's last reason. `exit_condition` is the listener's loop-exit reason. Neither is business state `resolved`.

17.8 Budget: what is controlled and what is not

`max_turns` bounds model calls; `max_tool_concurrency` bounds tools per turn. `sliding_window(6)` limits history sent on each call, not total tokens or cost.

The RC1 loop reports accumulated tokens to listeners, but this source path has no hard "stop after total tokens" gate. If the business requires a spend limit, the host must enforce rejection in the provider, operator/interceptor, or outer control plane and map that rejection to an explicit state.

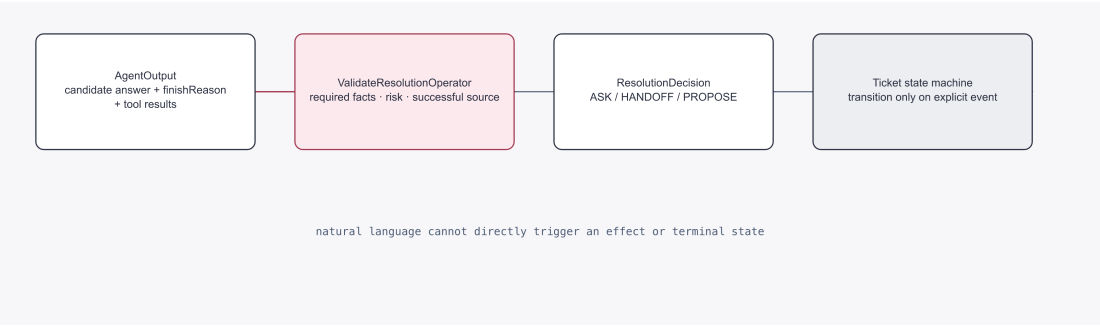


17.9 Deterministic downstream gate

After a normal Agent return, `ValidateResolutionOperator` reads structured facts:

```
ResolutionDecision validate(AgentOutput out, TicketSnapshot ticket) {
    if (!"stop".equalsIgnoreCase(out.finishReason())) return HANDOFF;
    if (out.content() == null || out.content().isBlank()) return HANDOFF;
    if (!ticket.requiredFactsComplete()) return ASK_CUSTOMER;
    if (ticket.refundOrSlaDispute()) return HUMAN_REVIEW;
    return PROPOSE_RESOLUTION;
}
```

The gate does not grade whether prose sounds human. It checks facts required before closure. `PROPOSE_RESOLUTION` remains a proposal; the state machine transitions to `resolved` only after an explicit event.



The stable responsibility chain is:

```
Agent produces a candidate answer
  → deterministic gate checks structured facts
    → state machine receives an explicit event
      → human or system owner owns the final state
```

17.10 Single-factor failure: change `max_turns` from 4 to 2

Keep model responses and tools fixed. Turn 1 asks for a knowledge lookup; turn 2 requests another tool and never returns `stop`. `AgentLoopOperatorTest.throwsWhenMaxTurnsAreExhausted` asserts:

```
exception = MaxTurnsExceededException
maxTurns = 2
checkpoint.turnCount = 2
```

Recovery is not showing the checkpoint's last sentence as an answer. Retain checkpoint and exit reason, leave the ticket in `triaging`, then let policy choose retry, customer clarification, or human handoff.

17.11 Another failure: a tool errors, but the loop sounds fluent

RC1 serializes a tool exception into a tool message so the model can explain or degrade. The test's tool throws `boom`; the next turn returns `Handled fallback.`

The loop returned successfully, but the business fact was not obtained. The deterministic gate must know whether a required fact came from a successful tool result. Fluent fallback cannot turn "refund record unavailable" into "refund is normal."

17.12 Real-world transfer: an emergency triage assistant

An emergency assistant may read registration data and triage guidance. It cannot discharge a patient because a model says "safe to go home." A rule gate checks vital signs and required exams; a clinician owns the final transition.

Support system	Emergency system
read-only ticket/KB tools	patient/guideline lookup
`max_turns`	interview-step limit
partial checkpoint	incomplete interview record
resolution gate	required-exam and risk gate
human handoff	clinician decision

The mapping is not simply "both use AI." Nondeterministic advice is enclosed by a deterministic responsibility boundary.

17.13 Your turn: draw an authority and exit card

Choose one candidate Agent in your system:

1. List at most three read-only tools and no default write tools.
2. Mark each parameter as model-supplied or host-injected.
3. Define `max_turns`, per-turn concurrency, and the owner of a true total budget.
4. Draw `stop`, tool failure, and max-turns paths.
5. Design a downstream gate that reads structured facts only.

Deliver an authority-boundary diagram and an exit matrix. Stop when no natural-language output can bypass the gate to trigger a high-risk effect or terminal state.

17.14 What this chapter proved—and did not prove

The RC1 tests prove that declared Agent DSL compiles into an embedded operator, allowlisted tools execute, concurrency limits hold, tool failures can return as observations, and exhausted turns throw an exception with checkpoint. They do not prove answer correctness, prompt-based authorization, a total token budget, or automatic ticket closure.

Field, Builder, memory, streaming, and listener catalogs remain in Appendix E. Chapter 18 — Testing Your Graphs

next establishes the test layers needed before Arc 6 separates business verdict, trust, and capability.

17.15 Coding practice

17.15.1 BP-13 — Define stop and resource boundaries

State input limits, business termination, capacity, and budgets separately. A method returning or cancellation being requested does not establish actual worker exit or resource release.

Try one change: Test empty input, the limit, over-limit input, and a condition that never becomes true. Distinguish an Agent budget exit from a valid business result and process-local recovery permits from cluster-wide limits.

Reasoning, counterexamples, and limits

17.16 Experiment acceptance card

- **Expected and observed:** A scoped Agent returns a finish reason, turn budget, and deterministic gate.
- **Failure and recovery:** Set max_turns to 2 or fail a tool; recover with an explicit partial result.
- **Proof boundary:** Proves controlled loop and permissions, not a correct or approved model answer.
- **Exercise contract:** One ticket; change only allowlist or budget; deliver an exit card; stop when every exit has a downstream action.

17.17 Exact fact entry points

- `AgentLoopOperator.java`
- `AgentLoopOperatorTest.java`
- `AgentDslCompileTest.java`
- `AgentOutput.java`
- `MaxTurnsExceededException.java`
- Customer-support Story Bible

Arc 4 Recap — Lifecycle Models (Chapters 14–17)

CHAPTER

—

Arc 4 Recap — Lifecycle Models (Chapters 14–17)

You can now move beyond one-shot graphs and model longer-lived orchestration in ways that keep lifecycle structure visible.

0.1 What You Learned

Chapter 14 — Multi-Turn Sessions promoted the old appendix-only session material into the main reading path. You learned how `SessionGraph`, `ONCE` phases, and `ROUND` phases model a conversation that spans multiple external signals; durability and restart recovery require the durable manager and stores.

Chapter 15 — State Machines introduced explicit lifecycle modeling. You learned how states, event-driven transitions, guards, and timeout rules make status progression visible instead of hiding it inside branch-heavy DAGs.

Chapter 16 — Composing Sessions and State Machines showed how to combine the two runtimes without losing signal ownership or lifecycle clarity. You learned that nested composition is useful but not symmetric, and that the outer runtime must own the main business lifecycle.

Chapter 17 — Putting a Nondeterministic Agent Inside a Deterministic Lifecycle added a bounded tool-use loop. You learned to constrain tools and turn count and to let an explicit graph—not the model— own exit and failure policy.

0.2 What You Should Be Able to Do Now

- Decide whether a workflow is best modeled as a graph, a `session`, a `state_machine`, or a deliberate composition of them
- Split a multi-turn conversation into `ONCE` and `ROUND` phases with explicit limits and timeout policy
- Model a named lifecycle with event-driven, automatic, guarded, and timeout transitions
- Explain which runtime owns signals, recovery, and identity in a nested flow
- Trace nested output paths without flattening away state and phase provenance

0.3 Common Mistakes at This Stage

Mistake	Why It Happens	Fix
Replacing every `await` with `session`	Sessions feel like the “more advanced” tool	Use `session` only when the interaction is truly multi-turn; one suspend/resume boundary still fits a graph

Mistake	Why It Happens	Fix
Treating state machines like fancy branch nodes	The workflow still "looks conditional" at first glance	Ask whether named states and backward transitions matter to the business model
Forgetting phase-level `max_rounds`	The session-level cap looks like it should be enough	Set explicit limits on every real `ROUND` phase
Ignoring ownership in nested composition	Both runtimes seem to offer suspend/resume concepts	Choose the outer owner first; all signals enter through that boundary
Assuming nested session and nested state-machine support are symmetric	The surface syntax looks similar	Remember the current constraint: nested sessions inside a state machine must complete synchronously

0.4 Key Vocabulary Additions

Term	Meaning
`SessionGraph`	Immutable definition of a multi-turn interaction made of named phases
`ROUND` phase	Phase that resumes per signal payload and can repeat until `until` matches
`SessionExecutor`	Runtime entry point for starting, signalling, restoring, and terminating sessions
`StateMachineDef`	Immutable lifecycle definition made of named states and transitions
Guarded transition	A transition that fires only when a predicate over state output is true
Lifecycle ownership	The rule that the outer runtime owns signals, recovery, and business identity in a nested orchestration flow
Bounded agent loop	A tool-use loop with explicit tools, turn limit, exit condition, and graph-owned failure behavior

0.5 Try It: Name the Lifecycle Owner

A support case has a 30-day lifecycle, a four-turn customer conversation, and an agent that may call two tools per turn. Draw three nested boxes and name the owner of case status, conversation round, and tool-loop exit. If two boxes claim the same transition, rewrite the boundary until each decision has one owner.

0.6 Where to Go Next

Arc 5 moves from lifecycle models to technical confidence: testing, observability, Spring wiring, and scale. Start with Chapter 18 — Testing Your Graphs.

Learning Arc

Arc 5 — Technical Confidence

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

CHAPTER
18

Arc 5 — Technical Confidence

Chapter 18 — Testing Your Graphs

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

Promise: *By the end of this chapter you will know how to verify a BLOGE graph's declared contract before it reaches production — by asserting on `GraphResult`, isolating operators with test doubles, executing whole graphs with `GraphTestRunner`, catching DSL mistakes at compile time, and controlling time-dependent behavior with deterministic test clocks.*

18.1 Learning Goals

1. Use `GraphResult` as the primary test surface: assert success, inspect typed outputs safely, and verify per-node status or timing when behavior changes.
2. Isolate operator behavior with `MockOperator` so tests stay focused on graph wiring instead of unrelated I/O.
3. Execute a graph in a purpose-built harness with `GraphTestRunner` and verify execution order, skips, and status transitions.
4. Compile DSL snippets with `DslTestHelper` so parse or compile failures stop before runtime, and choose the test level that matches the failure.
5. Control retries, delays, and timer-driven behavior deterministically with `ManualTimeSource`, `DeterministicTimerService`, and `TestGraphEngine`.

18.2 Prerequisites

- Chapter 2 — Your First Graph — you need to know how graphs are loaded and executed.
- Chapter 6 — Resilience by Design — retries, timeout, and fallback behavior influence what a successful assertion looks like.
- Chapter 7 — Designing Good Operators — you need the operator contract before you can isolate it in tests.

18.3 Source Examples

File	What it shows
`GraphResultObservabilityTest.java`	`GraphResult` execution identity, timings, and compatibility behavior
`GraphResultSafeAccessTest.java`	Safe typed output access patterns for tests
`MockOperator.java`	Test doubles that return, throw, delay, or record invocations
`GraphTestRunner.java`	Graph harness with ordering, status, and execution-log assertions
`DslTestHelper.java`	Parse/compile assertions for `.bloge` snippets
`TestGraphEngine.java`	Deterministic test harness with `ManualTimeSource` and timer control
`TestGraphEngineTest.java`	Minimal examples of advancing logical time instead of waiting on wall clock
`DeterministicRetryTest.java`	Retry/backoff assertions driven by manual time advancement

18.4 Why This Matters

A graph can look right in a code review and still be wrong in one of three costly ways:

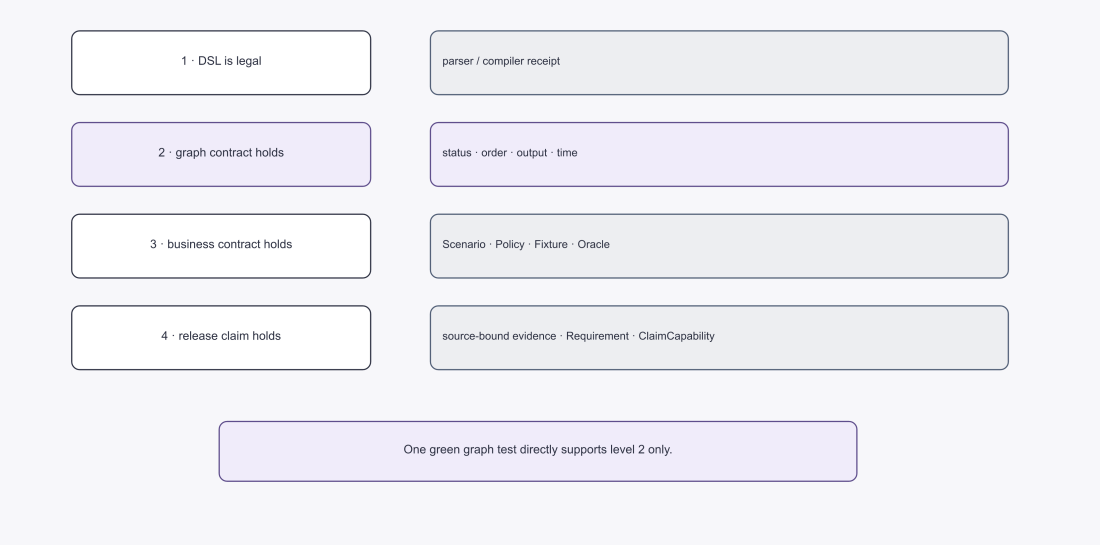
- a branch change skips a node that used to run,
- a fallback hides a failure you meant to surface, or
- a DSL refactor compiles in your head but not in the actual compiler.

Good BLOGE tests make those failures visible at the cheapest possible layer. You do not need to boot Spring or hit real services just to verify a dependency edge, a fallback path, or a typed output contract.

The testing mindset for BLOGE is simple: **assert the graph's contract, not an accidental implementation detail.** Start at `GraphResult`, then zoom in with `MockOperator`, `GraphTestRunner`, or `DslTestHelper` when you need more focus.

18.5 One Green Test, Four Different Claims

The order test can be completely green while a wider claim is still unknown. Read every test result together with the boundary it actually exercised:



Claim	Evidence needed	What this chapter contributes
The DSL is legal	Real parser/compiler result	`DslTestHelper`
The graph follows its declared contract	Status, order, output, time, and failure assertions	`GraphTestRunner` and `GraphResult`
The business story is approved and still satisfied	Owner-approved Scenario, Policy, Fixture, and Oracle	Not established here
The release claim is source-bound and governed	Sealed evidence, exact source identity, Requirement and ClaimCapability	Not established here

A passing `OrderGraphTest` therefore supports a precise sentence: "under these test inputs and doubles, the graph produced these observed node facts." It does not support "the order business is correct" or "this release is qualified."

18.6 Mental Model

Think in three concentric safety nets:



Each layer answers a different question:

- **DSL compile tests** ask: "Is this graph definition legal?"
- **Graph tests** ask: "Does the workflow satisfy the contract declared by this test?"
- **Operator unit tests** ask: "Does this operator transform inputs correctly?"

When a test fails, the layer tells you where to look next.

18.7 First Working Example

This example shows the most common production need: run a graph, assert it succeeded, inspect outputs, and verify per-node timings. Everything below is plain JUnit 5 with no external dependencies beyond `bloge-core`. The 55-line

class is kept whole so imports, graph construction, execution, and all six assertion surfaces remain a copyable test rather than disconnected fragments.

```
import com.leanowtech.bloge.core.dsl.GraphBuilder;
import com.leanowtech.bloge.core.engine.GraphEngine;
import com.leanowtech.bloge.core.engine.GraphResult;
import com.leanowtech.bloge.core.model.NodeStatus;
import com.leanowtech.bloge.core.operator.Operator;
import com.leanowtech.bloge.core.spi.DefaultOperatorRegistry;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class OrderGraphTest {

    @Test
    void orderGraph_completesSuccessfully() {
        Operator<Void, String>   validate = (in, ctx) -> "valid";
        Operator<String, String> price    = (in, ctx) -> "$42.00";
        Operator<String, String> confirm  = (in, ctx) -> "ORD-001";

        var gb = new GraphBuilder("order");
        var graph = gb
                .node("validate", validate)
                .node("price", price).dependsOn("validate")
                    .input((results, ctx) -> results.get("validate", String.class))
                .node("confirm", confirm).dependsOn("price")
                    .input((results, ctx) -> results.get("price", String.class))
                .build();

        var engine = GraphEngine.builder()
                .registry(new DefaultOperatorRegistry())
                .build();

        GraphResult result = engine.executeWithOperators(graph, null, gb.operators());

        // 1. Overall success
        result.requireSuccess();                        // throws if any error

        // 2. Typed output extraction
        assertEquals("ORD-001", result.getOutput("confirm", String.class));

        // 3. Safe access when a node might not exist
        assertTrue(result.findOutput("confirm", String.class).isPresent());
        assertTrue(result.findOutput("missing", String.class).isEmpty());

        // 4. Per-node status
        assertEquals(NodeStatus.COMPLETED, result.getStatus("validate"));
        assertTrue(result.nodeSucceeded("price"));

        // 5. Per-node timing (always non-null after execution)
        assertNotNull(result.nodeTimings().get("validate"));
        assertFalse(result.nodeTimings().get("validate").isZero());
```

```
        // 6. Execution identity
        assertNotNull(result.executionId());
    }
}
```

Source: `GraphResultObservabilityTest.java` and `GraphResultSafeAccessTest.java` demonstrate every access method shown above.

18.8 Break It Apart

18.8.1 GraphResult — your primary assertion target

`GraphResult` is an immutable record with nine fields. The accessors you will use most in tests:

Method	Returns	Use when
`isSuccess()`	`boolean`	Quick pass/fail check
`requireSuccess()`	`this` or throws `GraphExecutionException`	Fail-fast in tests — exception message lists all errors
`getOutput(nodeId, type)`	`T`	You are certain the node completed
`findOutput(nodeId, type)`	`Optional<T>`	The node may have been skipped or failed
`getOutputOrDefault(nodeId, type, default)`	`T`	You need a fallback value
`getOutputIfSuccess(nodeId, type)`	`Optional<T>`	Only care about the output when the whole graph succeeded
`nodeSucceeded(nodeId)`	`boolean`	Check individual node without extracting output
`statusMap()`	`Map<String, NodeStatus>`	Iterate all statuses
`nodeTimings()`	`Map<String, Duration>`	Performance assertions
`errors()`	`List<NodeError>`	Inspect specific failure details
`suspendedNodes()`	`Map<String, String>`	nodeId → suspendKey for nodes that suspended
`compensationResults()`	`List<CompensationResult>`	compensation outcomes after graph failure
`nodeSchemas()`	`Map<String, SchemaDescriptor>`	per-node output schemas

18.8.2 The bloge-test module

The `bloge-test` module provides three tools purpose-built for graph testing:

`MockOperator<I, O>` — a test double that records every invocation.

```
var fetch = MockOperator.<String, String>returning("Alice");
// ... execute graph ...
assertEquals(1, fetch.callCount());
assertEquals("test-input", fetch.lastInput());
```

Factory methods: `returning(value)`, `throwing(exception)`, `delaying(duration, value)`, `recording()`, `of(function)`.

Source: `MockOperator.java`

`GraphTestRunner` — wraps `GraphEngine` with an internal `ExecutionListener` that captures `START:`, `COMPLETE:`, `FAILED:`, and `SKIPPED:` log entries.

```
var runner = new GraphTestRunner(Map.of(
    "validate", validate,
    "price",    price
));
```

```
runner.execute(graph);

runner.assertNodeExecuted("validate");
runner.assertNodeSkipped("unreachable");
runner.assertExecutionOrder("validate", "price");

List<String> log = runner.executionLog();
// ["START:validate", "COMPLETE:validate", "START:price", "COMPLETE:price"]
```

Source: GraphTestRunner.java

DslTestHelper — compiles DSL strings and asserts parse/compile errors.

```
var helper = new DslTestHelper(registry);
Graph graph = helper.compile("""
    graph test {
        node a : MyOperator { input { key = ctx.value } }
    }
    """);

ParseException err = helper.expectParseError("graph { bad }");
assertTrue(err.getMessage().contains("Expected"));
```

Source: DslTestHelper.java

18.8.3 Asserting on suspended nodes, compensation, and schemas

`GraphResult` exposes three additional fields that open up assertion surfaces for suspend/resume workflows, saga-style compensation, and schema contract testing.

`suspendedNodes()` — a `Map<String, String>` where each key is a nodeId and each value is the suspend key the node handed back. Use it to verify that a node correctly suspended and that its key is what your resume logic expects.

```
GraphResult result = engine.execute(graph, context);

// The "approval" node should have suspended
assertTrue(result.isSuspended());
assertEquals("pending-manager-approval",
             result.suspendedNodes().get("approval"));

// Other nodes should not be suspended
assertFalse(result.suspendedNodes().containsKey("validate"));
```

`compensationResults()` — a `List<CompensationResult>` collected after a graph failure when compensation operators are configured. Each entry tells you whether the compensation ran and whether it succeeded.

```
GraphResult result = engine.execute(failingGraph, context);
assertFalse(result.isSuccess());

List<CompensationResult> compensations = result.compensationResults();
assertFalse(compensations.isEmpty());

// Check that the payment node was compensated successfully
CompensationResult paymentComp = compensations.stream()
        .filter(c -> c.nodeId().equals("processPayment"))
        .findFirst().orElseThrow();
assertTrue(paymentComp.isSuccess());
assertEquals("RefundOperator", paymentComp.operatorRef());

// Check that a failed compensation is surfaced, not swallowed
CompensationResult inventoryComp = compensations.stream()
```

```
        .filter(c -> c.nodeId().equals("reserveInventory"))
        .findFirst().orElseThrow();
assertFalse(inventoryComp.isSuccess());
assertNotNull(inventoryComp.error());
```

`nodeSchemas()` — a `Map<String, SchemaDescriptor>` containing per-node output schemas. Use this for **schema contract tests** that verify the shape of each node's output without running the full graph.

```
GraphResult result = engine.execute(graph, context);

// Verify the "fetchUser" node exposes the expected schema
SchemaDescriptor userSchema = result.nodeSchemas().get("fetchUser");
assertNotNull(userSchema);
assertInstanceOf(StructuredSchema.class, userSchema);

StructuredSchema structured = (StructuredSchema) userSchema;
assertTrue(structured.fieldNames().containsAll(
        List.of("id", "name", "email", "vipLevel")));
```

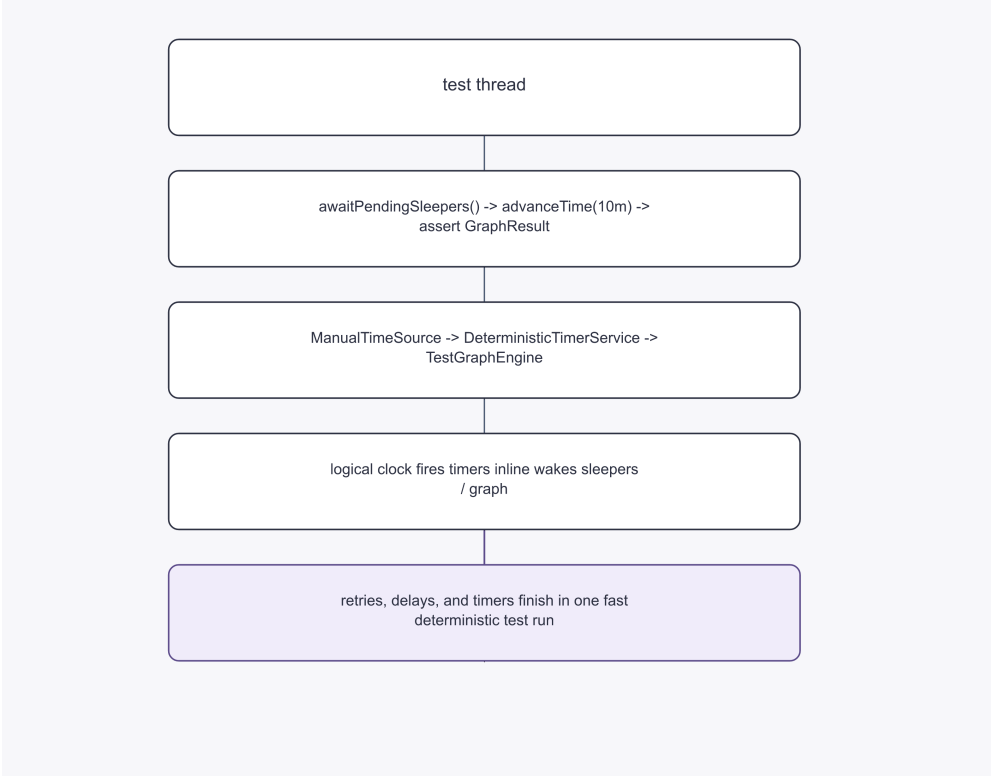
These three fields default to empty collections when not populated, so existing tests continue to pass without changes.

18.8.4 Testing time-dependent workflows

Time is one of the easiest ways for a good test suite to become slow, flaky, or both.

BLOGE's testing module avoids that by giving you deterministic time control:

- `ManualTimeSource` advances only when the test tells it to
- `DeterministicTimerService` fires scheduled timers inline when time advances
- `TestGraphEngine` wires those pieces into a ready-to-use `GraphEngine`




```
var registry = new DefaultOperatorRegistry();
var testEngine = TestGraphEngine.create(registry);
```

```
var op = MockOperator.<Object, String>delaying(Duration.ofMinutes(10), "done");
var builder = new GraphBuilder("delay-test");
Graph graph = builder.node("slow", op).build();

var resultRef = new AtomicReference<GraphResult>();
var done = new CountDownLatch(1);

Thread.startVirtualThread(() -> {
    resultRef.set(testEngine.executeWithOperators(graph, new GraphContext(), builder.operators()));
    done.countDown();
});

testEngine.awaitPendingSleepers(1);      // wait until the delay is actually registered
testEngine.advanceTime(Duration.ofMinutes(10));

assertTrue(done.await(10, TimeUnit.SECONDS));
assertEquals("done", resultRef.get().getOutput("slow", String.class));
```

This pattern is the key detail many teams miss: **advance logical time only after the sleeper or timer has been registered.** That is why `awaitPendingSleepers(...)` matters.

`ManualTimeSource.advance(...)` also fires timer callbacks inline, so one time advance can release sleepers, trigger durable timers, and complete the graph without any wall-clock waiting.

18.8.5 What deterministic time covers well today

Use this harness confidently for:

- delayed operators and retry backoff
- loop / wait timer behavior in plain graphs
- durable lease expiry and timer-driven resume infrastructure
- timer-driven state-machine behavior when the executor is constructed with a dedicated timer service

One caveat is worth making explicit: the current session idle-timeout path is still driven by its own scheduled runtime behavior rather than `ManualTimeSource`, so deterministic unit tests are easiest for graph and state-machine timers today. Session idle-timeout behavior is still worth testing, but it is better covered with focused smoke tests and policy-level assertions than by assuming full logical-clock control.

18.9 Scoping Lease-Fencing Tests — `@WithExecutionLeaseFencing`

When two engines share a durable store, only one of them is supposed to hold the **execution lease** for a given run. If the lease expires and a second engine picks the run up, the original engine must not be allowed to commit a stale checkpoint. The runtime enforces this via a **fencing token** on every store write, but tests need a way to *prove* the enforcement is wired correctly.

`@WithExecutionLeaseFencing` works with `ExecutionLeaseFencingExtension` from `bloge-test`. It does one focused job:

1. temporarily changes the default fencing policy for the annotated test;
2. optionally enables strict rejection when no ambient lease context exists;
3. restores the previous defaults after the test.

It does **not** create an engine, inject stores, expire a lease, or assert every write automatically. The test still owns those arrangements and assertions.

```
@ExtendWith(ExecutionLeaseFencingExtension.class)
@WithExecutionLeaseFencing
class CheckoutDurabilityTest {
```

```
    @Test
    @WithExecutionLeaseFencing(enabled = true, strictMode = true)
    void missingLeaseContextRejectsDurableWrite() {
        InMemoryExecutionStore store = preparedStoreWithExecution("checkout-1");

        assertThrows(StaleFencingEpochException.class, () ->
            store.updateStatus("checkout-1", ExecutionStatus.COMPLETED, 1));
    }
}
```

The helper in this compact example only creates an execution record. For a takeover test, claim once, release, claim from a second owner, then execute the old write inside the first claim's `ExecutionLeaseContext` and assert rejection. That sequence proves stale-owner fencing; the annotation alone does not.

18.10 Snapshot Testing

Asserting a single output value misses the **shape** of an execution. Snapshot testing captures every node lifecycle event into a stable `GraphExecutionSnapshot`, then asserts against the whole shape. This is the right tool for ”the graph still does what it always did” regression tests.

```
SnapshotCapturingListener listener = new SnapshotCapturingListener();

GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(listener))
    .build();

GraphResult result = engine.execute(checkoutGraph, ctx);

GraphExecutionSnapshot snapshot = GraphExecutionSnapshot.capture(result, listener);

GraphSnapshotAssert.assertMatchesBaseline(
    snapshot,
    "checkout-happy-path.snapshot.json"
);
```

`GraphSnapshotAssert` compares a captured snapshot with a JSON baseline:

- `assertMatchesBaseline(...)` uses set semantics for completed and skipped node lists, avoiding false failures from legal parallel completion order.
- `assertMatchesBaselineStrict(...)` also requires the completed-node order to match.
- `assertEquals(...)` and `assertEqualsStrict(...)` compare two in-memory snapshots.
- `updateBaseline(...)` writes a reviewable baseline; use it deliberately, not as an automatic test update.

Use snapshots when the whole observable execution shape is a stable contract. For narrow rules and edge cases, individual assertions usually communicate the failure more directly.

18.11 Common Trap

18.11.1 [X] Asserting only the terminal output

```
GraphResult result = runner.execute(graph);
assertEquals("ORD-001", result.getOutput("confirm", String.class));
```

That assertion proves only one thing: the terminal node returned the value you expected. It does **not** prove that the graph actually succeeded cleanly, that an upstream node was not skipped, or that a resilience rule did not mask a failure you care about.

A safer pattern is to assert in layers:

```
GraphResult result = runner.execute(graph);
result.requireSuccess();
assertTrue(result.nodeSucceeded("validate"));
assertTrue(result.nodeSucceeded("price"));
assertEquals("ORD-001", result.getOutput("confirm", String.class));
```

Start with overall success, then check the few node-level facts that matter to this scenario, and only then assert on final outputs.

18.12 Guided Rewrite

Take the three-node graph from the first example and strengthen its tests in three passes:

1. Replace the real `price` operator with `MockOperator.returning("$42.00")` and assert that it was called exactly once.
2. Execute the graph with `GraphTestRunner` and assert the order `validate → price → confirm`.
3. Break the DSL on purpose — remove the operator type from one node — and use `DslTestHelper.expectCompileError(...)` to assert on the failure instead of discovering it at runtime.

By the end you should have one small graph covered at three layers: compile, execution, and operator behavior.

18.13 Brain Check

1. What is the fastest way to make a test fail with a clear summary of all node failures?
   (`requireSuccess()` on `GraphResult`.)
2. When should you prefer `findOutput(nodeId, type)` over `getOutput(nodeId, type)`?
   (When the node may legitimately be skipped, fail, or be absent in the scenario under test.)
3. What problem does `MockOperator` solve better than a hand-written lambda in many tests?
   (It records invocation data and offers reusable return/throw/delay behavior, so you can assert on calls without extra boilerplate.)
4. What kind of bug is best caught by `DslTestHelper`?
   (A parse or compile bug in the `.bloge` source — missing operators, invalid bindings, schema issues, or malformed syntax.)
5. Why is `GraphTestRunner` more useful than checking only the terminal output?
   (Because it lets you assert execution order, skips, and per-node behavior, not just the final value.)

18.14 Lab

Goal: build a test suite for a small order graph before you trust it.

1. Create a graph `validate → price → confirm` using `GraphBuilder`.
2. Use `MockOperator` for all three operators.
3. Execute the graph with `GraphTestRunner` and assert:
   - all three nodes executed,
   - none were skipped,
   - execution order is `validate`, then `price`, then `confirm`.

4. Execute the same graph again and assert on the returned `GraphResult`:
   - `requireSuccess()` passes,
   - `confirm` returns the expected order ID,
   - `price` shows a non-null timing.
5. Write one failing DSL snippet and assert that `DslTestHelper` reports the compile error you expect.

Stretch: add a fallback to `price`, force the primary behavior to throw, and assert that the graph still succeeds for the reason you intended.

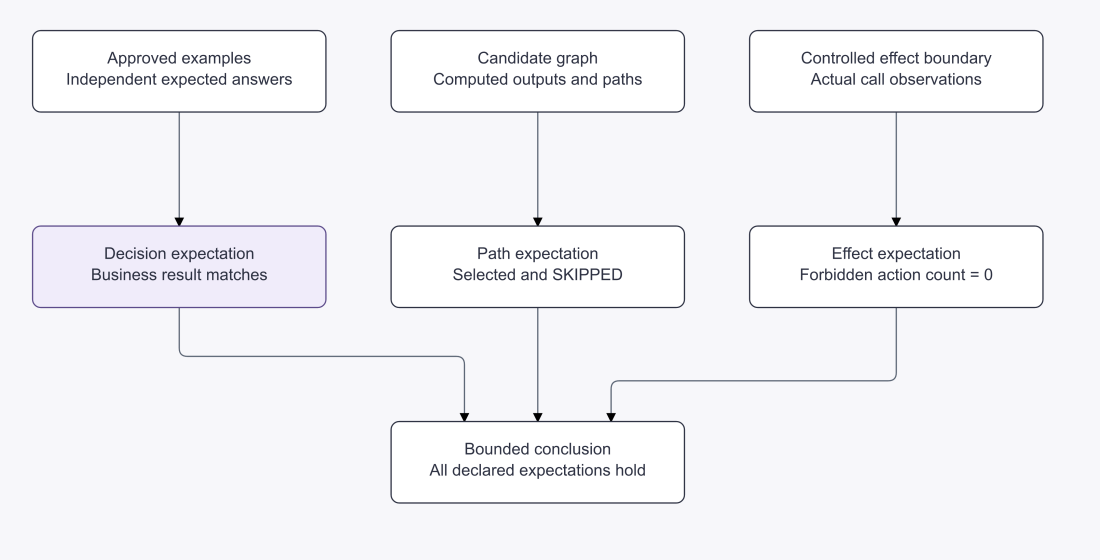
18.15 Bridge: from graph assertions to business verification

`GraphResult`, `GraphTestRunner`, and `DslTestHelper` tell you whether the graph executed as the test expected. They do not bind an owner-approved Scenario, protect required Cases with Policy, control declared effects with Fixture, or publish source-bound evidence. When the question changes from "did this graph run?" to "does this business story still satisfy its approved contract?", continue with Chapter 23 — Business Correctness Is Not Test Pass.

18.16 Experiment acceptance card

- **Expected and observed:** A green graph test supports only its declared graph contract.
- **Failure and recovery:** Remove one node-status assertion to create a false green; restore output, status, and evidence assertions.
- **Proof boundary:** Proves controlled runtime facts, not an external system or business answer.
- **Exercise contract:** One green test; remove one assertion; deliver before/after claim cards; stop when each conclusion points to an assertion.

18.17 Coding practice



18.17.1 BP-14 — Test independent answers and forbidden actions

Use approved expected answers, not current outputs. Observe business results, node paths, and effect evidence independently; mock only the boundary whose behavior is intentionally controlled.

Try one change: Run the workshop blacklist-priority case and its reordered-rule mutant. Explain why preserving old

behavior, rejecting one mutant, and proving all policies correct are different claims.

Reasoning, counterexamples, and limits

18.18 Recap

- **`GraphResult`** is the first place to assert workflow behavior. It also exposes `suspendedNodes()`, `compensationResults()`, and `nodeSchemas()` for suspend/resume, saga compensation, and schema contract assertions.
- **`MockOperator`** keeps tests focused by replacing unrelated I/O with a controllable test double.
- **`GraphTestRunner`** gives you graph-level assertions: order, execution, and skip behavior.
- **`DslTestHelper`** lets you test `.bloge` source the same way the compiler sees it.
- **`TestGraphEngine`**, `ManualTimeSource`, and `DeterministicTimerService` keep time-dependent tests fast and repeatable.
- Strong BLOGE tests stack layers instead of overloading one giant integration test.

18.19 Next Step

In Chapter 19 — Observability in Production you will switch from verifying a declared contract in tests to understanding how a graph behaves when it is already running in the wild.

18.20 Reference Links

- `bloge-test` README — testing helpers overview
- `GraphResult` — immutable execution result record
- `MockOperator` — reusable test doubles
- `GraphTestRunner` — graph harness and assertions
- `DslTestHelper` — DSL parse and compile assertions
- `GraphResultSafeAccessTest.java` — safe-access examples
- Core Architecture — engine internals relevant to graph execution assertions

CHAPTER
19

Arc 5 — Technical Confidence

Chapter 19 — Observability in Production

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

Promise: *By the end of this chapter you will know how to see what a BLOGE graph is doing in production — through listeners, interceptors, metrics, tracing, structured logging, context propagation, and audit journaling — without changing the workflow logic itself.*

19.1 Learning Goals

1. Explain the two main observability extension points: `ExecutionListener` for lifecycle events and `Operator Interceptor` for around-invocation behavior.
2. Capture graph and node lifecycle signals with a custom listener and know which of the fine-grained sub-interfaces to implement.
3. Wire `MetricsExecutionListener`, `TracingOperatorInterceptor`, and `LoggingExecutionListener` into a `GraphEngine` correctly.
4. Use `CallerContextCarrier` to propagate MDC or OpenTelemetry context into the engine's virtual threads.
5. Recognize when aggregate metrics are enough and when you need the structured, per-event fidelity of `Audit JournalListener`.

19.2 Prerequisites

- Chapter 6 — Resilience by Design — retry, timeout, and fallback events matter only if you already understand the resilience surface they describe.
- Chapter 7 — Designing Good Operators — interceptors wrap operator execution, so you need the operator contract first.
- Chapter 18 — Testing Your Graphs — this chapter builds directly on the same assertion and listener surfaces you used in tests.

19.3 Source Examples

File	What it shows
`GraphEngineListenerTest.java`	Lifecycle callbacks and listener exception isolation
`ResilientOperatorWrapperObservabilityTest.java`	Retry, timeout, and fallback events
`MetricsExecutionListenerTest.java`	Micrometer timers and counters
`TracingOperatorInterceptorTest.java`	Graph and node spans
`AuditJournalListenerTest.java`	Structured audit entries and event coverage
`BlogeObservabilityAutoConfiguration.java`	Spring Boot observability auto-configuration

19.4 Why This Matters

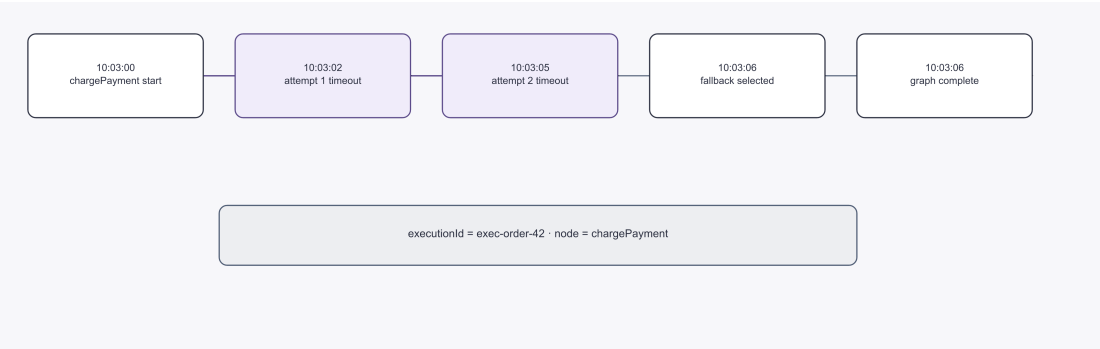
A graph that runs correctly on your machine today can still fail silently in production:

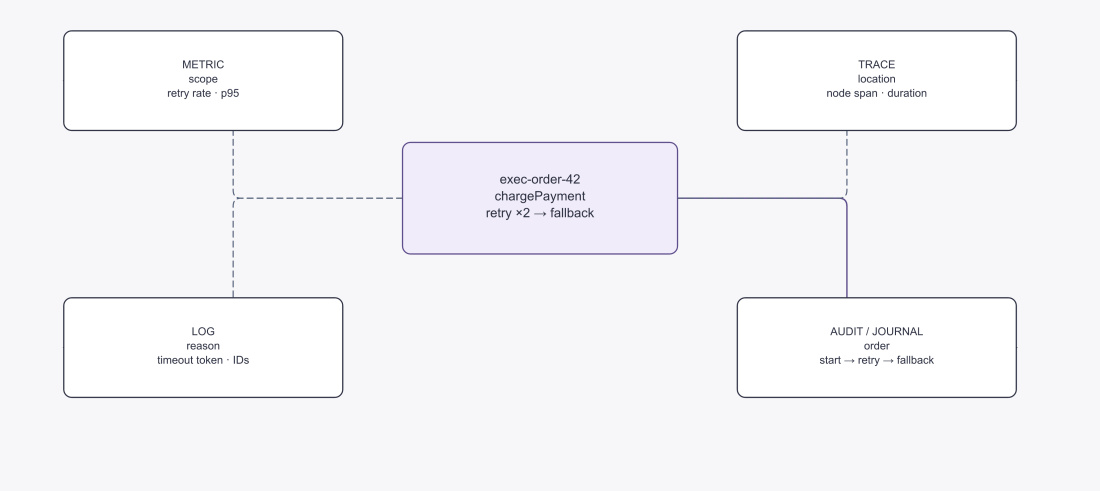
- A **node starts throwing** after a dependency upgrade. Without metrics you won't notice until a customer complains.
- A **retry loop** burns 30 seconds per request. Without per-node timing data you can't identify the bottleneck.
- A **branch change** skips a node that used to run. Without lifecycle signals, the system looks idle when it is actually making the wrong choice.

BLOGE separates concerns: the **engine** runs nodes, and **extension hooks** observe what happened. That separation is what makes production visibility possible without forcing metrics, tracing, or logging logic into every operator.

19.5 One Incident, One Execution ID, Four Projections

At 10:03, the order dashboard reports a latency spike. The failing request is `executionId=exec-order-42`; its `chargePayment` node retries twice and then uses a fallback. Keep that identity fixed while changing the question:



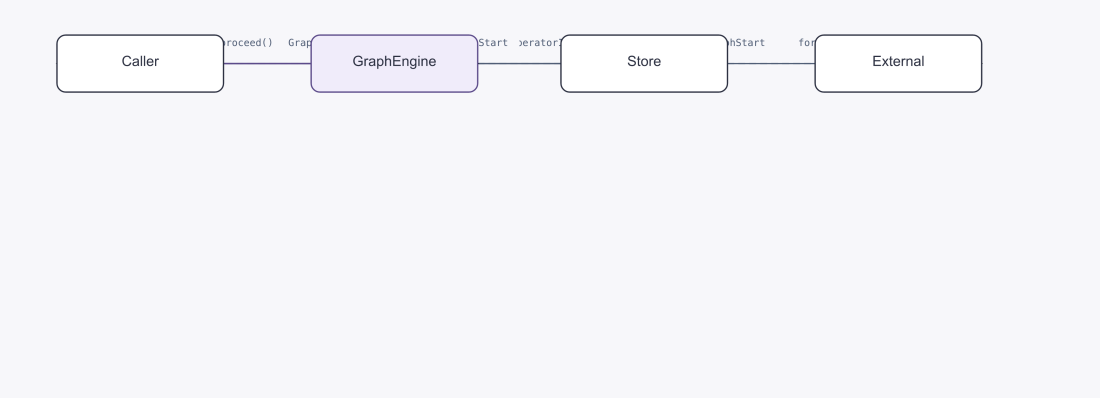


Projection	Best question	Observation for `exec-order-42`	What it cannot establish alone
Metric	Is this widespread?	retry count and p95 latency rose	Which exact attempt failed
Trace	Where was time spent?	`chargePayment` child span dominates	The durable event sequence
Structured log	What did the runtime report?	timeout token plus execution/node IDs	A complete, ordered history
Audit/event journal	What happened in order?	start → retry → retry → fallback → complete	Fleet-wide rate without aggregation

The correlation key is the spine, not a fifth observability product. A metric starts the investigation; the same execution ID narrows the trace and logs; the journal reconstructs order. This incident is a teaching reconstruction from source-tested event types, not a captured production outage.

19.6 Mental Model

Think of the engine as a stage with fixed spotlight positions. Every node execution walks through the same spotlight sequence:



If you know which spotlight position you need, you know where to plug in your observability feature.

19.7 First Working Example

A minimal production-style wiring combines a listener and an interceptor around an otherwise ordinary engine:

```
var meterRegistry = new SimpleMeterRegistry();
var metrics = new MetricsExecutionListener(meterRegistry, "bloge");
var tracing = new TracingOperatorInterceptor(tracer);

var engine = GraphEngine.builder()
        .registry(registry)
        .listeners(List.of(metrics, tracing))
        .interceptors(List.of(tracing))
        .build();
```

That one builder call gives you three different views of the same execution:

- graph/node duration metrics,
- graph and node tracing spans, and
- the raw lifecycle event stream that powers both.

The important detail is not the specific library. It is the contract: listeners observe events; interceptors wrap execution.

19.8 Break It Apart

19.8.1 ExecutionListener — the event backbone

`ExecutionListener` is a composite interface that extends seven fine-grained sub-interfaces:

Sub-interface	Callbacks
`GraphLifecycleListener`	`onGraphStart`, `onGraphComplete`
`NodeLifecycleListener`	`onNodeStart`, `onNodeComplete`, `onNodeFailed`, `onNodeSkipped`, `onNodeSuspended`, `onNodeResumed`
`ResilienceListener`	`onNodeRetry`, `onNodeTimeout`, `onNodeFallback`
`StreamingListener`	`onNodeStreamStart`, `onNodeStreamChunk`, `onNodeStreamEnd`, `onNodeStreamError`
`IterationLifecycleListener`	Loop/foreach iteration events
`TimerEventListener`	Timer-fired events
`EventCorrelationListener`	Event-matching events

New integrations can implement just the narrowest sub-interface they need. Existing code can continue implementing the aggregate `ExecutionListener`.

> ***Safety guarantee:*** *A throwing listener does not break graph execution. The engine catches listener exceptions and continues with the remaining listeners. This is tested in* `GraphEngineListenerTest.listener_exceptionInListener_doesNotBreakExecution`.

19.8.2 OperatorInterceptor — the AOP layer

`OperatorInterceptor` wraps every operator call. The interceptor receives an `OperatorInvocation` and must call `invocation.proceed()` to continue the chain (or short-circuit by returning a value directly).

```
public interface OperatorInterceptor {
    Object intercept(OperatorInvocation invocation) throws Exception;
}
```

This is how `TracingOperatorInterceptor` creates child spans per node — it starts a span before `proceed()` and ends

it in the `finally` block.

19.8.3 CallerContextCarrier — context across virtual threads

The engine runs each node on a virtual thread. Thread-local state like OpenTelemetry context or SLF4J MDC does not propagate automatically. `CallerContextCarrier` bridges this gap with a three-phase contract:

1. **`capture()`** — called on the caller thread, snapshots the context.
2. **`Snapshot.attach()`** — called on the virtual thread, restores it.
3. **`Scope.close()`** — called when the node finishes, cleans up. `Scope.NOOP` is available for carriers that need no cleanup.

Two implementations ship in `bloge-metrics-otel`:

- `OtelContextCarrier` — propagates `io.opentelemetry.context.Context`
- `MdcContextCarrier` — propagates SLF4J MDC maps

19.8.4 Production observability components

The `bloge-metrics-otel` module provides three ready-to-use components:

`MetricsExecutionListener` — emits Micrometer timers and counters:

- `bloge.graph.duration` — Timer, tagged `graph` + `outcome`
- `bloge.node.duration` — Timer, tagged `graph` + `node` + `outcome`
- `bloge.node.errors` — Counter, tagged by exception class
- `bloge.node.retries`, `bloge.node.timeouts`, `bloge.node.fallbacks`, `bloge.node.skipped` — Counters
- `bloge.stream.chunk.count`, `bloge.stream.duration`, `bloge.stream.errors` — Streaming metrics

All names use a configurable prefix (default `bloge`).

Source: `MetricsExecutionListener.java`

`TracingOperatorInterceptor` — creates a parent span (`bloge.graph.execute`) on graph start and a child span (`bloge.node.execute`) around each operator invocation. It implements **both** `OperatorInterceptor` and `ExecutionListener` — register the same instance in both the `interceptors` and `listeners` lists.

Source: `TracingOperatorInterceptor.java`

`LoggingExecutionListener` — structured SLF4J logs with `bloge.graph`, `bloge.node`, and `bloge.executionId` MDC keys. Two optional flags — `includeInput` and `includeOutput` — control whether payloads appear in logs. **Keep both disabled unless you have reviewed payloads for sensitive data.**

Source: `LoggingExecutionListener.java`

19.8.5 Audit journal

For compliance-grade event recording, `bloge-runtime-spi` includes `AuditJournalListener`. It captures `NODE_START`, `NODE_COMPLETE`, `NODE_FAILED`, `NODE_RETRY`, `NODE_SUSPEND`, and `NODE_RESUME` events as structured `AuditEntry` records and writes them to an `AuditJournalStore`. An asynchronous flush mode (virtual-thread worker + configurable batch size) keeps the hot path lightweight.

`AuditJournalListener` accepts an optional `TimeSource` parameter so that audit timestamps are derived from the same clock the engine uses. In tests you can inject `ManualTimeSource` to get deterministic audit entries; in production the default `SystemTimeSource.INSTANCE` is used.

```
// Production — wall clock
var listener = new AuditJournalListener(store, config, jsonCodec);

// Test — deterministic time
var listener = new AuditJournalListener(store, config, jsonCodec, manualTimeSource);
```

`AuditConfig` controls: `captureInput`, `captureOutput`, `asyncFlush`, `flushInterval` (default 100ms), `batchSize` (default 64).

19.8.6 ExecutionEventStore — tenant-scoped queries and purging

The `ExecutionEventStore` interface exposes several methods that matter for production observability pipelines:

Method	Purpose
`loadEventsByTenant(tenantId, namespace, from, to, page, size)`	Page through events scoped to one tenant/namespace within a time range
`countEventsByTenant(tenantId, namespace)`	Count total events for a tenant — useful for dashboard metrics
`purgeEventsBefore(cutoff)`	Delete events older than a cutoff instant
`purgeEventsBefore(cutoff, limit)`	Batch-aware purge to avoid long-running delete transactions
`purgeExecutionEvents(executionId)`	Delete all events for a single execution (e.g. after archival)

All new methods have default implementations that throw `UnsupportedOperationException`, so existing store implementations continue to compile. Upgrade them incrementally as your observability pipeline needs these capabilities.

19.9 Read Stable Log Tokens by Severity

Metrics and traces tell you that **something** is slow. Stable log tokens identify a specific runtime or DSL fallback without making the surrounding prose part of the contract. Alert policy belongs on the emitted severity and operational impact; `FINE`/`FINER` tokens are diagnostic breadcrumbs, not alarms by themselves.

Token	Severity	What it means	First step
`[TIMER_RESTORE_FAILED]`	SEVERE	`TimerManager` could not reload active durable timers; persisted timers will not fire until the next restart.	Inspect the attached exception, durable timer context, store connectivity, and the records being loaded.
`[STREAMING_ITEM_ERROR_SEND_FAILED]`	WARNING	A streaming `foreach` item failed and the runtime could not emit its error chunk because the output channel also rejected the send.	Inspect the original item failure, channel-close race, and selected item-failure policy.
`[SCHEMA_NUMBER_PARSE_FALLBACK]`	FINER	Number coercion is trying a wider numeric type after an integer or long parse miss.	Use for coercion diagnosis; alert only on a separately observed business or error-rate impact.
`[ACCESSOR_METHOD_LOOKUP_MISS]`	FINER	A direct method accessor was absent, so DSL path access is trying a JavaBean getter fallback.	Confirm the property/getter contract when path resolution later fails.
`[SCHEMA_ENRICH_SKIPPED]`	FINE	Operator metadata lookup failed during DSL schema enrichment; compilation continues with reduced enrichment.	Inspect the attached cause and registry metadata if diagnostics lose schema detail.
`[FINGERPRINT_SKIPPED]`	FINE	The DSL compiler could not compute an Operator fingerprint and continues without it.	Inspect Operator registration and fingerprint support before relying on mismatch detection during recovery.

*These tokens are **stable**; the surrounding text is not. Match the bracketed token, not the prose.*

19.10 Execution Event Journal

`EventJournalListener` is a built-in `ExecutionListener` that records **every** engine-level event into an append-only journal. Unlike the audit journal (which captures *state changes* for durability) or the operator interceptor chain (which wraps *single node calls*), the event journal is your **replayable** record of an execution.

```
ExecutionEventJournal journal = new InMemoryExecutionEventJournal();

GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(new EventJournalListener(journal)))
    .build();
```

Each entry is an `ExecutionEvent` carrying an execution id, monotonic sequence number, timestamp, and one of these types:

Type	Emitted when
`GRAPH_STARTED` / `GRAPH_COMPLETED` / `GRAPH_FAILED`	Engine enters/leaves a graph.
`NODE_STARTED` / `NODE_COMPLETED` / `NODE_FAILED` / `NODE_SKIPPED`	Per node lifecycle.
`FOREACH_BATCH_STARTED` / `FOREACH_BATCH_COMPLETED`	Each batch of a `foreach`.
`STREAM_ITEM_EMITTED` / `STREAM_COMPLETED`	Streaming sources.
`WAIT_FOR_TIMER_ARMED` / `WAIT_FOR_TIMER_FIRED`	Durable timer lifecycle.
`LEASE_ACQUIRED` / `LEASE_REFRESHED` / `LEASE_LOST`	Durable lease lifecycle.

19.10.1 Agent events — `AgentEventJournalBridge`

When you use the `bloge-agent-ext` module, an `AgentEventJournalBridge` turns agent-internal lifecycle events into entries on the **same** journal. This gives you a unified replay timeline of "graph turn + agent turn + tool call":

```
GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(
        new EventJournalListener(journal),
        new AgentEventJournalBridge(journal)))   // route agent events too
    .build();
```

Agent event types added by the bridge:

Type	Meaning
`AGENT_TURN_STARTED` / `AGENT_TURN_COMPLETED`	Agent's reasoning loop iteration.
`AGENT_LLM_CALLED`	LLM request sent (carries token usage and model id).
`AGENT_TOOL_INVOKED` / `AGENT_TOOL_COMPLETED` / `AGENT_TOOL_FAILED`	Tool/function call lifecycle.
`AGENT_MEMORY_TRIMMED`	A memory strategy dropped/summarised earlier turns.
`AGENT_STREAM_CHUNK_EMITTED`	A streaming agent produced a chunk.

A common pattern is to keep an in-memory journal for short-lived debugging and ship a `JdbcExecutionEventJournal` to your warehouse for long-term replay.

19.11 Common Trap

19.11.1 [X] Registering `TracingOperatorInterceptor` only as a listener

```
var tracing = new TracingOperatorInterceptor(tracer);

var engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(tracing))       // graph span [OK]
```

```
    // missing .interceptors(List.of(tracing))  — no node spans! [X]
    .build();
```

`TracingOperatorInterceptor` wears two hats. The `ExecutionListener` side creates the **graph-level** span in `onGraphStart` / `onGraphComplete`. The `OperatorInterceptor` side creates **node-level** child spans in `intercept()`. If you only register it as a listener, you will see a single flat span per execution with no per-node breakdown.

Fix: register the **same instance** in both lists.

19.12 Guided Rewrite

Open `MetricsExecutionListenerTest.java` and adapt it in three ways:

1. Change the metric prefix to `myapp.bloge` and verify that the timer names follow the new prefix.
2. Add a fallback path and assert that the fallback counter increments exactly once.
3. Register an inline listener that records event strings and compare its event order with the metrics emitted for the same execution.

The exercise is useful because it keeps the graph constant while you vary the observability lens.

19.13 Brain Check

1. What is the difference between an `ExecutionListener` and an `OperatorInterceptor`?
   (A listener reacts to lifecycle events after they are emitted; an interceptor wraps the operator invocation itself and can add behavior around it.)
2. Why must `TracingOperatorInterceptor` be registered in two places?
   (Its listener side manages the graph-level span; its interceptor side creates node-level spans.)
3. What problem does `CallerContextCarrier` solve?
   (It propagates caller-thread context such as MDC or OpenTelemetry state into the engine's virtual threads.)
4. When should you choose `AuditJournalListener` over plain metrics?
   (When you need structured, per-event auditability instead of aggregate counters or timers.)
5. What guarantee does the engine provide if a listener throws?
   (Listener failures are isolated; execution continues and other listeners still run.)

19.14 Lab

Goal: make one graph observable in three complementary ways.

1. Build a small graph with one successful node and one flaky node.
2. Register `MetricsExecutionListener` and verify a retry counter appears.
3. Register `TracingOperatorInterceptor` in both places and verify graph and node spans are created.
4. Register `AuditJournalListener` with an in-memory store and assert that the expected `NODE_START`, `NODE_RETRY`, and `NODE_COMPLETE` events were stored.
5. Add MDC propagation with `MdcContextCarrier` and verify a node log line sees the caller's correlation ID.

Stretch: run the same graph once with observability enabled and once without it, then compare what each lens helps you answer fastest.

19.15 Experiment acceptance card

- **Expected and observed:** One executionId connects metric, trace, log, and audit.
- **Failure and recovery:** Remove correlation; restore the shared identity and rebuild the timeline.
- **Proof boundary:** Proves projections can correlate, not that metrics explain business cause.
- **Exercise contract:** One retry-to-fallback incident; change correlation only; deliver four projections; stop when all resolve to one execution.

19.16 Recap

- **Listeners** observe lifecycle events; **interceptors** wrap operator calls.
- **Metrics**, **tracing**, **logging**, and **audit journaling** all hang off the same engine extension surfaces.
- **`CallerContextCarrier`** is the missing link between caller-thread context and virtual-thread execution.
- The safest observability integrations are reusable because they add visibility without changing graph logic.

19.17 Next Step

In Chapter 20 — Spring and Production Wiring, you will turn runtime visibility into an application integration: conditional beans, configuration boundaries, and operator-facing endpoints.

19.18 Reference Links

- `ExecutionListener` — aggregate listener interface
- `OperatorInterceptor` — AOP interception contract
- `CallerContextCarrier` — capture/attach/close contract
- `MetricsExecutionListener` — Micrometer listener
- `TracingOperatorInterceptor` — graph and node span wiring
- `LoggingExecutionListener` — structured logging listener
- `AuditJournalListener` — structured audit capture
- `ExecutionEventStore` — event journal storage SPI with tenant-scoped queries
- `BlogeObservabilityAutoConfiguration` — Spring Boot auto-configuration
- Core Architecture — engine internals and SPI overview

CHAPTER
20

Arc 5 — Technical Confidence

Chapter 20 — Spring and Production Wiring

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

Promise: *By the end of this chapter you will be able to start one BLOGE graph through Spring Boot with no manual engine bean, read the health result, and apply one deliberate `GraphEngineCustomizer` override without taking ownership of the whole auto-configuration graph.*

20.1 Learning Goals

1. Add the `bloge-spring` dependency and let `BlogeAutoConfiguration` wire a `GraphEngine`, `OperatorRegistry`, `GraphLoader`, and compiled `List<Graph>` with zero manual bean definitions.
2. Annotate runtime `Operator`, `SuspendableOperator`, or `StreamingOperator` implementations with `@BlogeOperator` so Spring discovers and registers them.
3. Inject `GraphEngine` and the compiled graph list into a service class, execute a graph, and return results from an HTTP endpoint.
4. Read `UP + graphCount=1` as a wiring observation, not as business proof.
5. Apply one lightweight-engine override with `GraphEngineCustomizer`, and know when replacing a bean would create unnecessary ownership.

20.2 Prerequisites

- Chapter 7 — Designing Good Operators — you need the operator design model before wiring operators into Spring.
- Chapter 13 — Durable Execution — runtime stores, checkpoints, and suspend/resume, since the starter conditionally wires all of them.
- Familiarity with Spring Boot auto-configuration, `@ConfigurationProperties`, and Spring Boot Actuator basics.

20.3 Source Examples

File	What it shows
`BlogeAutoConfiguration.java`	Entry auto-configuration that imports the core slice

File	What it shows
BlogeProperties.java	All spring.bloge.* configuration properties and nested groups
BlogeOperator.java	The @Component-composed annotation for operator scanning
BlogeObservabilityAutoConfiguration.java	Metrics, tracing, MDC, health indicator wiring
BlogeEndpointAutoConfiguration.java	/actuator/bloge and /actuator/blogeStats endpoints
BlogeAuditAutoConfiguration.java	Audit journal listener activation
SessionExecutorAutoConfiguration.java	SessionExecutor wiring with access guards and listeners
spring-ticket-triage.bloge	Minimal DSL loaded by the starter example
ch19/loan-approval.bloge	Loan approval graph wired with Spring operators
SpringBootTicketTriageApplication.java	End-to-end Spring Boot application using the starter
SpringTicketTriageService.java	Service layer injecting GraphEngine and List<Graph>

20.4 Why This Matters

In earlier chapters you built engines by hand:

```
var engine = GraphEngine.builder()
        .registry(registry)
        .executionCheckpointStore(checkpointStore)
        .listeners(listeners)
        .build();
```

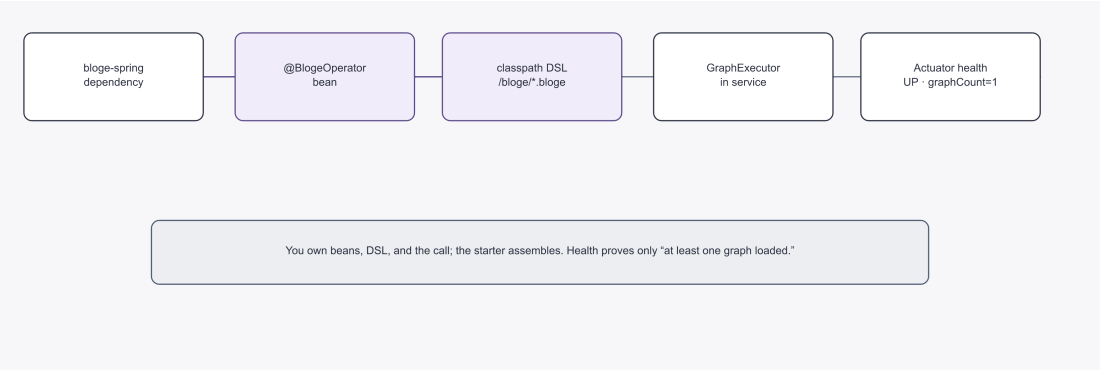
That's fine for tests and learning. In production, the wiring list grows: interceptors, listeners, context carriers, durable stores, checkpoint codecs, recovery config, shard resolvers, timer services, event deduplication — and every bean has conditions and fallbacks.

The bloge-spring module addresses this assembly work. It is a Spring Boot Starter that auto-configures components whose classpath and property conditions are satisfied. Declaring operators as Spring beans and placing .bloge files on the classpath lets the starter assemble a runnable engine for that application slice. Adding bloge-durable or bloge-metrics-otel lets the starter detect and wire their conditional beans without repeated @Bean declarations. This establishes application wiring; it does not prove that durable stores, external effects, or production capacity are ready.

> ***Key insight:*** *The starter doesn't hide the engine — it assembles it. Most replaceable singleton defaults use @ConditionalOnMissingBean; extension collections and endpoint slices follow their own conditions, so confirm the target bean's contract before overriding it.*

20.5 The Shortest Wiring Path

Do not begin with every optional production feature. Begin with one vertical slice: one dependency, one annotated operator, one classpath DSL, one injected executor, and one health observation.

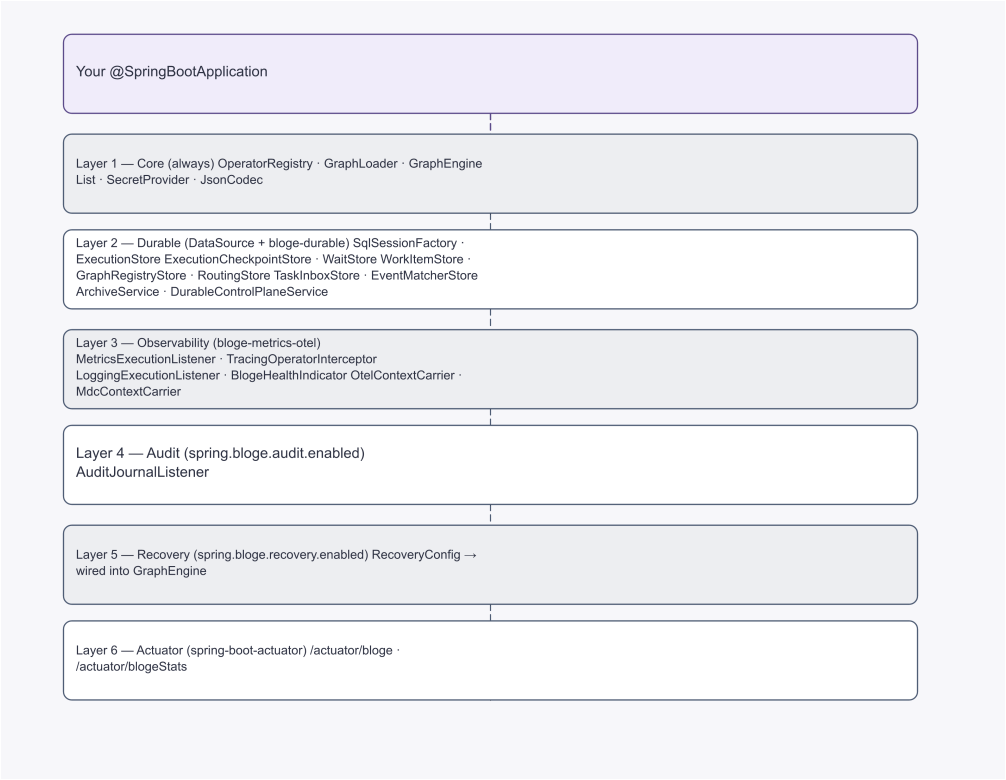


You own	The starter owns	Observable check
`@BlogeOperator` bean implementing a runtime operator contract	`OperatorRegistry`	operator name resolves during DSL load
`classpath:/bloge/*.bloge`	`GraphLoader` and compiled `List<Graph>`	expected graph is present
service input and output	lightweight `GraphEngine` / `GraphExecutor`	one execution returns the expected node facts
deployment dependencies	`BlogeHealthIndicator` when Actuator is present	`UP`, `graphCount=1`

`UP` here means the application loaded at least one graph. It does not mean an order Scenario passed, an external payment system is healthy, or a release is qualified.

20.6 Mental Model

Think of `bloge-spring` as a **layered assembly line** for beans. Each layer activates only when its classpath and property prerequisites are met:



Layers 2–6 are additive and independent. You never need all of them at once, and removing a classpath dependency removes the layer cleanly.

20.7 First Working Example

This is the ticket-triage example from `bloge-examples/src/main/java/com/leanowtech/bloge/examples/integration/spring/`.

20.7.1 Step 1 — Add the dependency

```
<dependency>
    <groupId>com.leanowtech.bloge</groupId>
    <artifactId>bloge-spring</artifactId>
    <version>${bloge.version}</version>
</dependency>
```

20.7.2 Step 2 — Write an operator as a Spring bean

Simplified from `SpringTicketClassifierOperator.java`:

```
@BlogeOperator(
        value = "SpringTicketClassifierOperator",
        description = "Classifies a support ticket",
        owner = "examples",
        tags = {"spring", "starter", "triage"}
)
public class SpringTicketClassifierOperator implements
        Operator<Map<String, Object>, ClassifiedTicket> {

    public record ClassifiedTicket(String queue, int priorityScore, boolean vip) {}
```

```
    @Override
    public ClassifiedTicket execute(Map<String, Object> input, OperatorContext ctx) {
        boolean vip = "vip".equalsIgnoreCase(String.valueOf(input.get("customerTier")));
        String queue = vip ? "vip-escalation" : "general-support";
        int priorityScore = vip ? 70 : 30;
        return new ClassifiedTicket(queue, priorityScore, vip);
    }
}
```

`@BlogeOperator` is composed from `@Component`, so Spring discovers the bean. The registry then accepts it only when the bean implements `Operator`, `SuspendableOperator`, or `StreamingOperator`. The `value` attribute sets the DSL operator name. The repository example deliberately keeps its domain beans plain and supplies adapters through an explicit `OperatorRegistry`; the compact version above chooses the zero-manual-registry path instead.

20.7.3 Step 3 — Place the DSL on the classpath

From `spring-ticket-triage.bloge`:

```
graph springTicketTriage {
  node classifyTicket : SpringTicketClassifierOperator {
    input {
      ticketId = ctx.ticketId
      message = ctx.message
      customerTier = ctx.customerTier
    }
    timeout = 2s
  }

  node draftReply : SpringReplyDraftOperator {
    depends_on = [classifyTicket]
    input {
      ticketId = ctx.ticketId
      queue = classifyTicket.output.queue
      priorityScore = classifyTicket.output.priorityScore
      vip = classifyTicket.output.vip
    }
    timeout = 2s
  }
}
```

The starter's default DSL location is `classpath:bloge/`. Place this file at `src/main/resources/bloge/spring-ticket-triage.bloge` and it will be loaded automatically.

20.7.4 Step 4 — Inject and execute

From `SpringTicketTriageService.java`:

```
@Service
public class SpringTicketTriageService {

    private final GraphEngine engine;
    private final Map<String, Graph> graphsByName;

    public SpringTicketTriageService(GraphEngine engine, List<Graph> graphs) {
        this.engine = engine;
        this.graphsByName = graphs.stream()
            .collect(Collectors.toUnmodifiableMap(Graph::name, Function.identity()));
    }
```

```
    public TicketTriageResponse triage(String ticketId, String message, String customerTier) {
        Graph graph = graphsByName.get("springTicketTriage");
        var result = engine.execute(graph, new GraphContext(Map.of(
                "ticketId", ticketId,
                "message", message,
                "customerTier", customerTier
        )));
        return /* map result to response */;
    }
}
```

No `GraphEngine.builder()`. No `DefaultOperatorRegistry`. No `GraphLoader` calls. The starter assembled all of it.

20.7.5 Step 5 — Expose via HTTP

From `SpringTicketTriageController.java`:

```
@RestController
@RequestMapping("/api/bloge/tickets")
public class SpringTicketTriageController {

    private final SpringTicketTriageService triageService;

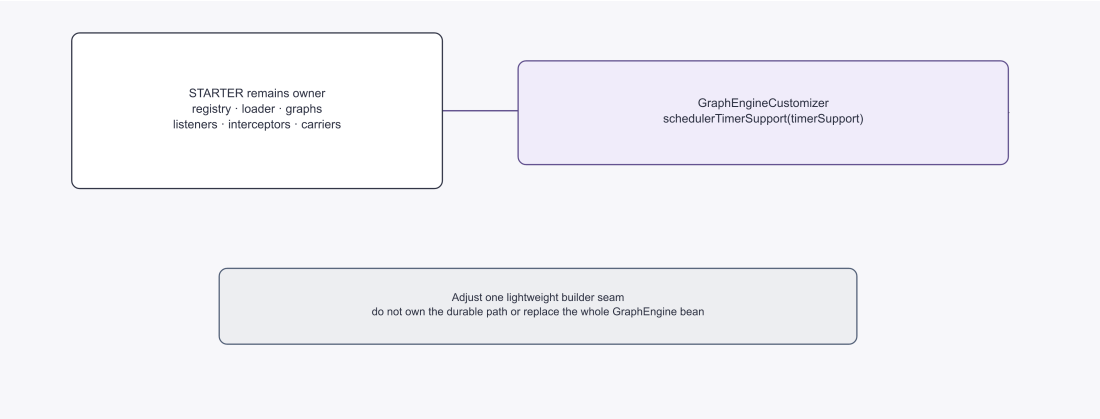
    public SpringTicketTriageController(SpringTicketTriageService triageService) {
        this.triageService = triageService;
    }

    @GetMapping("/triage")
    public TicketTriageResponse triage(
            @RequestParam("ticketId") String ticketId,
            @RequestParam("message") String message,
            @RequestParam(name = "customerTier", defaultValue = "standard") String customerTier) {
        return triageService.triage(ticketId, message, customerTier);
    }
}
```

20.8 One Override, One Explicit Owner

Suppose the default lightweight engine is correct except for one scheduler integration. Keep every starter-owned bean and customize only the builder seam:

```
@Bean
GraphEngineCustomizer schedulerOverride(MyTimerSupport timerSupport) {
    return builder -> builder.schedulerTimerSupport(timerSupport);
}
```



The customizer runs after the lightweight engine-mode preset and before the core `GraphEngine` is built. It does **not** configure the durable runtime path. If you replace the whole `GraphEngine` bean, you also accept responsibility for its listeners, interceptors, context carriers, and future starter defaults.

The remainder of this chapter is a production reference. The first-read path is complete once the minimal slice, health boundary, and single override are clear.

20.9 Production Reference

The shortest path above is enough to understand ownership. When you need the complete bean inventory, property groups, durable-store wiring, health checks, or HTTP error contract, continue with Appendix I — Spring Production Reference.

20.10 Common Trap

20.10.1 [X] Registering operators manually when `@BlogeOperator` already exists

```
// DON'T do this if you also annotate the class with @BlogeOperator:
@Bean
public OperatorRegistry operatorRegistry() {
    var registry = new DefaultOperatorRegistry();
    registry.register("FetchUserOperator", new FetchUserOperator(userService));
    return registry;
}
```

Because `BlogeAutoConfiguration` provides an `OperatorRegistry` bean that scans for `@BlogeOperator`-annotated beans, **and** its bean is `@ConditionalOnMissingBean`, declaring your own `OperatorRegistry` bean replaces the auto-configured one entirely. Your custom registry won't scan for `@BlogeOperator` beans — so all annotated operators silently disappear.

Fix: Either rely entirely on `@BlogeOperator` scanning (the normal path), or provide a complete `OperatorRegistry` bean that includes everything you need. Don't mix both approaches.

20.11 What Goes Wrong

20.11.1 Silent operator disappearance

You annotate a class with `@BlogeOperator("MyOperator")` but also define a custom `OperatorRegistry` bean:

```
@Bean
public OperatorRegistry operatorRegistry() {
```

```
    return new DefaultOperatorRegistry();  // empty — no scanning
}
```

The auto-configured scanning registry is skipped because your bean exists. At runtime:

```
GraphDefinitionException: Node 'doWork' references unregistered operator 'MyOperator'
```

Fix: Either rely on auto-scanning (remove the custom `@Bean`) or populate your custom registry with all required operators.

20.12 One Production Extension at a Time

Add observability, audit, recovery, or an interceptor only when a stated operational need requires it. Appendix I provides the configuration map and a deployment checklist; keep this chapter's design rule: each override must have one reason and one owner.

20.13 Brain Check

1. What is required for automatic operator registration? *(Answer: the bean is marked with `@BlogeOperator` and implements `Operator`, `SuspendableOperator`, or `StreamingOperator`; the annotation alone only guarantees Spring component discovery.)*
2. Where does the starter look for `.bloge` files by default, and which property overrides this? *(Answer: `classpath:bloge/`, configurable via `spring.bloge.dsl-locations`.)*
3. If you declare your own `OperatorRegistry` bean, what happens to the auto-configured one? *(Answer: it is skipped — `@ConditionalOnMissingBean` means your bean wins, and automatic `@BlogeOperator` scanning no longer occurs.)*
4. Name two production features that activate purely by adding a classpath dependency, with no property changes. *(Answer: Actuator endpoints activate when `spring-boot-actuator` is present; observability metrics and tracing activate when `bloge-metrics-otel` is present.)*
5. What does `BlogeHealthIndicator` report when no `.bloge` files are found? *(Answer: `DOWN` with detail `"reason": "No graphs loaded"`.)*
6. *Design question:* Your team uses three separate Spring Boot microservices, each with its own `bloge-spring` dependency and different graphs. Should each service have its own `GraphEngine` instance, or should they share one central engine? *(Each service should have its own `GraphEngine`. The engine is stateless and lightweight — there is no benefit to centralising it. Sharing a database for durable stores is fine if the services need cross-service visibility, but the engines themselves are independent.)*

20.14 Lab

Starting from a fresh Spring Boot application:

1. **Add `bloge-spring`** to your POM and create one `Operator` implementation annotated with `@BlogeOperator` that returns a greeting string.
2. **Write a one-node `.bloge` graph** in `src/main/resources/bloge/` that references your operator.
3. **Create a `@Service`** that injects `GraphEngine` and `List<Graph>`, looks up your graph by name, and executes it.
4. **Verify the health indicator**: start the application and hit `/actuator/health`. Confirm the `bloge` component shows `UP` with your graph name.
5. **Break it intentionally**: rename the `.bloge` file so no graphs load. Restart and verify the health indicator reports `DOWN`.
6. **Add a custom `ExecutionListener`** bean that logs graph-level completion times. Execute your graph again and confirm the listener fires without any changes to the engine bean.

20.15 Experiment acceptance card

- **Expected and observed:** The starter wires engine, loader, and health with one owner per customizer.
- **Failure and recovery:** Remove a bean or duplicate an override; recover minimal auto-configuration.
- **Proof boundary:** Proves Spring wiring and health, not production capacity or business correctness.
- **Exercise contract:** Minimal app; change one bean/customizer; deliver context and health; stop with successful startup and one owner.

20.16 Coding practice

20.16.1 BP-05 — Separate wiring from new capabilities

Endpoint and credential changes normally belong in configuration; provider protocol changes belong behind an adapter. Changed business guarantees require a new capability or explicit version, while ordered steps belong in a graph.

Try one change: Classify changing a payment URL, replacing a provider, and replacing authorization with charging. State which changes affect callers.

Reasoning, counterexamples, and limits

20.17 Recap

- `bloge-spring` is a Spring Boot Starter that can auto-configure `GraphEngine`, `OperatorRegistry`, `GraphLoader`, and compiled graphs when their classpath, property, and bean conditions are satisfied.
- `@BlogeOperator` is a `@Component`-composed annotation. Annotated beans are registered automatically only when they implement one of BLOGE's runtime operator contracts; plain domain beans need an adapter/customizer path. Its `promptHint`, `usageExample`, and `constraintsDescription` attributes support agent/LLM integration.
- The starter uses **layered, conditional auto-configuration**: core beans are present inside an active starter context; durable stores activate when `bloge-durable` + `DataSource` exist; observability activates when `bloge-metrics-otel` exists; audit and recovery activate when their respective properties are set to `true`.
- The auto-configuration is split into focused slices (`BlogeCoreAutoConfiguration`, `BlogeGraphEngineAutoConfiguration`, `BlogeDurableAutoConfiguration`, etc.) for cleaner separation.
- New `spring.bloge.engine-mode` property selects engine presets (`AUTO` or `REQUEST_RESPONSE`).
- New `spring.bloge.tenant.*` properties enable auto-configured multi-tenant support — tenant resolution, namespace resolution, concurrency caps, and rate limits — without writing a custom `TenantContextResolver` bean.
- New `spring.bloge.version-routing.*` properties support `latest`, `pinned`, and `canary` graph version routing at the starter level.
- `GraphEngineCustomizer` lets you fine-tune the engine builder without replacing the entire bean.
- Most replaceable singleton defaults are guarded by `@ConditionalOnMissingBean`; inspect the specific slice before overriding it.
- Custom `OperatorInterceptor`, `ExecutionListener`, and `CallerContextCarrier` beans are collected automatically via `ObjectProvider` and injected into the engine.
- Actuator endpoints (`/actuator/bloge`, `/actuator/blogeStats`) give runtime visibility into loaded graphs and execution metrics.
- `BlogeHealthIndicator` provides `/actuator/health` integration out of the box.

20.18 Next Step

In Chapter 21 — Scheduling and Complexity in One JVM, you will isolate the local scheduling mechanism: ready sets, completion events, and graph-complexity guards. Distributed ownership and capacity remain in Chapter 22.

20.19 Reference Links

- `bloge-spring` module README — full property tables and quick-start guide
- `BlogeAutoConfiguration.java` — core auto-configuration source
- `BlogeProperties.java` — all `spring.bloge.*` properties
- `BlogeObservabilityAutoConfiguration.java` — metrics, tracing, health indicator
- `BlogeEndpointAutoConfiguration.java` — Actuator endpoints
- `BlogeAuditAutoConfiguration.java` — audit journaling
- `SpringBootTicketTriageApplication.java` — complete Spring Boot example
- `AutoConfiguration.imports` — starter registration file
- Getting Started — setup guide
- Core Architecture — engine internals reference

CHAPTER
21

Arc 5 — Technical Confidence

Chapter 21 — Scheduling and Complexity in One JVM

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

Promise: *You will use a four-node order graph to see how BLOGE discovers ready nodes, runs work concurrently, waits at a join, and protects the feedback loop with complexity limits.*

21.1 Learning goals

By the end of this chapter, you can:

1. Draw how the ready set changes during completion-driven scheduling.
2. Separate inter-graph concurrency, intra-graph concurrency, and shared-resource capacity.
3. Explain why `DependencyScheduler` updates direct descendants instead of rescanning the graph.
4. Reason about node count, depth, fan-out, and branch nesting.
5. Record a reproducible single-machine measurement instead of quoting a naked QPS number.

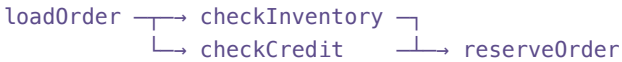
21.2 This chapter adds one variable

An order graph follows dependencies. This chapter narrows the scope to one machine, one JVM, and one `GraphEngine`: **when several nodes are eligible, how does the engine move forward?**

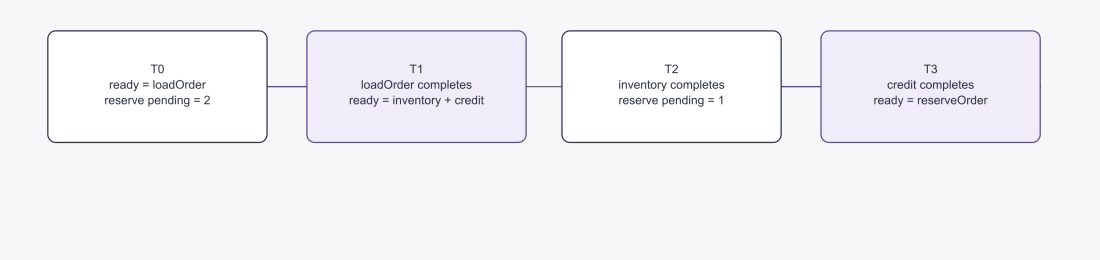
Leases, shards, remote workers, and database capacity stay in the next chapter. Local scheduling must be visible before distributed concerns are added.

21.3 Opening problem: which order step can start

The order flow has four nodes:



After `loadOrder`, the two checks may finish in either order. `reserveOrder` must wait for both. Before reading on, write the ready set for each moment.



21.4 Observe the contract in BLOGE's own tests

Run the RC1 scheduler and complexity tests:

```
cd submodule/bloge
mvn -pl bloge-core \
  -Dtest=SchedulerSequenceTest,GraphComplexityValidatorTest test
```

Observed on 2026-09-14 at commit `cc38fbe5`:

```
GraphComplexityValidatorTest  Tests run: 18, Failures: 0
SchedulerSequenceTest         Tests run: 12, Failures: 0
Tests run: 30, Failures: 0, Errors: 0, Skipped: 0
```

`fanInFromMany_waitsForAll` uses a latch to bring four predecessors into flight before asserting that the sink runs. `multiDiamond_orderingConstraints` asserts only required precedence; it does not demand a fixed order between concurrent nodes.

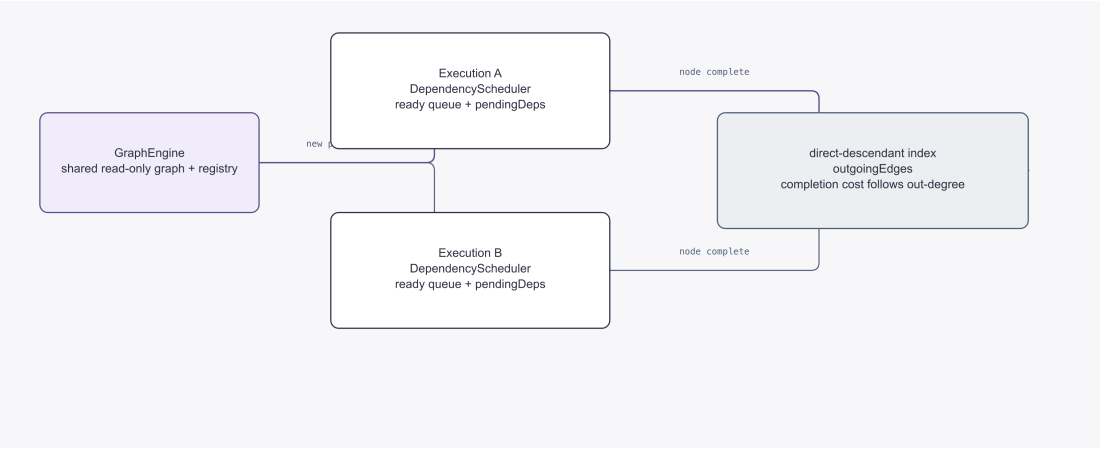
That is the scheduling contract: **dependency order is deterministic; completion order within a ready layer need not be.**

21.5 Mechanism: completion events, not periodic rescans

Each execution owns a `DependencyScheduler`. Initialization records unfinished dependency counts and builds an `outgoingEdges` index:

```
pendingDeps(loadOrder)      = 0
pendingDeps(checkInventory) = 1
pendingDeps(checkCredit)    = 1
pendingDeps(reserveOrder)   = 2
```

Only `loadOrder` enters the first ready queue. Completion follows two outgoing edges and reduces both check nodes to zero. One completed check changes `reserveOrder` from 2 to 1; the second changes it to zero.



The important design is not merely “virtual threads.” Progress cost follows the completed node's out-degree. Rescanning every edge after every completion would repeatedly pay for unrelated work.

21.6 Do not mix three kinds of concurrency

Question	Owner	What this chapter proves
Many orders at once	host caller plus engine	execution state is isolated
Parallel checks in one order	dependency scheduler	eligible nodes may overlap
Database/API load	external resource and capacity config	not proven here

A shared `GraphEngine` does not make downstream pools infinite. A thousand virtual threads waiting on a 20-connection database still have a 20-connection bottleneck.

21.7 Complexity limits protect feedback speed

RC1 checks four dimensions while constructing a graph. `ComplexityLimits.DEFAULT` supplies recommended and hard values:

Dimension	Recommended	Hard	Risk intuition
operator nodes	15	30	larger state and failure surface
DAG depth	6	10	longer critical path
fan-out per node	5	8	burst amplification
branch nesting	2	3	path combinations become opaque

Crossing a recommendation emits a warning; crossing a hard limit rejects the graph. The focused tests cover rejection of 31 nodes, depth 11, and fan-out 9.

21.7.1 Single-factor break: fan-out 8 becomes 9

Keep operators, inputs, and all other edges fixed. Add a ninth child to the root. `GraphComplexityValidatorTest` asserts a construction failure whose message names the graph, root, hard limit 8, and a decomposition hint.

Recovery options, in order:

1. Check whether nine tasks belong in one graph or form a subgraph/batch.
2. Check whether pure projections were incorrectly modeled as operators.
3. If the business really needs the topology, raise limits explicitly and establish a measurement baseline for it.

Raising a limit makes a graph legal. It does not make it fast or understandable.

21.8 Record a trustworthy single-machine measurement

A QPS number needs commit, JDK, CPU quota, heap, graph topology, operator behavior, concurrency, warmup, measurement, fork count, and raw result file. Without them, it is a rumor from one laptop.

Add variables one layer at a time:

```
empty-operator scheduling cost
  → fixed topology under concurrency
    → realistic operators with controlled dependencies
      → metrics and tracing overhead
```

Find which layer changed the result before optimizing. A functional test's elapsed time is not a benchmark, and a JMH mean is not a production capacity promise.

21.9 Real-world transfer: a restaurant dispatch board

A kitchen does not scan every ticket every second. When prep finishes, it notifies only stations that depend on it. Cold and hot dishes may proceed together; plating waits for both.

- The ready queue is the set of station cards that can start now.
- `pendingDeps` is the number of missing prerequisites.
- `outgoingEdges` says whom completion should notify.
- Complexity limits ask an oversized ticket to become courses or batches.

More cooks do not create more oven capacity. In software, thread concurrency is not external-resource capacity.

21.10 Your turn: draw and measure your ready set

Choose a four-to-eight-node graph:

1. Draw dependencies and mark the initial ready set.
2. Pick one legal completion order and update `pendingDeps` frame by frame.
3. Add one edge or child, predict the complexity result, then run it.
4. Create a minimal JMH or repeated-measurement record with environment and topology.

Deliver a four-frame ready-set diagram and one measurement card. Stop when someone else can reproduce the run and the card states that the result applies only to this fixed single-machine environment.

21.11 What this chapter proved—and did not prove

The RC1 tests prove dependency precedence for the covered topologies, all-predecessor fan-in, and default complexity rejection. They do not prove operator thread safety, external API capacity, database headroom, or a reusable production QPS.

The next chapter adds one variable: Chapter 22 — Distributed Runtime and Capacity Validation examines ownership, routing, and bottlenecks when several JVMs share durable state.

21.12 Coding practice

21.12.1 BP-13 — Define stop and resource boundaries

State input limits, business termination, capacity, and budgets separately. A method returning or cancellation being requested does not establish actual worker exit or resource release.

Try one change: Test empty input, the limit, over-limit input, and a condition that never becomes true. Distinguish an Agent budget exit from a valid business result and process-local recovery permits from cluster-wide limits.

Reasoning, counterexamples, and limits

21.13 Experiment acceptance card

- **Expected and observed:** Ready sets and completion events drive single-JVM scheduling; fan-out warns about feedback cost.
- **Failure and recovery:** Raise fan-out from 8 to 9; recover the structure or record acceptance.
- **Proof boundary:** Proves scheduling and a complexity signal, not universal QPS.
- **Exercise contract:** Four-node graph; add one fan-out edge; deliver a timeline and measurement card; stop with a local single-axis conclusion.

21.14 Exact fact entry points

- `CompletionDrivenScheduler.java`
- `DependencyScheduler.java`
- `SchedulerSequenceTest.java`
- `GraphComplexityValidator.java`
- `GraphComplexityValidatorTest.java`
- Durable and benchmark reference

CHAPTER
22

Arc 5 — Technical Confidence

Chapter 22 — Distributed Runtime and Capacity Validation

Testing, observability, Spring wiring, and capacity boundaries

Goal: match each technical claim to evidence that actually supports it.

Promise: *You will place the same order graph behind two workers sharing durable stores, explain why leases need fencing and routes need stable bindings, then produce a capacity claim that applies only to an explicit workload and environment.*

22.1 Learning goals

By the end of this chapter, you can:

1. Explain how lease owner, timeout, and epoch make a takeover safe.
2. Separate stable execution routing from a shard resolver's new choice.
3. Locate the resource that saturates before adding workers.
4. Design a minimum capacity experiment around arrival rate, service time, concurrency, and backlog.
5. State the distributed contract a test proves and its external boundary.

22.2 This chapter adds one variable

The previous chapter fixed the world inside one JVM. The order graph is unchanged, but Worker A and Worker B now share durable stores. One variable is new: **process memory no longer guarantees execution ownership.**

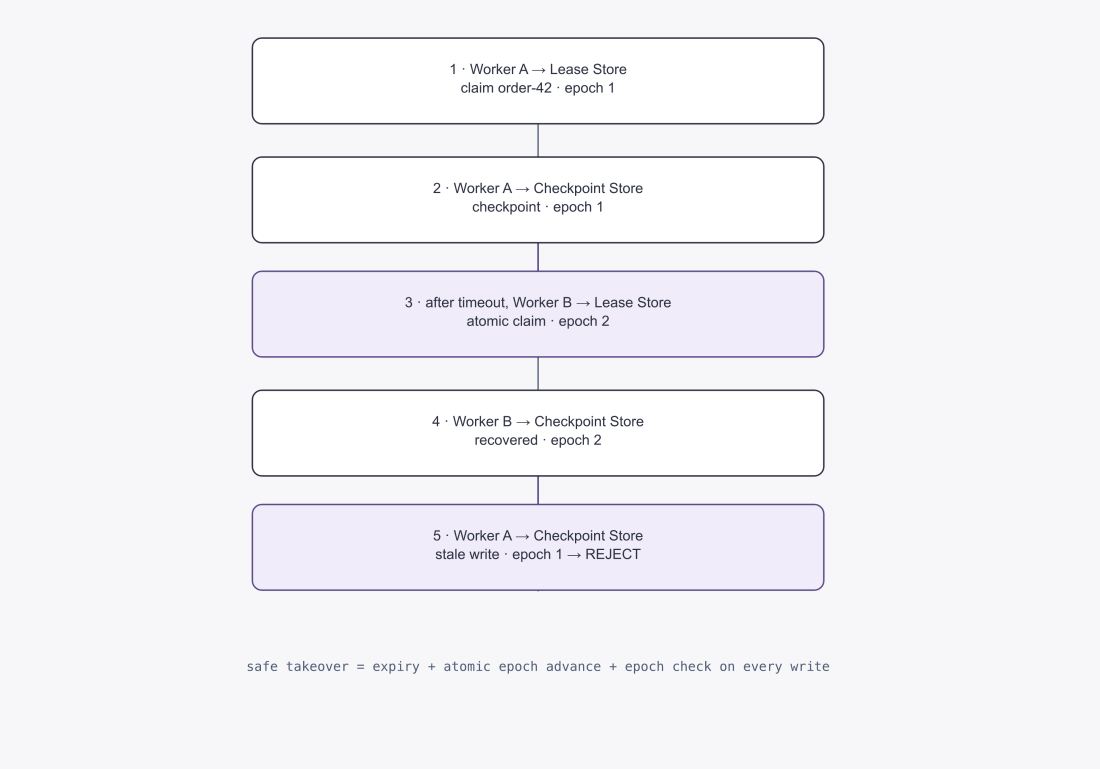
Two hazards follow: A writes after being considered dead; the same business key drifts to another shard because routing policy changed.

22.3 Opening problem: can B continue as soon as A times out

Predict whether this timeline is safe:

```
10:00:00  Worker A claims order-42, epoch=1
10:00:20  A writes a checkpoint
10:01:01  lease expires; Worker B claims, epoch=2
10:01:03  A's network returns; A writes again
```

If a store checks only an owner name, A can overwrite B's newer state. Safe takeover needs both timeout and a monotonically increasing fencing epoch.



22.4 Run two RC1 distributed-contract tests

```
cd submodule/bloge
mvn -pl bloge-durable \
  -Dtest=ClusterFailoverSimulationTest,RuntimeRoutingShardingTest test
```

Observed on 2026-09-14 at commit `cc38fbe5`:

```
RuntimeRoutingShardingTest    Tests run: 2, Failures: 0
ClusterFailoverSimulationTest Tests run: 1, Failures: 0
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

These tests use a logical clock and in-memory stores. They establish deterministic contracts, not database throughput.

22.5 Mechanism one: takeover is a three-condition chain

`ClusterFailoverSimulationTest` performs four actions:

1. A claims an execution at epoch 1 and writes a checkpoint in that lease context.
2. Logical time moves beyond the one-minute timeout.
3. B's recovery loop reclaims at epoch 2, writes a recovery marker, and completes the execution.
4. A tries to write with epoch 1; the store throws `StaleFencingEpochException`.

Safe takeover requires all three conditions:

```
lease expired
  + new owner atomically claims and advances epoch
  + every protected write validates current epoch
```

Without the third, a zombie worker can corrupt state. Without a recovery scan, fencing alone leaves the execution stranded.

22.5.1 Single-factor break: accept a stale epoch

Keep timeout, checkpoints, and recovery unchanged; only allow A's final epoch-1 write. Final status may still say `COMPLETED`, while the old owner overwrites the recovery marker. Equal final status does not prove a correct takeover.

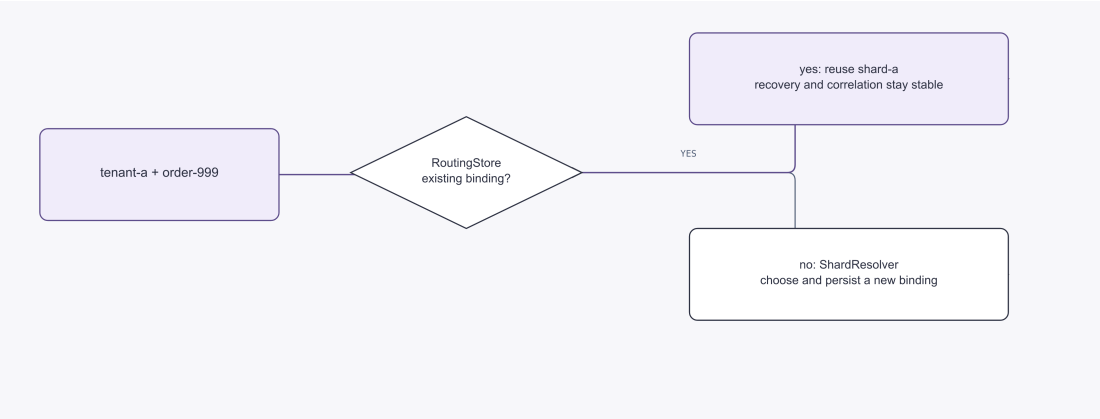
Recovery means carrying lease context on every protected mutation and failing closed in persistence. Logs cannot repair the conflict afterward.

22.6 Mechanism two: routing is a stable business-key binding

`RuntimeRoutingShardingTest` first persists:

`tenant-a + order-999 → shard-a`

Even when a second engine's resolver returns `shard-b`, the existing binding wins and the second execution stays on `shard-a`.



The resolver answers "where should an unbound key go?" The routing store records "where does this business object already live?" Rehashing on every request can lose recovery and event correlation after the shard count changes.

Migration must be an explicit control-plane operation: freeze the key, move state, update the binding, verify reads and writes, then resume traffic. It is not ordinary resolver recomputation.

22.7 Capacity: find the bottleneck before scaling it

Think of throughput as a narrow pipe:

```
arrival rate
  → runnable worker concurrency
    → external API quota
      → connection pool
        → durable-store writes and lock contention
```

Adding workers widens only the second segment. If checkpoint latency is saturated, more workers add queueing and lock pressure.

Little's Law gives a first estimate: in a stable system, `in-flight ≈ arrival rate × average service time`. It helps design measurement; it is not a BLOGE capacity guarantee.

Measure	Meaning
arrival rate	new executions or work items per second
service time	distribution from runnable to complete
in-flight	work currently holding workers/resources
backlog age	age of the oldest waiting item
store latency	p50/p95/p99 of checkpoint, claim, and poll

22.8 An attributable scaling experiment

Fix graph, input distribution, database, connection pool, and external stubs. Change only worker count from 1 to 2, 4, and 8. Record throughput, p95, backlog age, and store latency.

- Throughput grows while store latency stays flat: workers may be the current bottleneck.
- Throughput stays flat while store latency rises: store or lock contention is more likely.
- Backlog falls while downstream 429s rise: pressure was merely moved.

Change one axis per experiment or the worker-count effect is not attributable.

22.9 A remote worker is a dispatch boundary, not free capacity

`WorkerDispatcher` sends selected work items to a remote pool; the Kafka module provides one transport. Dispatch adds serialization, delivery, duplicates, heartbeat, result correlation, and cancellation semantics. A GPU worker may solve hardware affinity but does not create exactly-once effects.

The main chapter keeps only that boundary. Topic, heartbeat, batch-size, schema-migration status, and benchmark parameters live in the runtime reference so architectural judgment is not buried in a configuration catalog.

22.10 Real-world transfer: takeover in a warehouse network

Warehouse A scans a parcel and loses connectivity. Warehouse B must not ship merely because A has been silent for a minute. Headquarters issues a newer work token and rejects A's old token. An order already bound to the east warehouse must not drift just because a south warehouse was added today.

Those are fencing epoch and routing binding. More warehouses help only when picking stations are the bottleneck. If the central inventory lock is saturated, more sites increase contention.

22.11 Your turn: deliver a capacity conclusion, not a chart

Choose one durable graph and define:

1. One fixed input distribution and target arrival rate.
2. Worker counts 1, 2, and 4 with all other conditions fixed.
3. Throughput, p95, backlog age, and store p95.
4. One worker interruption and one stale-epoch write attempt.
5. A stop rule, such as backlog age rising for three consecutive windows.

Deliver raw results, an environment card, a single-axis comparison, and one sentence: “Under these conditions, this resource saturated first.” Do not write “production traffic is supported” unless the production traffic model and external dependencies are inside the evidence boundary.

22.12 What this chapter proved—and did not prove

The focused tests prove that, in in-memory simulations, a new owner reclaims an expired execution at a higher epoch, stale writes are rejected, and an existing business-key binding beats a new resolver choice. They do not prove real-database isolation, Kafka delivery semantics, cross-region recovery, or production capacity.

The next stage begins with Chapter 23 — Business Correctness Is Not Test Pass: it moves from bounded runtime evidence to asking whether the declared business promise is satisfied.

22.13 Experiment acceptance card

- **Expected and observed:** Takeover requires matching lease, epoch, and route while the business key stays bound.
- **Failure and recovery:** Use a stale epoch; recover the new lease/epoch and reject the old owner.
- **Proof boundary:** Proves in-memory takeover and routing, not database isolation or capacity.
- **Exercise contract:** One failover; change only epoch; deliver takeover and route receipts; stop when stale writes fail and one owner remains.

22.14 Exact fact entry points

- `ClusterFailoverSimulationTest.java`
- `RuntimeRoutingShardingTest.java`
- `ExecutionLeaseContext.java`
- `StoreRouter.java`
- `bloge-dispatch-kafka`
- Durable and scale reference

Arc 5 Recap — Technical Confidence (Chapters 18–22)

CHAPTER

—

Arc 5 Recap — Technical Confidence (Chapters 18–22)

You can now state exactly what your tests, telemetry, wiring, and capacity evidence prove—and where their boundary ends.

0.1 The Confidence Stack

Chapter	Question answered	Evidence produced
18 — Testing Your Graphs	Did the chosen behavior occur?	Operator, graph, and DSL checks with controlled time
19 — Observability in Production	Can one real execution be explained?	One incident reconstructed through four projections
20 — Spring and Production Wiring	Who creates and overrides runtime components?	Minimal auto-configuration plus one named override
21 — Scheduling and Complexity in One JVM	How do ready sets and graph complexity shape local work?	Scheduling contract tests plus a local measurement protocol
22 — Distributed Runtime and Capacity Validation	What makes a cluster claim credible?	Routing, lease, store, recovery, and migration boundaries

The chapters form a ladder. A green graph test does not prove production explainability; a trace does not prove business correctness; one-host throughput does not prove horizontal scale.

0.2 What You Should Be Able to Do Now

- Separate operator, graph, DSL, and time-dependent tests by the claim each supports.
- Reconstruct one incident from lifecycle, business, trace, and capacity views.
- Auto-configure a Spring application and override one component with one owner.
- Tune complexity limits and report a reproducible single-JVM measurement from your recorded environment.
- Refuse a distributed-capacity claim until routing, lease, shared-store, recovery, and migration evidence exists.

0.3 Common Claim Errors

Claim	Hidden gap	Better statement
“The workflow is correct.”	Only terminal output was checked	“This scenario produced the expected node states and output.”
“Production incidents are debuggable.”	Logs cannot be joined into one execution	“These four projections share one execution identity.”
“Spring manages everything.”	A manual bean silently replaced auto-configuration	“This named override has one explicit owner.”
“BLOGE scales horizontally.”	Benchmark ran on one host	“One JVM sustained X under the recorded workload.”

0.4 Try It: Reject an Overclaim

A benchmark reaches 20,000 graph runs per second on one 32-core host. The team writes “BLOGE scales horizontally.” List the missing evidence, then rewrite the claim so it is exactly supported. Separate scheduler throughput from routing, lease ownership, shared-store recovery, and migration.

0.5 Where to Go Next

Arc 6 changes the question from “did the runtime execute?” to “did the business promise hold, and what evidence may enter a release decision?” Continue with Chapter 23 — Business Correctness Is Not Test Pass. For production configuration details, use Appendix I.

Learning Arc

Arc 6 — Business Promises

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

CHAPTER
23

Arc 6 — Business Promises

Chapter 23 — Business Correctness Is Not Test Pass

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

Promise: *You will separate "the code runs," "the graph succeeds," "the business result satisfies the contract," and "the evidence can support release," then see why* `PASS` *cannot be rewritten as "safe to release."*

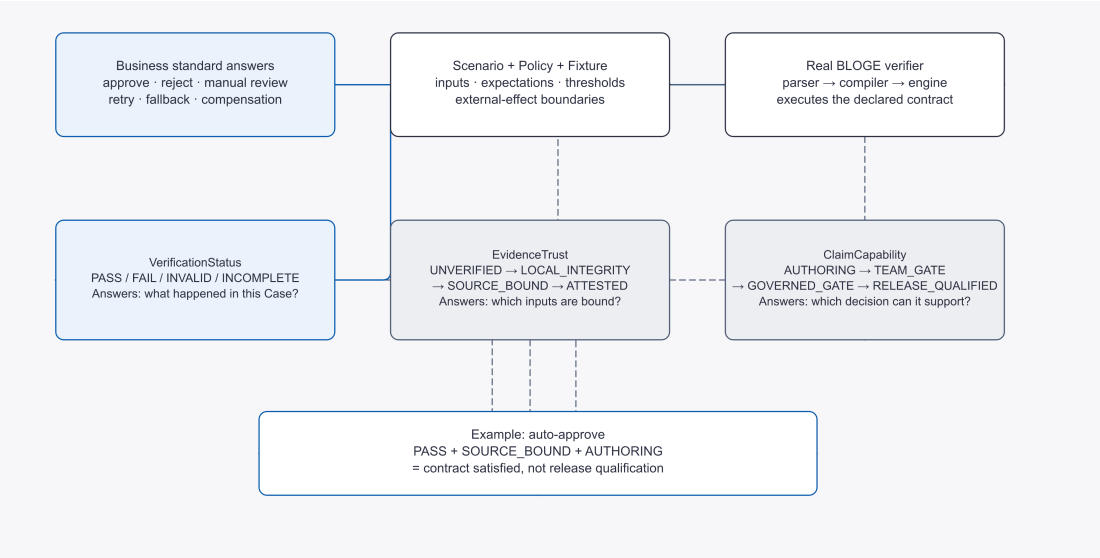
23.1 Learning Goals

1. Distinguish syntax, execution, business, and evidence correctness.
2. Build three independent axes: `VerificationStatus`, `EvidenceTrust`, and `ClaimCapability`.
3. Identify the boundary between the business owner, BLOGE developer, verifier, and release owner.
4. Run a real BLOGE verifier with a minimal Scenario instead of comparing only the final `GraphResult`.

23.2 Start with the business question

The loan operations owner gives a reviewable standard: low-risk applications are auto-approved, high-risk applications are auto-rejected, boundary applications enter manual review, and funding failures follow the policy for retry, fallback, or compensation.

This is not one "tests pass" statement. Start with the relationship map:



The point of the diagram is that the three axes do not substitute for one another:

`VerificationStatus` — answers what happened in this Suite / Case. It does not prove that the evidence is bound to source and runtime inputs.

`EvidenceTrust` — answers which source, Scenario, Policy, Fixture, and runtime are bound. It does not prove that the business policy itself is correct.

`ClaimCapability` — answers whether the evidence supports authoring, team, governed, or release decisions. It does not prove that the business verdict is `PASS`.

23.3 Separate the four kinds of correctness

Chapter 18 already teaches Graph tests for node status, execution order, and `GraphResult`. Those tools remain useful, but they cover only part of the complete question.

Syntax correctness — Can the `.bloge` file parse and compile? Use the parser, compiler, and DSL test. This does not prove that the business branch is right.

Execution correctness — Did nodes follow dependencies and propagate errors? Use `GraphTestRunner` and `GraphResult`. This does not prove the business standard.

Business correctness — Is `riskDecision` really `auto_approve`? Use a `Scenario` with `BUSINESS_CONTRACT`. This does not prove source and runtime identity.

Evidence usability — Can others use the result for a team or release decision? Use a source-bound reader and `VerificationClaims`. This does not approve the business policy.

`GraphResult.isSuccess()` means that the graph did not fail under engine semantics. It does not automatically show that a rejection branch was not bypassed or that an external effect matches the production contract.

23.4 The first verifier entry point

`ScenarioVerifier.verify(Path)` is the shortest single-Suite entry point. The starter uses `BlogeScenarioTests.fromDirectory(...)` to turn a Scenario directory into JUnit dynamic tests; both paths use the real parser, compiler, and engine.

The snippet shows the entry shape only. Replace `bootstrap`, `projectRoot`, and Policy with project facts:

```
ScenarioVerifier verifier = ScenarioVerifier.builder()
        .bootstrap(caseContext -> VerificationEnvironment.controlled(operatorRegistry))
        .projectRoot(projectRoot)
        .policy(projectRoot.resolve("src/test/bloge/verification-policy.yaml"))
        .build();
```

```
VerificationReport report = verifier.verify(
        projectRoot.resolve("src/test/bloge/scenarios/loan-approval.scenario.yaml"));

assertEquals(VerificationStatus.PASS, report.status());
```

This `PASS` means that the declared Suite satisfied its Expectations in this run. Without source-bound artifacts, a receipt, and an independent reader, do not silently upgrade it to `SOURCE_BOUND` or `RELEASE_QUALIFIED`.

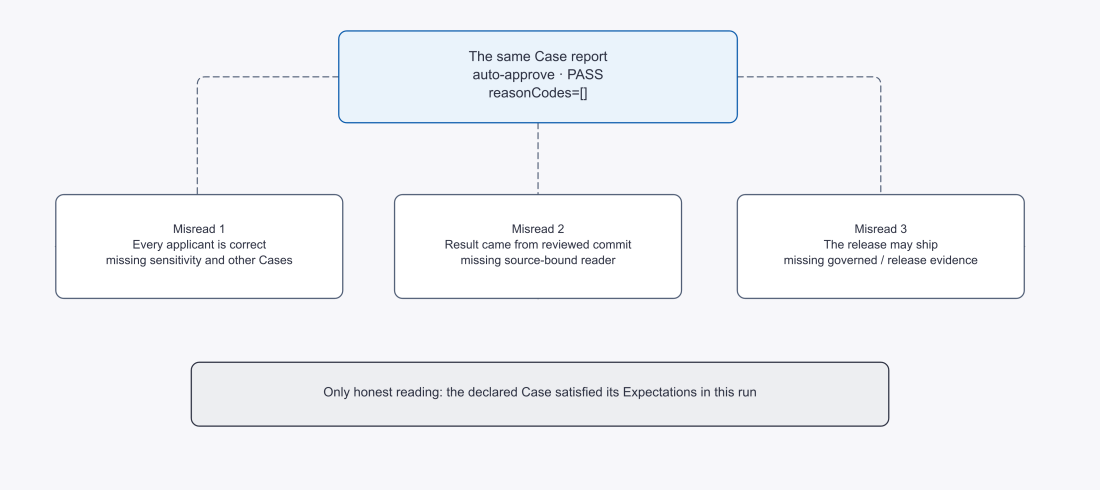
23.5 Read one PASS three ways—and reject all three

Suppose the loan report contains only these facts:

```
suiteId=loan-approval-decisions
caseId=auto-approve
status=PASS
reasonCodes=[]
```

Three readers can turn that honest report into three dishonest claims:

Misreading	What is missing	Correct reading
"The decision rule is correct for every applicant."	Cases outside this Suite and a sensitivity challenge	This Case satisfied its declared Expectations.
"The result came from the reviewed commit."	seal, source digest, worktree identity, and reader result	The report object alone is `UNVERIFIED`.
"The release can ship."	policy continuity, governed receipts, comparison, and release attestation	No release capability follows from `PASS` alone.



This is not philosophical caution; RC1 makes the axes disagree on purpose. `VerificationReportWriterTest.rechecksArtifactBytesAfterThePublicationGuard` starts with a `PASS` report, then lets the publication guard push the summary above 4096 bytes. The writer returns `INCOMPLETE / RESOURCE_LIMIT_EXCEEDED` and leaves the output directory empty. A business verdict cannot override a failed publication boundary.

`VerificationContractEvidenceTest` adds the opposite guard: a `PASS` Case cannot carry failure reasons, a `FAIL` Case cannot claim satisfaction, and aggregate identities must match the Suite, Case, Requirement, and assurance records. On 2026-09-14 at BLOGE commit `cc38fbe5`, these focused classes produced 5 tests, 0 failures, 0 errors, and 0 skips.

The practical reading order is fixed:

1. Read `VerificationStatus`: did the declared contract match this run?
2. Read `EvidenceTrust`: what immutable inputs does the evidence bind?
3. Read `ClaimCapability`: which decision is the complete evidence allowed to support?

Stopping after step 1 is fine during authoring. Renaming step 1 as step 3 is not.

23.6 Three common false greens

Skipped `manual_review` — an ordinary test may see a final object that is still `approved`. Business verification must assert business output and the human path separately; skipped nodes are structural facts.

Hidden fallback — an ordinary test may see a Graph that is still `SUCCESS`. Business verification must record retry disposition, fallback, and effect handling, then confirm that the degradation is allowed.

Direct Operator call — an ordinary test may see a `PASS` from the Operator unit test. Business verification must run the Scenario through the real parser/compiler/engine and restrict undeclared effects.

This does not make JUnit useless. Each layer should answer the question it is good at, and the results should be composed within their evidence boundaries.

23.7 What each role provides

Business owner — provides standard answers, boundary Cases, Requirements, acceptable failures, and Oracle provenance. Does not turn a green test directly into release approval.

BLOGE developer — provides the DSL, Operators, Scenario, Policy, Fixture, and bootstrap. Does not copy implementation code as the standard answer.

Verifier / platform — provides controlled execution, observations, reason codes, artifacts, seal, and reader. Does not change business policy to remove a failure.

Release owner — provides the commit, worktree, receipt, ClaimCapability, and gaps. Does not rename `AUTHORING` as `RELEASE_QUALIFIED`.

23.8 Lab: split one green test apart

1. Pick a loan Case that asserts only `GraphResult` and add one `BUSINESS_OUTPUT` assertion.
2. Add one `STRUCTURAL_CONTRACT` assertion for a branch skip or invocation fact.
3. List two facts still unproven, such as missing source binding or missing independent external-system evidence.
4. Record status, evidence trust, and claim capability as three columns; one word must not stand in for all three.

Stop when the input, Policy, or bootstrap is incomplete. The report should remain `INVALID` or `INCOMPLETE`; do not retry a contract problem until it becomes a green light.

23.9 Transfer: a medication-release workflow

In a hospital pharmacy, “the workflow completed” means the prescription moved through its declared steps. It does not mean the dose satisfies the approved medication rule, that the result came from the reviewed formulary, or that a pharmacist may release it. Map graph status, dosage Expectation, source-bound formulary evidence, and release authority onto the same four layers used here. Stop if one green badge is doing more than one of those jobs.

23.10 Experiment acceptance card

- **Expected and observed:** Read one PASS separately as verdict, evidence trust, and claim capability.
- **Failure and recovery:** Remove a guard or corrupt evidence; restore it and reread all three axes.
- **Proof boundary:** Proves a controlled case claim, not release approval.
- **Exercise contract:** One green case; attack one evidence boundary; deliver three readings; stop when no conclusion exceeds capability.

23.11 Chapter summary

A reliable `FAIL` can show that a business contract was not satisfied. An unbound `PASS` only shows that one run looked successful. The next chapter applies this rule to two Suites and seven Cases in the starter.

Next: Chapter 24 — Your First Loan Business Verification

CHAPTER
24

Arc 6 — Business Promises

Chapter 24 — Your First Loan Business Verification

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

Promise: *You will turn one business-owner-approved loan decision into an executable Scenario, trace every Expectation through the real graph, and read the verification evidence produced by BLOGE `0.9.8-RC1`. You will also know where verification must stop when the business answer is missing.*

24.1 Learning goals

By the end of this chapter, you can:

1. Separate "the program completed" from "the business result is correct."
2. Express a rule as `given / when / then`, with `then` pointing to observable business output or structural fact.
3. Trace one evidence chain across Scenario, graph, output mapping, Policy, and Report.
4. Cover decisions, manual review, retry, fallback, and compensation with a small set of representative Cases.
5. Stop automation when the business oracle is missing instead of letting engineering invent policy.

24.2 This chapter adds one variable

Earlier chapters mostly asked, "Can the graph run as designed?" This chapter adds one question: **Is the business answer produced by that run correct?**

Those are different verdicts. Every node may complete without an exception while a customer who requires manual review is automatically approved. The runtime says execution succeeded; the policy owner says the decision is wrong.

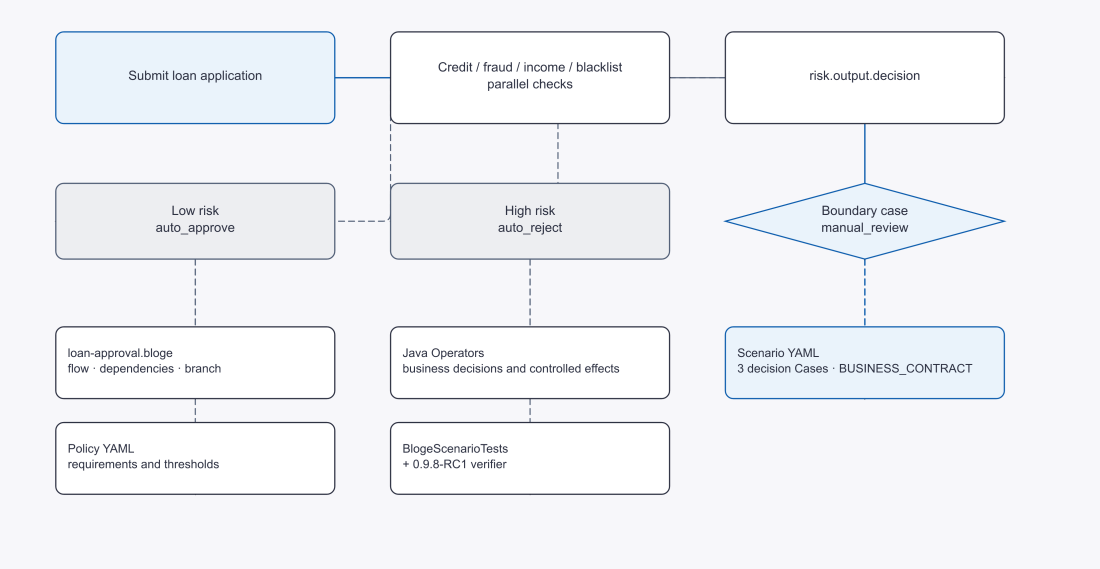
Business verification is therefore not a synonym for more unit tests. It connects an approved answer, a real graph path, and reviewable evidence.

24.3 Predict first: which path should a low-risk application take?

The loan policy owner approves one representative example: application `LA-VERIFY-1` is low risk and should be automatically approved. Before reading the test, predict the observable facts:

Observation	Prediction	Why HTTP 200 is insufficient
Risk decision	`auto_approve`	Proves the routing decision
Approval status	`approved`	Proves the approval branch produced a business result
Reject node	`SKIPPED`	Proves the mutually exclusive branch did not run

The third fact catches a structural defect in which approve and reject both execute even if one final field still reads `approved`.



24.4 Five assets, five questions

An auditable chain uses five small assets with distinct responsibilities:

Asset	Question answered	This chapter's instance
Business oracle	What result counts as correct?	Low-risk applications auto-approve
Scenario	Which input, action, and observations?	`loan-approval.scenario.yaml`
Graph	Which nodes and branches actually run?	`loan-approval.bloge`
Policy	Which Cases and evidence depth are mandatory?	`verification-policy.yaml`
Report	What did this particular run observe?	`target/bloge-verification/`

The business owner supplies the answer, engineering translates it into executable observations, and BLOGE runs the graph and preserves the evidence.

24.5 Translate one rule into Given / When / Then

The real Scenario declares ownership, graph source, and Case identity before its three-part contract. This excerpt keeps only `auto-approve`:

```yaml
- caseId: auto-approve
  title: Low-risk application is auto-approved
  requirementRefs: [LOAN-DECISION-001]
  oracle:
    type: APPROVED_EXAMPLE
    sourceRef: starter-policy:auto-approve
```

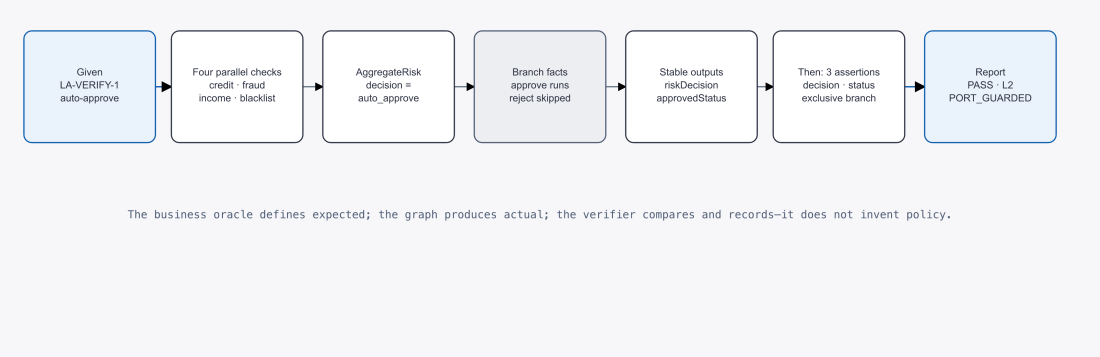
```
    owner: loan-policy-owner
  given:
    context:
      applicationId: LA-VERIFY-1
      caseName: auto-approve
  when:
    action: EXECUTE
  then:
    outcome: SUCCESS
    expectations:
      - actual: {kind: BUSINESS_OUTPUT, outputId: riskDecision}
        matcher: EQUALS
        expected: auto_approve
      - actual: {kind: BUSINESS_OUTPUT, outputId: approvedStatus}
        matcher: EQUALS
        expected: approved
      - actual: {kind: NODE_STATUS, nodeId: reject}
        matcher: EQUALS
        expected: SKIPPED
```

Do not memorize YAML first. Ask three business questions:

- `given`: Which real-world application shape does this Case describe?
- `when`: What should the system do? Here it executes the real graph.
- `then`: Which observations are enough for the owner to say the answer is right?

`oracle` identifies the source, type, and owner of the expected answer. `APPROVED_EXAMPLE` means an owner-approved representative case, not a policy inferred by BLOGE.

24.6 Trace the graph: where do Expectations get facts?



The Case follows this causal chain:

1. `FetchLoanApplication` constructs deterministic application data from `caseName=auto-approve`.
2. `credit`, `fraud`, `income`, and `blacklist` read the application and emit independent risk facts.
3. `AggregateRisk` combines them and emits `decision=auto_approve`.
4. The graph enters only `approve`; `reject` and manual review must not run.
5. `outputs` map node internals to stable business outputs such as `riskDecision` and `approvedStatus`; the verifier compares expected and actual values.

That is why the Scenario uses two kinds of assertion:

- `BUSINESS_OUTPUT` checks a business result such as approval status.
- `NODE_STATUS` checks execution structure such as the reject branch being skipped.

Stable business outputs can survive internal graph refactoring. Reach into structure only when mutual exclusion, retry, fallback, or compensation is itself part of the contract.

24.7 Run real verification

From the book root:

```
cd submodule/bloge
mvn -pl bloge-starter-loan-approval -am \
  -Dtest='LoanApprovalApplicationTest,LoanApprovalBusinessScenarioTest,LoanApprovalOperatorFlowTest' \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

On 2026-09-14, BLOGE commit `cc38fbe5`, version `0.9.8-RC1`, produced:

```
LoanApprovalApplicationTest       1 passed
LoanApprovalOperatorFlowTest      3 passed
LoanApprovalBusinessScenarioTest  7 passed
Total                             11 passed, 0 failed
```

Those seven dynamic Scenarios are three decision rules and four funding-resilience rules, not seven method-return tests. The report summary adds:

```
Suite: loan-approval-decisions
Status: PASS
Cases: 3/3 passed
Executed nodes: 10/10
auto-approve depth=L2 effectSafety=PORT_GUARDED replay=CONTRACT_CONTROLLED
Requirement LOAN-DECISION-001 satisfied=true cases=1/1
```

The machine-readable result for `auto-approve` is:

```
{
  "caseId": "auto-approve",
  "effectSafety": "PORT_GUARDED",
  "realEffectPortIds": [],
  "reasonCodes": [],
  "replayAssurance": "CONTRACT_CONTROLLED",
  "status": "PASS",
  "unverifiedOperatorRefs": [],
  "verificationDepth": "L2"
}
```

Read the labels as bounded facts, not slogans:

- `L2` means graph-Scenario execution evidence, not production acceptance.
- `PORT_GUARDED` says effect ports were controlled at the verification boundary; empty `realEffectPortIds` means no real external effect port was reached.
- `CONTRACT_CONTROLLED` describes contract-controlled replay conditions, not universal production reproducibility.
- Empty `reasonCodes` and `unverifiedOperatorRefs` mean no additional degradation reason or unverified Operator reference was recorded.

24.8 Expand one Case into a decision coverage map

One auto-approval Case supports the declared assertions for one policy region. This decision Suite samples three mutually exclusive regions:

Case	Business shape	Required result	Result that must be excluded
`auto-approve`	Low risk	Automatic approval	Reject branch skipped
`auto-reject`	High risk	Automatic rejection	Approve branch skipped

Case	Business shape	Required result	Result that must be excluded
`manual-review`	Boundary risk or facts needing review	Human review with final status	Automatic approve skipped

This is not case-count theatre. Removing one Case leaves one policy region without an approved answer.

24.9 Funding adds a harder question: what should remain after failure?

The funding Suite progresses from choosing a branch to validating the business world after a failure:

Case	Real-world mapping	Key evidence
`funding-success`	Fund and notify normally	`fundingStatus=notified`
`transient-booking-retried`	Ledger service briefly fails	First attempt schedules retry; second succeeds
`booking-fallback-used`	Retry exhausts and approved fallback applies	Third invocation records `FALLBACK_APPLIED`
`notification-failure-compensated`	Funds booked, then notification fails terminally	Graph fails; booking reverses; funds release

The last Case has a deliberately non-successful graph outcome. The correct business behavior is to fail explicitly and compensate reversible effects. A Scenario with `then.outcome: FAILURE` can itself pass verification.

24.10 Stop when the business answer is missing

Suppose a product manager says, “Manual review can be more lenient for high-net-worth customers.” This is not an executable oracle. It does not define:

- how high-net-worth is determined;
- which risk dimensions can move, and by how much;
- whether the final answer is approve, reject, or human discretion;
- who owns that standard.

Do not guess an `expected` value or infer correctness from the current code. Mark the Scenario as awaiting clarification, record the missing rule and owner, and stop the release-gate decision.

> ***Stop rule:*** *Without a traceable business oracle, you may verify what the system did, but not claim the business result is correct.*

24.11 Do not guess when the Maven entry point fails

The RC1 Maven verification entry point requires a usable test classpath. If it cannot build one, it reports:

```
TEST_CLASSPATH_UNAVAILABLE: cannot execute BLOGE verification
```

That is an environment/toolchain failure, not a failed business Case. Repair dependency or reactor scope, then rerun. Never report PASS because an old report directory exists when verification did not execute.

24.12 What this chapter proves—and does not prove

The evidence supports these claims:

- Three loan decisions and four funding Scenarios pass at the pinned commit.
- `auto-approve` leaves machine-readable business outputs, branch exclusion, and Requirement mapping.

- No real external effect port was reached; replay conditions were contract-controlled.

It does not prove:

- every possible input combination is correct;
- the policy itself is wise or legally compliant;
- production databases, brokers, and external services are accepted;
- a future version will produce the same result.

Business verification does not stamp a system "correct forever." It makes one conclusion answerable: **who defined the answer, which input ran, which path executed, what was observed, and which version made it true.**

24.13 Transfer the method to another industry

For an ecommerce refund:

- `given`: order delivered three days ago, unopened product, amount 199;
- `when`: execute the refund-eligibility graph;
- `then`: `refundDecision=auto_approve`, refund branch runs, human-review branch is skipped;
- `oracle`: a refund-policy example approved by the after-sales owner.

Vocabulary and answers change; the chain remains: business oracle → Scenario → graph → observations → Report.

24.14 Lab: change one variable

Copy `auto-approve` and change only `caseName` to `manual-review`. Predict three facts before comparing the repository's official Case:

1. What is `riskDecision`?
2. Which final status appears?
3. Which automatic branch must be `SKIPPED`?

If your answer conflicts with the owner's approved example, do not edit `expected` merely to get green. Determine whether implementation is wrong, the example is stale, or policy changed.

24.15 Coding practice

24.15.1 BP-14 — Test independent answers and forbidden actions

Use approved expected answers, not current outputs. Observe business results, node paths, and effect evidence independently; mock only the boundary whose behavior is intentionally controlled.

Try one change: Run the workshop blacklist-priority case and its reordered-rule mutant. Explain why preserving old behavior, rejecting one mutant, and proving all policies correct are different claims.

Reasoning, counterexamples, and limits

24.16 Experiment acceptance card

- **Expected and observed:** The low-risk loan satisfies Given/When/Then and reports 3/3 passed.
- **Failure and recovery:** Remove the golden answer or fail the plugin; restore approved facts or the build, and stop on missing facts.
- **Proof boundary:** Proves a fixed case and graph contract, not complete policy or signed release.
- **Exercise contract:** Auto-approve case; change one field; deliver path, expectation, and report; stop when attributable without guessing.

24.17 Exact fact entry points

- Loan decision graph
- Loan decision Scenario
- Funding resilience Scenario
- Verification Policy
- Dynamic Scenario adapter

The next chapter, Chapter 25 — Policy, Fixture, and Assurance, asks whether a passing Suite was executed under the intended action, dependency, and assurance boundaries.

CHAPTER
25

Arc 6 — Business Promises

Chapter 25 — Policy, Fixture, and Assurance

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

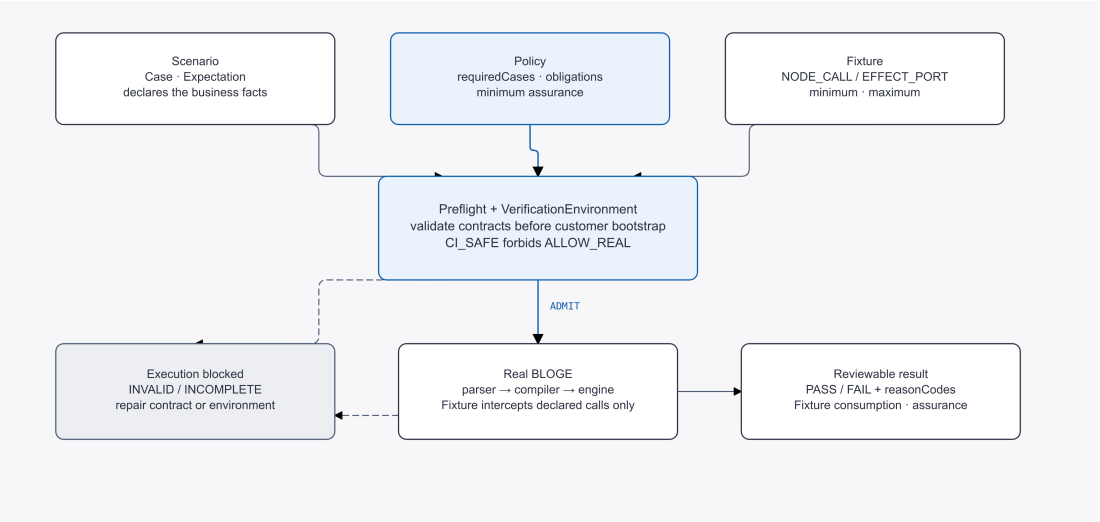
Chapter promise: *You will separate what must be verified, how external behavior is controlled, and what minimum evidence is required. Removing a protected Case, exceeding Fixture consumption, or allowing a real effect will stop at the verification boundary instead of producing a false green result.*

25.1 Learning goals

1. Separate the responsibilities of Scenario, Policy, Fixture, and `VerificationExecutionProfile`.
2. Choose the next action from `FAIL`, `INVALID`, or `INCOMPLETE`.
3. Use `verifyGoverned(Path)` for Requirements and `verifyFixtureFidelity(Path)` for Fixture contracts.
4. Reproduce three one-variable failures and diagnose the earliest reason code.

25.2 Three declarations guard one execution entrance

A team can make the loan tests green by deleting `manual-review`, widening a Fixture until every request succeeds, or letting an undeclared effect reach the real ledger. None of those changes edits an assertion, yet each shrinks or contaminates the proof.



The orange entrance is preflight. Scenario, Policy, and Fixture must agree before the verifier creates customer bootstrap state and enters the real BLOGE parser, compiler, and engine. Rejected configuration does not execute customer code first and append a warning later.

Scenario declares a Case input, action, and Expectations: what this run will observe.

Policy protects required Suites, Cases, Requirements, and minimum assurance: what cannot silently disappear.

Fixture names the node call or effect port it controls, its behavior, and consumption range: how external behavior is bounded.

`VerificationExecutionProfile` fixes logical time, random and ID seeds, harness timeout, and persistence mode. Declared host capabilities are bound inputs, not isolation evidence.

25.3 Protect the manual-review Case first

The starter Policy protects all three decision Cases:

```
requiredSuites:
  - suiteId: loan-approval-decisions
    requiredCases: [auto-approve, auto-reject, manual-review]
    minimumDepth: L2
    minimumBusinessExpectations: 6
    minimumReplayAssurance: CONTRACT_CONTROLLED
maximumUnverifiedEffects: 0
```

Run the governed entry point:

```
VerificationGovernedReport report = ScenarioVerifier.builder()
        .bootstrap(caseContext -> loanVerificationEnvironment(caseContext))
        .projectRoot(projectRoot)
        .policy(projectRoot.resolve("src/test/bloge/verification-policy.yaml"))
        .build()
        .verifyGoverned(projectRoot.resolve(
                "src/test/bloge/scenarios/loan-approval.scenario.yaml"));
```

`requiredCases` prevents a Case from disappearing. Requirement satisfaction also depends on `requirementObligations` and `requirementBindings`: only passing `BUSINESS_CONTRACT` Expectations contribute Requirement evidence. Merely listing a `requirementRef` does not satisfy it.

25.4 A Fixture is not a global mock

Use `NODE_CALL` to replace an Operator invocation. Use `EFFECT_PORT` to retain the Operator's calculation and control only its external access. The target types are not interchangeable.

```yaml
schemaVersion: 1
fixtureSetId: funding-effect
rules:
  - ruleId: book-once
    target:
      type: EFFECT_PORT
      portId: ledger.book
      parentNodeId: bookDisbursement
      inputEquals: {applicationId: LA-FUND-1}
    behavior:
      type: RETURN
      value: {status: booked}
    consumption: {minimum: 1, maximum: 1}
```

Zero matches, ambiguous matches, under-consumption, over-consumption, and unbound effects fail closed. `CI_SAFE` is the only current profile and rejects `ALLOW_REAL`. Effect `DELAY` uses the bootstrap `TimeSource`; the Fixture does not wait on wall-clock time with `Thread.sleep`.

25.5 Keep assurance separate from fidelity

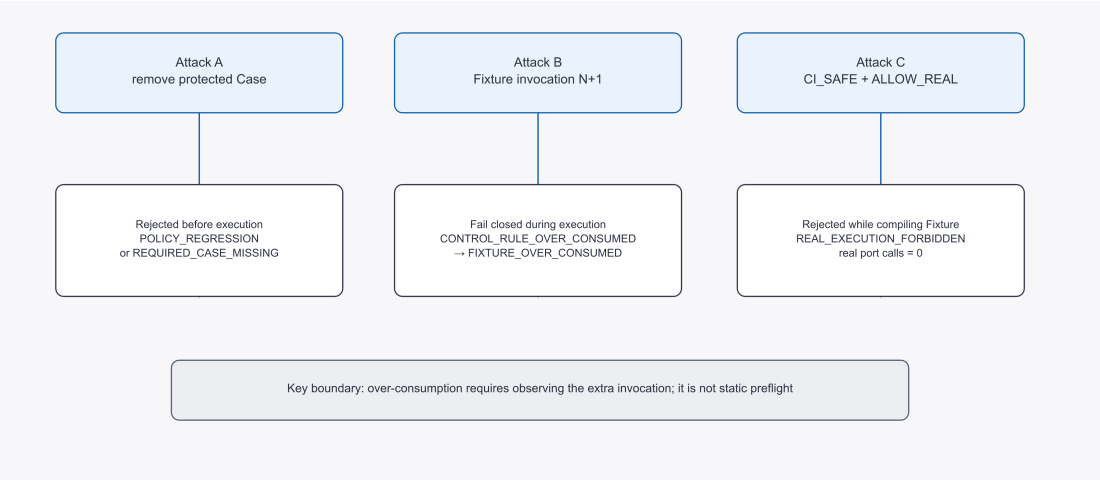
A Case can be a business `PASS` while Fixture fidelity remains `DECLARED_SYNTHETIC`. A `VerificationFixture FidelityProvider` supplies contracts keyed by `fixtureSetId/ruleId`; the verifier checks the Schema and optional adapter observations. The provider cannot return `PASS` or award itself a level.

```
VerificationFidelityReport fidelity = verifier.verifyFixtureFidelity(scenarioPath);
```

The first local-provider ceiling is `PROVIDER_ATTESTED_ADAPTER`. `ADAPTER_CONTRACT_VERIFIED` additionally requires the contract source, adapter implementation and configuration, and governance identity to enter source-bound evidence. Without a provider, the lower `DECLARED_SYNTHETIC` level means missing fidelity evidence, not a business execution failure.

25.6 Run three one-variable attacks

Use the same `manual-review` Case and restore the input after every attack. Otherwise the second result cannot be attributed to one change.



One changed fact	Earliest reliable observation	Customer code?	Recovery
Remove a Policy-protected Case	Candidate Policy weakening: `POLICY_REGRESSION`; missing Scenario Case: `REQUIRED_CASE_MISSING`	No	Restore the Case; do not weaken the baseline.
Invoke a Fixture rule beyond `maximum`	The next resolution fails closed as `CONTROL_RULE_OVER_CONSUMED`; verifier diagnostics expose `FIXTURE_OVER_CONSUMED:<set>/<rule>`	Execution has started; the extra real call is not allowed through	Fix the call count or the approved bound.
Change behavior to `ALLOW_REAL` under `CI_SAFE`	`REAL_EXECUTION_FORBIDDEN` while compiling the Fixture	No real port call	Keep a test double, or move the test to an explicitly isolated profile when such a profile exists.

The timing distinction matters. Case presence and `ALLOW_REAL` are knowable before execution. Over-consumption is not: the verifier must observe the extra invocation before it can reject it. Calling all three "preflight checks" would hide where the safety actually lives.

The focused RC1 run on 2026-09-14 at commit `cc38fbe5` covered `PolicyContinuityValidatorTest` (11), `FixtureBehaviorCompilerTest` (2), and `VerificationExecutionProfileTest` (2): 15 tests passed. `BehaviorActionTest` added 7 execution-control tests, including a two-response sequence whose third invocation raises `CONTROL_RULE_OVER_CONSUMED`. The source adapter in `ScenarioVerifier.controlReasons` translates the completion diagnostic to `FIXTURE_OVER_CONSUMED:<ruleReference>`.

25.7 Let status choose the next action

`FAIL` — the business answer is wrong. The contract was valid and execution completed, but a business or temporal Expectation did not match. Inspect the implementation, input, or approved answer before changing Policy.

`INVALID` — the verification declaration is wrong. Scenario, Policy, Fixture, DSL, or capability composition is invalid. Repair the declaration; retrying the same input is not useful.

`INCOMPLETE` — the run did not finish. Timeout, resource, adapter, or evidence capture stopped the run. Preserve diagnostics, repair the environment, then retry from a clean run.

Start with the earliest reason code:

- `REQUIRED_CASE_MISSING`: a Policy-required Case disappeared.
- `FIXTURE_UNDER_CONSUMED:<fixtureSetId>/<ruleId>`: the minimum consumption was not reached.
- `FIXTURE_OVER_CONSUMED:<fixtureSetId>/<ruleId>`: calls exceeded `maximum`.
- `REAL_EXECUTION_FORBIDDEN`: `CI_SAFE` encountered `ALLOW_REAL`.
- `FIXTURE_TARGET_ZERO_MATCH` / `FIXTURE_TARGET_AMBIGUOUS`: the target did not bind exactly one controlled call site.

25.8 Lab: create three failures, one at a time

Restore each factor before starting the next round.

1. Remove `manual-review` from `loan-approval.scenario.yaml`. The governed run should stop before customer bootstrap with `INVALID` and `REQUIRED_CASE_MISSING`. Restore the Case; do not lower Policy.
2. Set `maximum: 1` on a Fixture rule that is consumed twice. Expect `FIXTURE_OVER_CONSUMED:<fixtureSetId>/<ruleId>`. Inspect the attempt and target instead of making consumption unbounded.
3. Change an effect behavior to `ALLOW_REAL`. Under `CI_SAFE`, expect `INVALID` and `REAL_EXECUTION_FORBIDDEN`. Stop. A required real adapter belongs in a separately isolated and explicitly declared boundary.

Record the change, status, first reason code, whether bootstrap ran, and the recovery action. A terminal screenshot

alone is not reviewable evidence.

25.9 Transfer: high-value refund control

For an e-commerce refund flow, let Policy protect the high-value manual-review Case, let a Fixture control only `payment.refund`, and let `CI_SAFE` forbid the real payment adapter. Delete the Case, consume the Fixture twice, and allow the real port in three separate runs. The useful output is where each run stops, not whether the console eventually turns green.

25.10 Chapter boundary

- `BUSINESS_CONTRACT` covers business outcomes; `STRUCTURAL_CONTRACT` covers node, invocation, Fixture-consumption, and compensation facts.
- `INVOCATION` and `EFFECT_INPUT` are observation selectors, not production-adapter certification.
- Fixture success does not prove production HTTP, database, or message-protocol compatibility.
- `VerificationEffectResolverFingerprintProvider` binds resolver configuration; naturally changing runtime state does not belong in that configuration digest.

25.11 Experiment acceptance card

- **Expected and observed:** Policy, Fixture, and Assurance constrain action, simulation, and evidence strength separately.
- **Failure and recovery:** Remove a case, over-consume, or switch to ALLOW_REAL; repair from the reason.
- **Proof boundary:** Proves a controlled entry, not production equivalence of a simulator.
- **Exercise contract:** Manual-review case; attack one control at a time; deliver three receipts; stop when each reason has one action.

25.12 Coding practice

25.12.1 BP-14 — Test independent answers and forbidden actions

Use approved expected answers, not current outputs. Observe business results, node paths, and effect evidence independently; mock only the boundary whose behavior is intentionally controlled.

Try one change: Run the workshop blacklist-priority case and its reordered-rule mutant. Explain why preserving old behavior, rejecting one mutant, and proving all policies correct are different claims.

Reasoning, counterexamples, and limits

25.13 Summary

Policy prevents silent scope reduction, Fixture controls external behavior, and the execution profile fixes reproducibility inputs. Interpret a business `PASS` or `FAIL` only after policy/profile preflight succeeds and Fixture-consumption checks complete. The next chapter applies the same discipline to suspension, signals, resume, and logical time.

Next: Chapter 26 — Durable and Temporal Verification

CHAPTER
26

Arc 6 — Business Promises

Chapter 26 — Durable and Temporal Verification

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

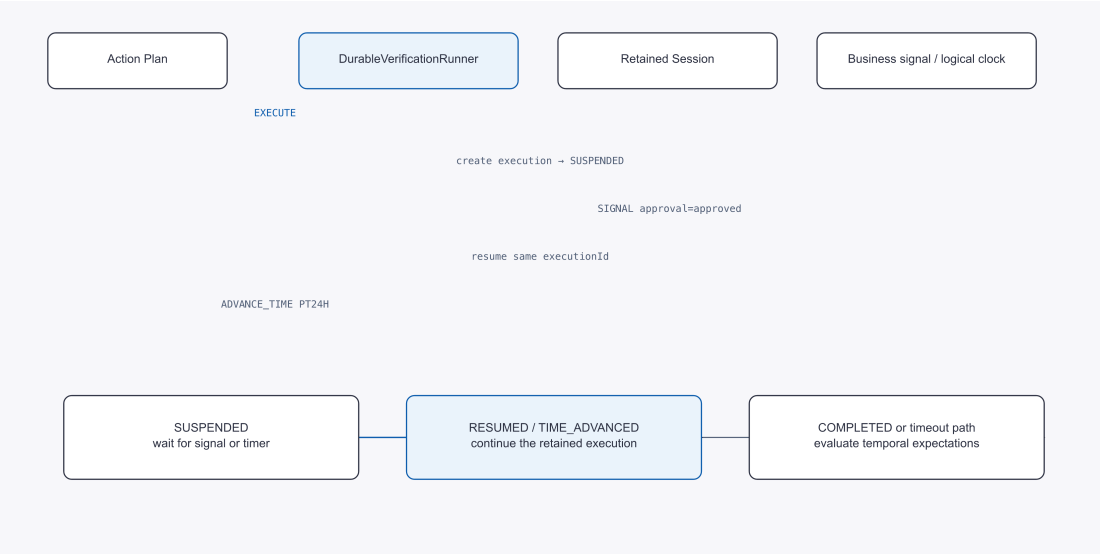
Chapter promise: *You will verify suspension, signal, resume, and logical time on one retained execution with a finite action plan, while keeping final outcome, forbidden-before, and required-within facts separate.*

26.1 Learning goals

1. Use `EXECUTE`, `SIGNAL`, `ADVANCE_TIME`, and `RESUME` instead of flattening a long-running flow into one method call.
2. Separate a local `DurableVerificationRunner` report from `DurableSourceBoundVerifier` source-bound evidence.
3. Express finite step, trace, and effect facts with `DurableTemporalContract`.
4. Recognize streaming, timer, checkpoint, and non-cooperative timeout stop boundaries.

26.2 Final output cannot show the waiting path

Two manual-review executions may both finish as `approved`. One sends the notification after the human decision; the other sends it while still waiting. A final-output-only test misses the second defect.



`DurableVerificationRunner` creates a real `DurableGraphEngine` and retained session for one Case. Its default stores are isolated and in memory. Actions run serially against one execution identity; logical time is verifier-controlled and does not depend on elapsed wall-clock test time.

26.3 An action plan is a bounded script, not a workflow language

This path starts the flow and receives approval within 24 hours:

```yaml
schemaVersion: 1
actions:
  - actionId: start
    type: EXECUTE
  - actionId: wait-before-decision
    type: ADVANCE_TIME
    duration: PT23H
  - actionId: approve
    type: SIGNAL
    nodeId: approval
    signalData: approved
```

The constraints matter more than the syntax:

- A plan contains 1–64 actions.
- Its first and only `EXECUTE` action is first.
- `ADVANCE_TIME` requires a positive ISO-8601 duration.
- `SIGNAL` must identify an exact currently suspended node.
- `RESUME` is for a failed execution with a stored checkpoint. A suspended execution needs `SIGNAL` first.
- `CANCEL` has no stable durable seam and returns `DURABLE_ACTION_UNSUPPORTED` from the sidecar loader.

Use a separate plan for the timeout path, advancing logical time to `PT24H` or the business-approved boundary. “Signal received” and “no signal received” are two business Cases, not branches to hide in one linear plan.

26.4 Check step state and event order together

```yaml
schemaVersion: 1
stepExpectations:
  - expectationId: initially-suspended
    afterActionId: start
```

```
    outcome: SUSPENDED
    suspendedNodeIds: [approval]
traceExpectations:
  - expectationId: notify-after-approval
    matcher: MUST_NOT_OCCUR_BEFORE
    event: {type: NODE_COMPLETED, graphName: approval-flow, nodeId: notification}
    afterActionId: approve
  - expectationId: notify-within-five-minutes
    matcher: MUST_OCCUR_WITHIN_LOGICAL_TIME
    event: {type: NODE_COMPLETED, graphName: approval-flow, nodeId: notification}
    within: PT5M
```

`stepExpectations` inspect the outcome and suspended set after an action. `traceExpectations` consume BLOGE listener facts: node identity, owning action, and logical-time offset. They do not evaluate scripts, exception text, or raw business payloads.

Use temporal v2 for duplicate business effects:

```
schemaVersion: 2
effectExpectations:
  - expectationId: ledger-booked-once
    matcher: EXACTLY_ONCE_EFFECT_KEY
    effect: {portId: ledger.book, key: LA-42}
```

Every participating Operator needs a complete dependency manifest, and the target `EffectPort` must declare a stable `idempotencyKey(input)`. Successful, failed, and denied calls all count because the external system may accept an effect before the caller sees a failure. The report stores a typed digest, not the raw key.

26.5 Run local durable verification

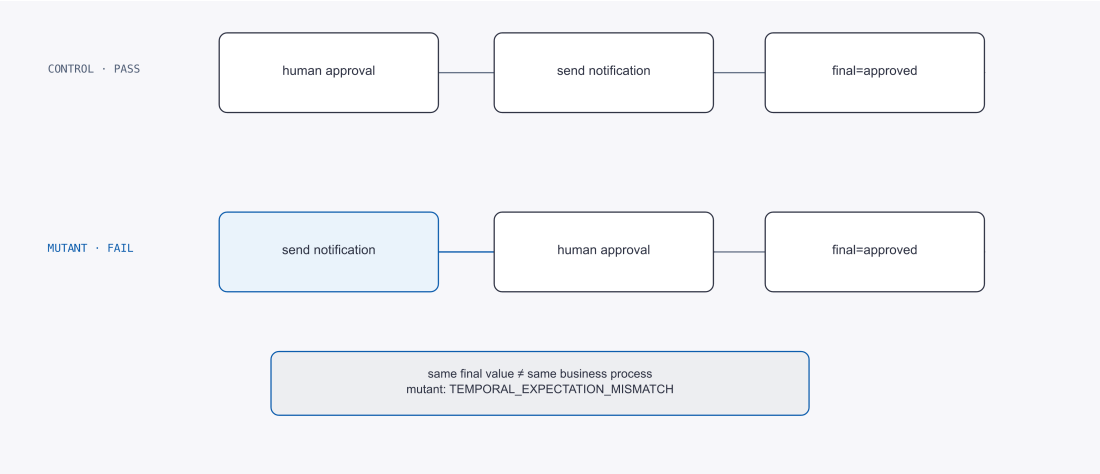
```
DurableVerificationPlan plan = DurableVerificationPlan.load(actionPath);
DurableTemporalContract temporal = DurableTemporalContract.load(temporalPath);

DurableVerificationReport report = DurableVerificationRunner.builder()
        .harnessTimeout(Duration.ofSeconds(30))
        .build()
        .verify(
                new DurableVerificationInput(
                        new VerificationCaseContext("loan-review", "approved-in-time"),
                        compiledGraph,
                        initialContext,
                        compiledOperators),
                plan,
                temporal);
```

This report is local authoring evidence. Publishing source-bound durable evidence requires `DurableSourceBoundVerifier` to compile tracked graph and sidecar inputs from a clean Git worktree, a freezable registry and resolver, a `DURABLE` execution profile, and an ignored non-classpath output directory. Its reader does not call customer bootstrap to recompile the candidate.

26.6 Observe the same final output with a different order

Run two versions of the same manual-review graph. Both end with `decision=approved`; only the notification position changes.




```
control: approve → notification → final approved
mutant:  notification → approve → final approved
```

`DurableTemporalContractTest.detectsAnEventThatOccurredBeforeItsRequiredActionDespiteFinalSuccess` executes the mutant. The final action still records `COMPLETED`, but the report is `FAIL`, its only reason code is `TEMPORAL_EXPECTATION_MISMATCH`, and `notification-after-approval.matched=false`.

This one test separates three facts that a final JSON object would collapse:

- the durable execution completed;
- the final business value was available;
- the required event order was violated.

The focused RC1 run at `cc38fbe5` executed 19 temporal-contract tests and 11 durable-scenario tests: 30 passed, with no failures, errors, or skips. The evidence is a finite declared relation, not a claim that every possible interleaving was explored.

26.7 A temporal mismatch is a business FAIL

When a valid run completes but `notification` occurs before `approve`, the result is `FAIL` with `TEMPORAL_EXPECTATION_MISMATCH`. That differs from an invalid sidecar: an unknown action, bad node identity, incomplete dependency manifest, or unsupported capability composition is normally `INVALID`.

- **`DURABLE_SIGNAL_TARGET_INVALID`** — the signal does not identify the suspended node. Inspect the suspended set; do not broaden the target.
- **`DURABLE_RESUME_STATE_INVALID`** — `RESUME` was used on a suspended execution. Use `SIGNAL`; reserve resume for a failed checkpointed execution.
- **`DURABLE_CHECKPOINT_MISSING`** — no checkpoint exists. Stop and inspect persistence; do not fabricate one.
- **`DURABLE_TIMER_SET_AMBIGUOUS`** — one time advance expires multiple timers. Preview fails closed; split the plan or revise the model.
- **`STREAM_DURABLE_UNSUPPORTED`** — a streaming graph entered the durable wrapper. Test the streaming boundary; do not claim undeclared support.
- **`TEMPORAL_EXPECTATION_MISMATCH`** — a valid temporal contract did not match. Treat it as business `FAIL` and inspect the first divergent event.

After a harness timeout, the runner interrupts customer code and waits for settlement grace. A task that still does not exit produces `DURABLE_HARNESS_ORPHANED`; the runner is then poisoned and must not admit another plan in the same process.

26.8 Lab: signal path and timeout path

1. Create `approved-in-time`: `EXECUTE → ADVANCE_TIME PT23H → SIGNAL approved`.
2. Require `notification` not to occur before `approve`, and to complete within the business-approved logical duration afterward.
3. Create a separate `review-timeout`: `EXECUTE → ADVANCE_TIME PT24H`, expecting the timeout path.
4. Move `notification` before the signal and confirm `FAIL/TEMPORAL_EXPECTATION_MISMATCH`; restore it, then verify both paths.
5. Preserve the action trace, temporal expectation result, and first reason code. Do not put raw signal payloads in the report.

Stop before opening a session if streaming classification is incomplete, a custom nested provider cannot be statically inspected, or effect inventory is incomplete. Repair the capability declaration before discussing business results.

26.9 Transfer: an insurance-claim document wait

An insurance claim may reach the same final `APPROVED` value whether missing documents arrived before the review deadline or after an escalation was sent. Model document arrival as a signal, the service-level deadline as logical time, and escalation-before-approval as a temporal relation. Preserve the event trace; the final claim object alone cannot tell those histories apart.

26.10 Chapter boundary

- `ADVANCE_TIME` advances verifier-owned logical time; it does not simulate real scheduling latency.
- Step and trace contracts prove named finite relations, not every possible interleaving.
- A persistence provider supplies per-Case stores and a unique mutable-resource namespace; the verifier still owns the engine, action order, and shutdown.
- Source-bound manifests contain a canonical SHA-256 of initial context, not business payloads.

26.11 Experiment acceptance card

- **Expected and observed:** The same final output still fails when event order violates the temporal expectation.
- **Failure and recovery:** Move notification before approval; restore order and replay.
- **Proof boundary:** Proves timing for the fixed action plan, not every concurrency interleaving.
- **Exercise contract:** Approval and notification events; swap order only; deliver matched results; stop when order alone explains the failure.

26.12 Coding practice

26.12.1 BP-14 — Test independent answers and forbidden actions

Use approved expected answers, not current outputs. Observe business results, node paths, and effect evidence independently; mock only the boundary whose behavior is intentionally controlled.

Try one change: Run the workshop blacklist-priority case and its reordered-rule mutant. Explain why preserving old behavior, rejecting one mutant, and proving all policies correct are different claims.

Reasoning, counterexamples, and limits

26.13 Summary

Long-flow correctness combines outcomes, state boundaries, and event-time relations. A finite action plan makes those facts repeatable; a temporal sidecar makes them diagnosable. The next chapter asks whether passing Cases can actually detect a wrong rule.

Next: Chapter 27 — Finding Verification Blind Spots

CHAPTER
27

Arc 6 — Business Promises

Chapter 27 — Contract Sensitivity: Can the Suite Reject a Wrong Answer?

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

Chapter promise: *Turn one green loan Case into a controlled negative experiment, then distinguish a `KILLED` witness from a `SURVIVED` blind spot without confusing either result with exhaustive mutation coverage.*

27.1 Learning goals

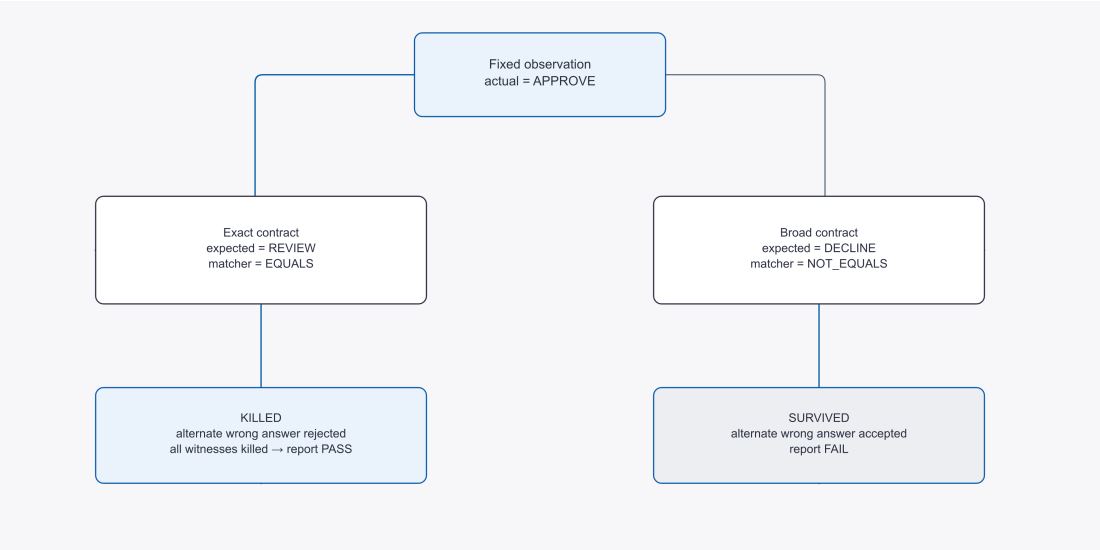
1. Explain why "candidate equals expected" does not prove that the contract rejects nearby wrong answers.
2. Use `verifyContractSensitivity(Path)` against one governed `BUSINESS_CONTRACT` Expectation.
3. Read `KILLED`, `SURVIVED`, report status, and source-bound evidence as separate facts.
4. Design one-variable witnesses that a business owner can review.

27.2 The same green Case can hide two very different contracts

The low-risk loan Case expects `APPROVE`, and the graph returns `APPROVE`. Now imagine two matchers:

- exact: actual must equal `APPROVE`;
- broad: actual must merely be different from `DECLINE`.

Both pass today. Only the first rejects `REVIEW`. A positive example establishes that one point matches; sensitivity asks whether a selected wrong answer crosses the contract boundary.



27.3 Change the expected answer, not the running system

The sidecar names the exact governed Expectation and replaces only its expected value:

```
schemaVersion: 1
sensitivityId: loan-decision-witnesses
scenario: loan-approval.scenario.yaml
mutants:
  - mutantId: low-risk-must-not-review
    requirementRef: LOAN-DECISION-001
    caseId: auto-approve
    expectationId: decision-approved
    kind: EXPECTATION_VALUE_REPLACED
    replacementExpected: REVIEW
```

```
VerificationContractSensitivityReport result =
        verifier.verifyContractSensitivity(sensitivityPath);
```

BLOGE executes the baseline Case once. It then projects the real observation onto the alternate expected value. It does not patch production code, rerun a different Operator, or invent a business rule. That isolation matters: if both the implementation and Oracle changed, a failure would no longer identify contract sensitivity.

27.4 Observe the transition, not just the final badge

The RC1 test `ContractSensitivityTest.passesOnlyWhenEveryRequiredWitnessIsKilled` uses an exact `EQUALS` Expectation. The graph still observes `APPROVE`; replacing expected with `REVIEW` is rejected, so the witness is `KILLED`, the report is `PASS`, and `reasonCodes` is empty.

The companion test `sealsASurvivingWitnessAsARecheckableSensitivityFailure` starts from a broader "not review" style contract and replaces expected with `DECLINE`. The same observation is still accepted. The witness becomes `SURVIVED`, the report becomes `FAIL`, and the published artifact remains independently recheckable as `SOURCE_BOUND` evidence.

Observation	Witness outcome	Report	Meaning
alternate expected is rejected	`KILLED`	`PASS` only if every required witness is killed	the contract notices this selected wrong answer
alternate expected is still accepted	`SURVIVED`	`FAIL`	the matcher or Oracle boundary is too broad for this witness

Observation	Witness outcome	Report	Meaning
plan points to a missing/unsupported target	`NOT_EVALUATED`	`INVALID`	the experiment itself is not valid

The important asymmetry is deliberate: a surviving witness is useful failure evidence. It should not disappear because the business Case happened to be green.

27.5 Design witnesses from business mistakes

Do not enumerate random strings. Start from a mistake the business can name:

1. Pick one protected decision, such as "low-risk applications must not enter manual review."
2. Keep Scenario, Policy, Fixture, graph, execution profile, and source commit fixed.
3. Replace one expected value with one plausible wrong answer.
4. Ask the owner whether the witness represents a real decision boundary.
5. Keep the witness in version control only after that review.

For loan approval, useful witnesses include `REVIEW` for an auto-approve Case and `APPROVE` for a blacklist Case. A nonsense value may test serialization while teaching nothing about the business contract.

27.6 Lab: make one witness survive

1. Run the low-risk Case and record its baseline `PASS` and actual `APPROVE`.
2. Add the `REVIEW` replacement above; verify that the exact matcher yields `KILLED`.
3. Change only the matcher to a broader condition that also accepts `REVIEW`.
4. Run the same sidecar again; record `SURVIVED` and `CONTRACT_MUTANT_SURVIVED:<mutantId>`.
5. Restore the exact matcher and rerun. The witness must return to `KILLED`.
6. Save the artifact receipt outside the run directory and ask another reader to verify it.

This is a two-run, one-variable experiment: the witness is fixed, and only contract strictness changes. That makes the result explainable.

27.7 Transfer: discount-rule sensitivity

For retail pricing, replace an expected 10% loyalty discount with 30% while keeping the cart and implementation fixed. An exact amount matcher should kill that witness; a matcher that checks only `discount > 0` will let it survive. The result measures whether the contract rejects this chosen pricing error, not whether every promotion interaction is covered.

27.8 Chapter boundary

- Sensitivity covers only author-declared mutations; it is not mutation coverage of all rules or outputs.
- A `KILLED` witness proves rejection of one selected wrong answer, not correctness over an input domain.
- A `SURVIVED` witness locates an ambiguity; it does not decide whether the matcher or business Oracle should change.
- Source-bound publication makes the result recheckable; it does not turn the tool into the business owner.

27.9 Experiment acceptance card

- **Expected and observed:** One observation becomes KILLED under an exact contract and SURVIVED under a loose one.
- **Failure and recovery:** Loosen one expectation so the witness survives; restore the exact answer.
- **Proof boundary:** Proves sensitivity to a declared witness, not discovery of every blind spot.
- **Exercise contract:** Green case and one witness; change one expectation; deliver the transition; stop when the bad witness is KILLED.

27.10 Coding practice

27.10.1 BP-14 — Test independent answers and forbidden actions

Use approved expected answers, not current outputs. Observe business results, node paths, and effect evidence independently; mock only the boundary whose behavior is intentionally controlled.

Try one change: Run the workshop blacklist-priority case and its reordered-rule mutant. Explain why preserving old behavior, rejecting one mutant, and proving all policies correct are different claims.

Reasoning, counterexamples, and limits

27.11 Summary

A green example says “this answer matched.” A killed witness adds “this selected wrong answer would not have matched.” A surviving witness exposes where the contract is too permissive. Chapter 28 changes the question again: instead of one hand-picked wrong answer, it explores a bounded input domain and reduces a real failure.

Next: Chapter 28 — Bounded Property Verification and Useful Counterexamples

CHAPTER
28

Arc 6 — Business Promises

Chapter 28 — Bounded Property Verification and Useful Counterexamples

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

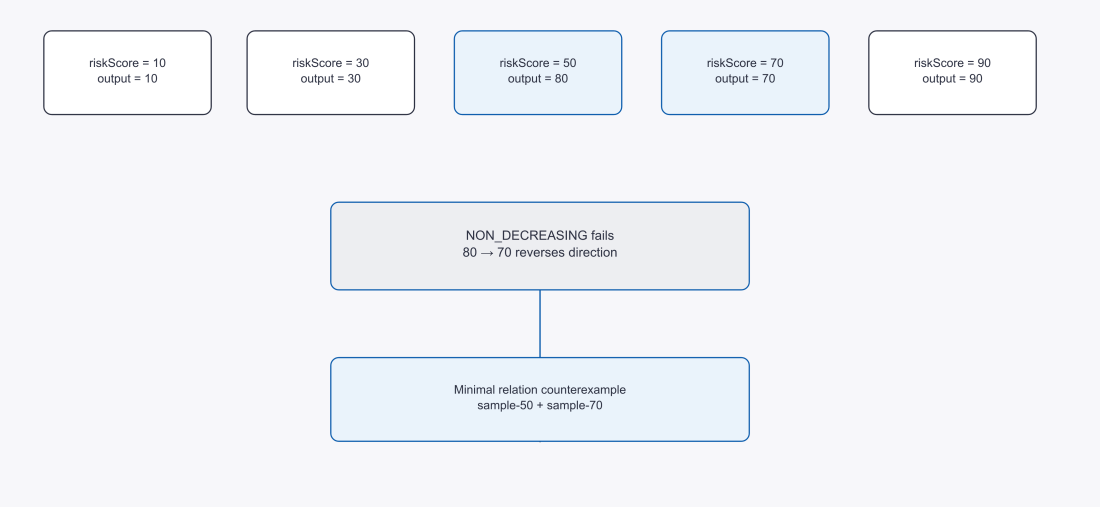
Chapter promise: *Explore a declared finite loan-risk domain, separate sample failures from relation failures, replay BLOGE's deterministic shrink trace, and know when a smaller counterexample still needs business review.*

28.1 Learning goals

1. Define a bounded generator attached to an existing Case and its Expectations.
2. Distinguish a failing sample, a relation counterexample, and a value-shrink result.
3. Read budget exhaustion and incomplete shrink as non-pass outcomes.
4. Keep the Oracle independent enough to challenge the candidate.

28.2 Five samples answer more than one example—and less than a proof

The loan team trusts three named Cases but worries about risk-score boundaries. They keep `auto-approve` as the anchor and replace only `given.context.riskScore` with five declared values.




```
schemaVersion: 1
propertySetId: loan-risk-boundaries
scenario: loan-approval.scenario.yaml
rootSeed: 11
sampleBudget: 5
generator:
  generatorId: risk-scores
  requirementRef: LOAN-DECISION-001
  caseId: auto-approve
  contextKey: riskScore
  integerRange: {minimum: 10, maximum: 90, step: 20}
  expectationIds: [decision-valid, risk-score-recorded]
relations:
  - relationId: risk-never-decreases
    kind: NON_DECREASING
    actual: {kind: BUSINESS_OUTPUT, outputId: computedRiskScore}
```

```
VerificationPropertyReport property = verifier.verifyProperties(propertyPath);
```

Each sample still runs every non-diagnostic Expectation in the anchor Case. `rootSeed` makes supported sample selection and ordering repeatable; it does not control clocks, randomness, network state, or environment reads inside customer Operators. The claim is exactly five declared points, not "all integer risk scores."

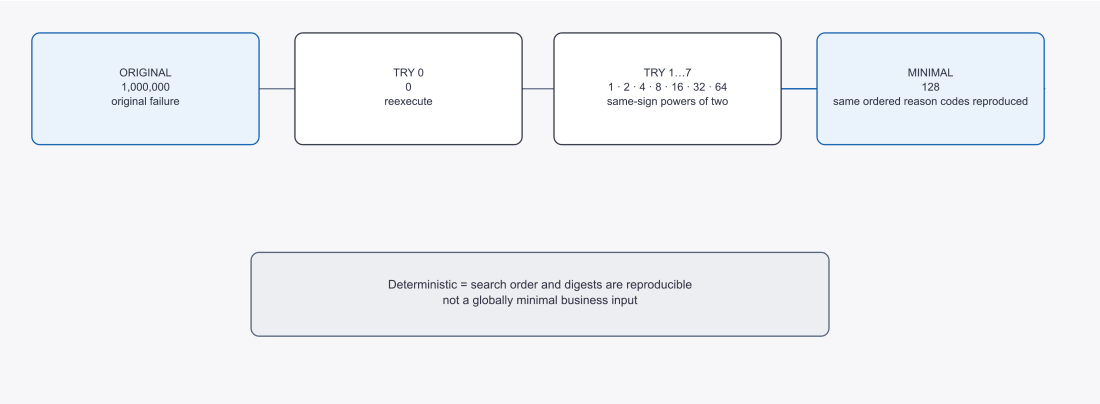
28.3 Three failures answer three different questions

Evidence object	What failed?	How it becomes smaller
sample	ordinary business Expectations for one generated input	value shrink may reexecute candidates
relation counterexample	a relation across executed samples	selects a smaller subset of those sample IDs
property report	any required sample, relation, or shrink protocol	aggregates status and stable reason codes

A relation `shrinkTrace` never invents an input. A monotonicity violation can reduce the executed set to the first pair that demonstrates inversion. Value shrink is different: it performs new executions after one relation-free sample has failed.

28.4 Replay the actual integer shrink

For a relation-free integer plan with `maximumAttempts: 64`, RC1 test `deterministicallyShrinksAnIntegerFailureByReexecutingCandidates` observes this exact sequence twice:



`1,000,000 → 0 → 1 → 2 → 4 → 8 → 16 → 32 → 64 → 128`

The first value is the original failing sample. Candidates begin at zero and then use same-sign powers of two. A candidate counts only when it reproduces the original ordered reason codes. The test repeats the experiment and compares both the observed sequence and candidate context digests; both runs are identical.

The final accepted candidate is not "the mathematically smallest bad loan." It is the minimal witness produced by this candidate order and failure-equivalence rule.

28.5 Counterexample reduction does not promote the Oracle

Suppose `[10, 30, 50, 70, 90]` yields `[10, 30, 80, 70, 90]`. Every sample may satisfy local shape checks, but `NON_DECREASING` fails because `80 → 70` reverses direction. A relation counterexample can retain only those two sample IDs.

That pair is a diagnostic asset, not automatically a new Golden Case. The business owner still decides whether the relation is correct, whether generated contexts are legal, and whether the reduced input deserves an approved expected answer.

28.6 Shrink can end without a usable minimum

- If the original sample passes, BLOGE does not attempt shrink.
- If the attempt budget ends while candidates remain, the result is `INCOMPLETE / RESOURCE_LIMIT_EXCEEDED`.
- If no candidate reproduces the original ordered reason codes, the result is `INCOMPLETE / PROPERTY_SHRINK_INCOMPLETE`.
- If a candidate fails for a different reason, it is not evidence of the same bug.

None can be relabeled as business `PASS`. "Could not finish diagnosis" and "verified correct" are opposite statements.

28.7 Keep the judge outside the candidate

`verifyOracleIndependence(Path)` separates three inspectable facts:

- `CONTENT_BOUND`: approved Oracle bytes match the catalog digest.
- `OWNER_SEPARATED`: catalog owner labels differ; this is a declaration, not identity authentication.
- `REFERENCE_MODEL`: a bound implementation may run in a child JVM, with its content closure checked against candidate runtime and Operator bytes.

The evidence reader may rerun only that reference model; it does not run candidate Operators or customer bootstrap. Byte disjointness cannot detect copied decision logic, so conceptual independence and owner identity remain external

governance questions.

28.8 Lab: turn one inversion into a review card

1. Run the five-score plan and save sample IDs, outputs, root seed, and receipt.
2. Introduce one threshold inversion while keeping every other input fixed.
3. Record the failing relation and its minimal sample-ID subset.
4. Use a separate relation-free plan to trigger value shrink; do not mix the two mechanisms.
5. Copy the complete candidate order into the review card.
6. Ask the business owner whether the reduced witness is legal, meaningful, and worthy of a governed Case.
7. Restore the implementation and confirm both the property and original Case pass.

28.9 Transfer: shrink a logistics-ordering failure

For shipment tracking, sample a bounded set of event sequences and require `deliveredAt >= shippedAt`. If one sequence violates the relation, reduce the sequence to the smallest replayable pair of events while retaining the original seed and candidate order. The reduced trace is a diagnosis aid; a logistics owner still decides whether it is a valid governed Case.

28.10 Chapter boundary

- Bounded property verification covers the declared finite plan, not an infinite domain or statistical confidence.
- Relation reduction chooses among executed samples; value shrink creates and reexecutes candidates.
- Deterministic shrink means reproducible search order, not globally minimal business input.
- Oracle byte separation is useful evidence, not proof of human independence or conceptual originality.

28.11 Experiment acceptance card

- **Expected and observed:** A bounded domain finds and shrinks a counterexample to a reviewable candidate.
- **Failure and recovery:** Move the oracle into the candidate or stall shrinking; restore an external judge and stop reason.
- **Proof boundary:** Proves a property within chosen bounds, not a universal proof.
- **Exercise contract:** One property and bounded domain; change the bound; deliver seed, counterexample, and shrink trace; stop when stable and replayable.

28.12 Coding practice

28.12.1 BP-14 — Test independent answers and forbidden actions

Use approved expected answers, not current outputs. Observe business results, node paths, and effect evidence independently; mock only the boundary whose behavior is intentionally controlled.

Try one change: Run the workshop blacklist-priority case and its reordered-rule mutant. Explain why preserving old behavior, rejecting one mutant, and proving all policies correct are different claims.

Reasoning, counterexamples, and limits

28.13 Summary

Property verification expands one named Case into a controlled finite experiment. Counterexamples identify a small already-observed relation failure; value shrink reexecutes candidates to leave a smaller reproducible witness. The next chapter asks whether these results still deserve trust after their files, receipt, source, or runtime context changes.

Next: Chapter 29 — From Report to Source-bound Evidence

CHAPTER
29

Arc 6 — Business Promises

Chapter 29 — From Report to Source-bound Evidence

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

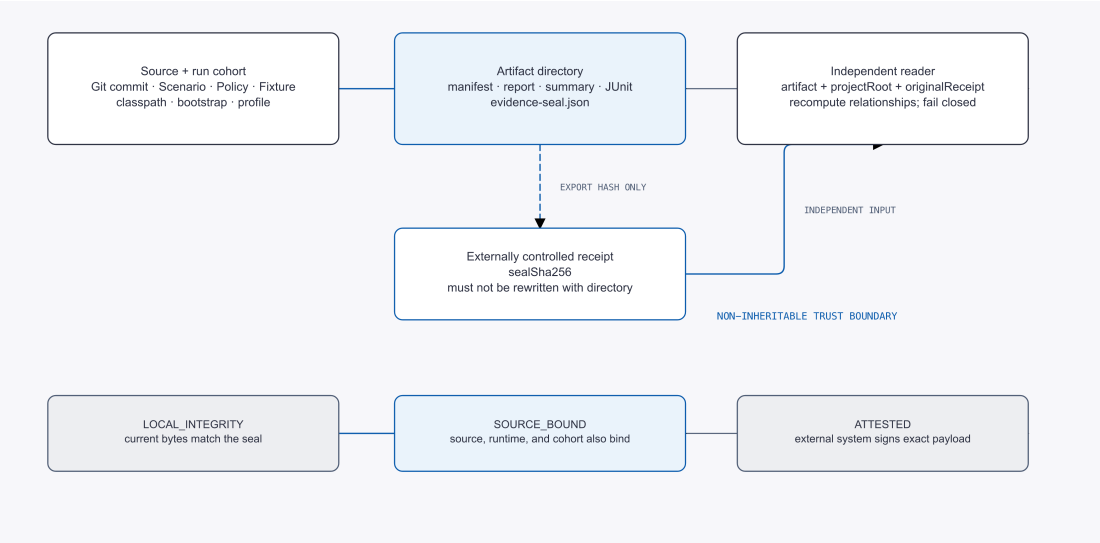
Chapter promise: *You will distinguish a business result from a sealed directory, keep the receipt outside that directory, and use an independent reader to decide whether the evidence is merely intact, bound to source, or externally attested.*

29.1 Learning goals

1. Explain the jobs of report, manifest, seal, receipt, and reader.
2. Distinguish `LOCAL_INTEGRITY`, `SOURCE_BOUND`, and `ATTESTED`.
3. Identify which claim is lost after bytes, directory location, Git commit, or runtime cohort changes.
4. Read evidence through a typed reader instead of parsing JSON fields into a release decision.

29.2 A copied green report is only a copied green report

The team copies `verification-report.json` from a developer laptop to a release machine. It still says `PASS`. That text preserves a claim somebody made; by itself it does not identify the source, toolchain, inputs, or run that made it.



The orange break is deliberate: the receipt must not travel inside the mutable artifact directory. Accepting a newly calculated receipt from the same party that can rewrite the directory collapses the boundary.

29.3 Five files form one sealed projection

A published Suite directory can contain:

- `verification-manifest.json`: bound inputs, runtime identity, Git and run cohort facts.
- `verification-report.json`: payload-free Suite, Case, Requirement, assurance, coverage, and specialized results.
- `verification-summary.md`: human review projection.
- JUnit XML: build-system projection of the same verdicts.
- `evidence-seal.json`: hashes and relationships for the published files.

`VerificationArtifacts` returns the run directory, run ID, and `sealSha256()`. Store that SHA-256 in an independently controlled CI record. It is the receipt the later reader must receive from outside the directory.

29.4 Integrity and provenance are different promotions

`LOCAL_INTEGRITY` means the files agree with the seal. A person who can rewrite the directory can also create a new internally consistent seal.

`SOURCE_BOUND` additionally binds the exact tracked inputs, clean Git commit and worktree, execution profile, classpath/runtime, bootstrap and Operator identity, plus the run cohort supported by that evidence version.

`ATTESTED` requires an external controlled provenance system to sign the exact source-bound payload under an explicitly trusted key and domain. A signature does not repair missing source binding.

Business status stays independent. A trustworthy `FAIL` can be source-bound; an unbound `PASS` remains unsuitable for a team gate.

29.5 Publish once, verify from the outside

Configure publication before the run:

```
ScenarioVerifier verifier = ScenarioVerifier.builder()
        .bootstrap(new LoanVerificationBootstrap())
        .projectRoot(projectRoot)
        .policy(policyPath)
        .outputDirectory(projectRoot.resolve("target/bloge-verification"))
```

```
        .executionProfile(VerificationExecutionProfile.builder("ci-safe").build())
        .sourceBoundEvidence()
        .build();

VerificationReport report = verifier.verify(scenarioPath);
VerificationArtifacts artifacts = report.artifacts();
String externalReceipt = artifacts.sealSha256();
```

Later, pass all three independent inputs to the reader:

```
VerificationEvidenceTrustResult trust = VerificationSourceBoundEvidence.verify(
        artifacts.directory(), projectRoot, externalReceipt);
```

The reader recomputes report, summary, JUnit, manifest, source, and receipt relationships. It does not run candidate Operators or customer bootstrap. Oracle v10/v11 is the narrow exception: the reader may rerun the explicitly bound reference-model child JVM.

29.6 Attack five boundaries and watch trust move

Start every row from the same untouched artifact: business status `PASS`, independently supplied receipt, and reader trust `SOURCE_BOUND`. Change exactly one thing.



One-variable attack	Independent reader observation	Highest defensible trust	What was lost
replace the external receipt	`EVIDENCE_RECEIPT_MISMATCH`	`UNVERIFIED`	identity of the sealed byte snapshot
edit summary, update its digest, and reseal	`DURABLE_EVIDENCE_PROJECTION_MISMATCH`	below valid local evidence	agreement between human and machine projections
edit report semantics, regenerate projections, and reseal	`DURABLE_EVIDENCE_INVALID`	below valid local evidence	an internally possible verifier result
alter an action digest, regenerate projections, and reseal	`SOURCE_BINDING_CASE_EVIDENCE_MISMATCH`	`LOCAL_INTEGRITY`	relation to the tracked action plan
mutate tracked source after publication	`SOURCE_BINDING_DRIFT`	`LOCAL_INTEGRITY`	relation to the current source cohort

These are observed RC1 contracts, not hypothetical labels. `DurableVerificationEvidenceTest.rejectsAProjectionMutationEvenWhenTheArtifactDigestWasUpdated` proves that updating hashes cannot make a contradictory summary valid. `rejectsAContradictoryReportEvenAfterAllFilesAreReprojectedAndResealed` proves

that a fully consistent forgery can still describe an impossible verifier state. `DurableSourceBoundVerifierTest.rejectsReplacementReceiptBeforeTrustingSourceBoundDocuments` rejects a borrowed receipt before trusting source-bound documents.

The source-drift test adds one operationally important observation: after publication, the plan file changes, the reader returns `LOCAL_INTEGRITY / SOURCE_BINDING_DRIFT`, and the customer-call counter remains `1`. The reader downgraded trust without executing customer code again.

29.7 Why resealing cannot restore provenance

The artifact owner can recompute hashes. Therefore a new seal can sometimes restore byte consistency, but it cannot restore the externally retained receipt, the original run identity, or agreement with tracked source. The reader calculates the highest still-supported level; it does not accept the directory's self-description.

Moving or copying the directory has the same principle. The bytes remain useful for inspection, but source-bound publication expects its ignored, non-classpath location inside the bound worktree. A copy does not inherit decision authority merely because filenames and JSON fields survived.

Readers return stable diagnostics and fail closed. Do not replace their result with a filename, a green `status` field, or a manually parsed `evidenceTrust` value.

29.8 Lab: attack one boundary at a time

1. Publish one source-bound loan Suite and save `sealSha256()` outside its run directory.
2. Verify the untouched artifact with the original project root and receipt; record `SOURCE_BOUND`.
3. Execute the five attacks in the table, restoring the untouched artifact before each one.
4. Record business status, reader trust, complete reason codes, and customer-call count.
5. For the reseal attacks, keep both the original receipt and replacement receipt; verify with each and explain why they answer different questions.
6. Copy the directory outside its original worktree and confirm that readable files do not inherit source-bound authority.
7. Finish with a card containing: `status`, `trust`, receipt custodian, project commit, first failed relationship, and rerun policy.

29.9 Transfer: payroll evidence custody

A payroll report copied from another worktree can still be valid JSON and show every employee calculation as `PASS`. It must not inherit authority from the original run. Bind the payroll rules, employee-input digest, runtime inventory, commit, and external receipt; then move or reseal one component at a time and record the first trust relationship that fails.

29.10 Chapter boundary

- A seal proves relationships among current bytes; it does not identify who produced them.
- `SOURCE_BOUND` does not prove external account identity, host isolation, production topology, or policy approval authenticity.
- Copying a report is useful for reading, but not for inheriting its source-bound decision capability.
- Use the matching reader and schema catalog; do not infer evidence versions from class names or JSON shape.

29.11 Experiment acceptance card

- **Expected and observed:** Five files form source-bound evidence; integrity and provenance upgrade separately.
- **Failure and recovery:** Replace a receipt, edit source, or reseal identity; restore the chain or rerun.
- **Proof boundary:** Proves the fixed reader can verify the chain, not that customer code was never altered.
- **Exercise contract:** One sealed projection; attack one file at a time; deliver trust/reason changes; stop when all five attacks are attributable.

29.12 Summary

Artifact files make a result inspectable, an external receipt makes rewriting detectable, and the reader reconnects both to current source facts. The next chapter determines whether one trusted Case fact contributes to one exact Requirement; Project and Reactor aggregation follow in Chapter 31.

Next: Chapter 30 — From Case Facts to an Exact Requirement Contribution

CHAPTER
30

Arc 6 — Business Promises

Chapter 30 — From Case Facts to an Exact Requirement Contribution

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

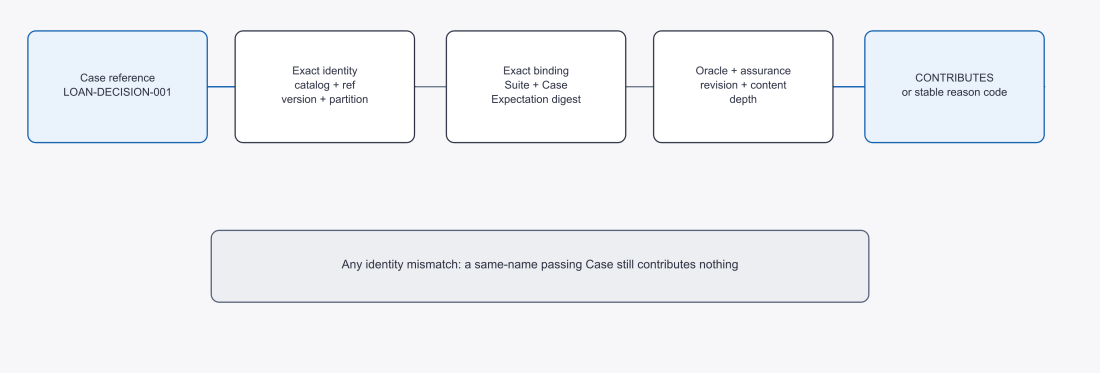
Chapter promise: *Give a business Requirement an exact identity, bind it to approved Oracle bytes and Expectation semantics, and explain why a passing Case with the same reference may still contribute nothing.*

30.1 Learning goals

1. Separate Requirement association from governed contribution.
2. Select exactly one versioned partition with an explicit evaluation context.
3. Bind exact Expectation semantics and Oracle revision, not only names.
4. Read pre-bootstrap governance failures without calling them business failures.

30.2 “Mentions the rule” is not “satisfies the rule”

Three loan Cases mention `LOAN-DECISION-001`. The release obligation, however, is version 3 for the `cn-retail` partition and release `2026.09`. A Case can pass and still fail to contribute because it belongs to version 2, an EU partition, a retired Oracle, a different Expectation digest, or insufficient assurance depth.



The distinction prevents an easy but dangerous shortcut: counting strings in reports and calling the Requirement satis-

fied.

30.3 Build an identity that cannot collide

```yaml
schemaVersion: 1
catalogId: loan-governance
requirements:
  - requirementRef: LOAN-DECISION-001
    version: 3
    owner: risk-policy-team
    status: ACTIVE
    effectiveFrom: 2026-09-01T00:00:00Z
    partitions:
      - partitionId: cn-retail
        dimensions: {jurisdiction: CN, product: RETAIL_LOAN}
        targetReleaseIds: ['2026.09']
        obligations:
          minimumPassingCases: 2
          minimumDepth: L2
          allowedOracleTypes: [APPROVED_EXAMPLE]
```

The effective identity is:

`catalogId + requirementRef + version + partitionId`

`GovernanceEvaluationContext` supplies `asOf`, target release, and finite dimensions. Exactly one partition must apply. A bare `LOAN-DECISION-001` cannot borrow evidence from another version or market.

RC1 test `producesOnlyTheExactApplicableVersionedPartitionContribution` observes one key—`loan-governance / LOAN-R-017 / v3 / cn-retail`—with `approve-low-risk` as both associated and contributing Case. The same ref at version 2 or partition `eu-retail` is a different key. Selecting release `2026.10` returns `REQUIREMENT_NOT_APPLICABLE` without another customer bootstrap.

30.4 Bind the answer, not just the label

The binding fixes:

- `suiteId + caseId`;
- exact `expectationIds` and normalized `expectationSha256`;
- `oracleId + oracleRevision` and approved artifact digest;
- minimum assurance and allowed Oracle type.

This closes the "same names, changed meaning" loophole. In `bindsExactExpectationSemanticsAndReportsInsufficientAssurance`, changing only expected `APPROVE` to `REVIEW` leaves all names intact but yields `INVALID / GOVERNANCE_BINDING_INVALID` before bootstrap. Raising required depth from `L0` to `L4` allows execution, but the evidence records `DEPTH_INSUFFICIENT` and `contributes=false`.

Case state	Associated?	Contributes?	Why
same ref, exact version/partition/binding, sufficient assurance	yes	yes	governed business evidence matches
same ref, wrong partition	yes in prose	no	`GOVERNANCE_BINDING_MISSING`
same names, changed expected semantics	yes	no	`GOVERNANCE_BINDING_INVALID`
exact binding, insufficient depth	yes	no	`DEPTH_INSUFFICIENT`
expired or digest-mismatched Oracle	no usable contribution	no	`ORACLE_GOVERNANCE_INVALID`

30.5 Follow one loan Case through the contribution filter

1. Scenario associates `auto-approve` with `LOAN-DECISION-001`.
2. Evaluation context selects version 3, `cn-retail`, release `2026.09`.
3. Binding finds the exact Suite, Case, and business Expectation digest.
4. Oracle catalog matches approved content and revision.
5. Verifier executes the Case and derives assurance from observed controls.
6. Only then does `verifyVersionedGoverned(Path)` emit a contribution.

The contribution is deliberately local to one Suite. It says what this Case supplied; it does not yet decide whether two required Cases across projects or modules exist.

30.6 Lab: keep the ref, break the meaning

1. Run the valid version-3 `cn-retail` Case and record its Requirement key and contribution.
2. Change only the partition context to EU. Confirm `GOVERNANCE_BINDING_MISSING` and zero bootstrap calls.
3. Restore context; change only expected `APPROVE` to `REVIEW` while keeping IDs. Confirm `GOVERNANCE_BINDING_INVALID`.
4. Restore semantics; increase `minimumDepth` beyond observed assurance. Confirm execution occurs but contribution is rejected with `DEPTH_INSUFFICIENT`.
5. Write a four-column review card: associated, evaluated, contributes, first reason.

30.7 Transfer the model

The same pattern applies to tax rules, medical eligibility, fraud holds, or account closure. Write one exact identity for a rule in your domain, then name one same-reference Case that must not contribute because its market, version, Oracle, or assurance differs.

30.8 Chapter boundary

- A Requirement reference is an association hint, not evidence of satisfaction.
- One Suite contribution cannot satisfy a cross-Suite minimum by itself.
- Owner labels and approved digests are governance facts, not authenticated human identity.
- A governance preflight failure means the experiment is invalid; it is not a customer decision verdict.

30.9 Experiment acceptance card

- **Expected and observed:** Requirement identity and semantics must both match to contribute.
- **Failure and recovery:** Keep the ref but change meaning; restore the exact version and semantic binding.
- **Proof boundary:** Proves the contribution filter, not owner approval of the Requirement.
- **Exercise contract:** One Requirement and case; change semantics only; deliver accepted/rejected results; stop when same-name/different-meaning is rejected.

30.10 Summary

Governance turns a familiar Requirement name into an exact obligation and filters passing Cases into valid contributions. Chapter 31 adds the missing completeness authority: one frozen cohort across Project and Reactor children, followed by a derived capability and explicit gaps.

Next: Chapter 31 — Project, Reactor, and Claim Capability

CHAPTER
31

Arc 6 — Business Promises

Chapter 31 — Project, Reactor, and Claim Capability

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

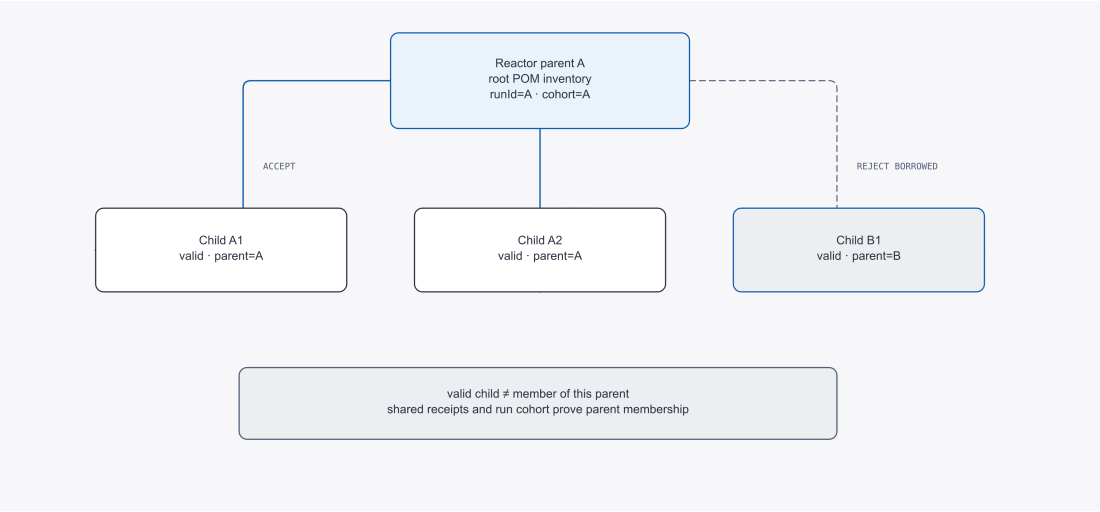
Chapter promise: *Aggregate exact Requirement contributions only from a complete frozen cohort, reject borrowed children, and end with the highest evidence-backed decision capability plus named gaps.*

31.1 Learning goals

1. Explain why aggregation needs an authoritative inventory and one run cohort.
2. Distinguish Project completeness from direct-module Reactor completeness.
3. Observe cohort mismatch and borrowed-child rejection before trusting totals.
4. Treat `ClaimCapability` and `missingEvidence` as derived reader output.

31.2 Two valid children can still make an invalid parent

The decision module contributes one valid loan Case; the funding module contributes another. Together they meet `minimumPassingCases: 2`—but only if both belong to the same governance snapshot, selection, source commit, and parent run.



This is the aggregation version of a cut-and-paste report problem. Child validity is necessary; parent membership is a separate fact.

31.3 Project aggregation freezes discovery before execution

```
VerificationGovernedProjectEvidence evidence =
        VerificationProjectRunner.runGovernedEvidence(
                scenarioDirectory,
                verifier,
                VerificationSelection.all(),
                projectIdentity,
                outputDirectory);
```

The runner freezes Scenario discovery, effective selection, governance context, project POM identity, and source-bound child receipts before calculating the Requirement. If a required binding is absent or filtered out, it reports `GOVERNANCE_BINDING_MISSING` or `GOVERNANCE_BINDING_FILTERED`; it does not lower the minimum to fit the available children.

RC1 test `aggregatesDistinctSuiteCasesAndRejectsAMissingGovernedChildBeforeBootstrap` protects this distinction: separate Suite contributions may combine only through the Project aggregate that owns their complete inventory.

31.4 Reactor adds a different completeness authority

`VerificationReactorRunner.runGovernedEvidence(...)` reads the root `pom.xml` direct `<modules>` list. Every module supplied to the runner must match that inventory, literal GAV, Suite/Case inventory, shared Git commit, selection, and governance cohort.

In `recomputesOneGlobalRequirementFromTwoIndependentModules`, either module alone returns `INVALID / GOVERNANCE_BINDING_MISSING` with zero calls. The Reactor run executes both once, lists two qualified Suite identities, and recomputes one Requirement with two satisfied Cases.

Current Preview covers direct modules in one Git worktree. Nested reactors, arbitrary Maven profiles, external-parent effective models, and deployment topology remain outside this completeness claim.

31.5 Attack the cohort before reading the count

The test `rejectsAReactorGovernanceCohortMismatchBeforeCustomerBootstrap` changes only the second module's `catalogId`. The result is:

```
status      = INCOMPLETE
reasonCodes = [REACTOR_GOVERNANCE_COHORT_MISMATCH]
```

```
module calls = 0 + 0
```

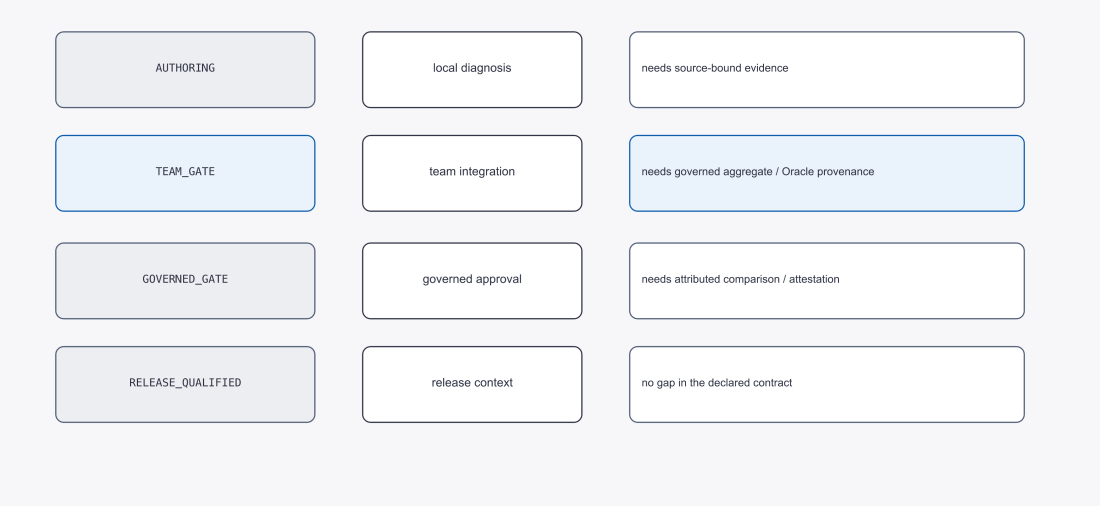
The runner rejects the parent before customer bootstrap. This is stronger than discovering a mismatch after two expensive or side-effecting executions.

Another attack runs two valid Reactor parents, then replaces one Suite in the first in-memory parent with a valid Suite from the second. `reactorClaimRejectsAValidChildBorrowedFromAnotherParentRun` drops capability to `AUTHORING` and restores `SOURCE_BOUND_PROJECT_AGGREGATE` as a `TEAM_GATE` gap. Valid child plus wrong parent still equals invalid aggregate.

31.6 Let Claims name the remaining decision boundary

```
VerificationClaimAssessment assessment =
        VerificationClaims.assess(evidence, projectRoot);

ClaimCapability current = assessment.capability();
Map<ClaimCapability, List<VerificationEvidenceGap>> gaps =
        assessment.missingEvidence();
```



Capability	Evidence-backed decision context	Typical remaining gap
`AUTHORING`	local diagnosis and witness discovery	source-bound Suite/Project evidence
`TEAM_GATE`	team integration with continuity, replay, and CI-safe controls	governed aggregate or Oracle provenance
`GOVERNED_GATE`	versioned business obligations and trusted Oracle provenance	attributed comparison and release attestation
`RELEASE_QUALIFIED`	exact governed evidence and comparison signed for release	no missing item in the declared release contract

Gaps are cumulative. A string written into JSON cannot promote the evidence, and a Policy cannot award itself a capability. A trustworthy `FAIL` may have strong provenance; capability still does not replace business status.

31.7 Lab: borrow one child

1. Run a two-module parent and retain its external receipt.
2. Run the same modules again into a second parent directory.
3. Confirm each parent independently verifies.
4. Replace one child projection in parent A with the valid child from parent B.
5. Assess the mixed parent. Record `AUTHORING` and `SOURCE_BOUND_PROJECT_AGGREGATE`.

6. Restore parent A and draw the ownership links: root POM → parent run ID → child receipt → Suite contribution.
7. End with a gap card: current capability, desired capability, missing evidence, owner, and next experiment.

31.8 Transfer: a multi-module billing release

Imagine tax, invoice, and ledger modules each publish valid child evidence. Combining a tax child from yesterday's parent with today's invoice and ledger children creates a readable but mixed cohort. Require one root build identity, one parent run, and owned child receipts before claiming project completeness; otherwise stop at `AUTHORING` and name the missing owner.

31.9 Chapter boundary

- Valid child evidence does not prove membership in a particular parent run.
- Project completeness does not imply direct-module Reactor completeness or deployment completeness.
- A Reactor total is meaningful only after inventory and cohort checks.
- `ClaimCapability` states the supported decision context; it neither changes the verdict nor approves a release.

31.10 Experiment acceptance card

- **Expected and observed:** Two valid children can still leave a parent capability gap.
- **Failure and recovery:** Borrow a child from another cohort; restore the frozen discovery set.
- **Proof boundary:** Proves Project/Reactor completeness, not acceptance of residual risk.
- **Exercise contract:** One parent and two children; replace one child; deliver the gap; stop when the borrowed child is identified.

31.11 Summary

Aggregation is not addition. It is addition under an authoritative inventory and one frozen cohort. The Claims reader then reports how far that evidence may travel and what still blocks the next decision. Chapter 32 runs the last controlled experiments: one DSL change and one BLOGE toolchain change.

Next: Chapter 32 — Change Attribution, Maven, and Release Decisions

CHAPTER
32

Arc 6 — Business Promises

Chapter 32 — Change Attribution, Maven, and Release Decisions

Controlled scenarios, source-bound evidence, governance, and attribution

Goal: prove an approved business promise without overstating the evidence.

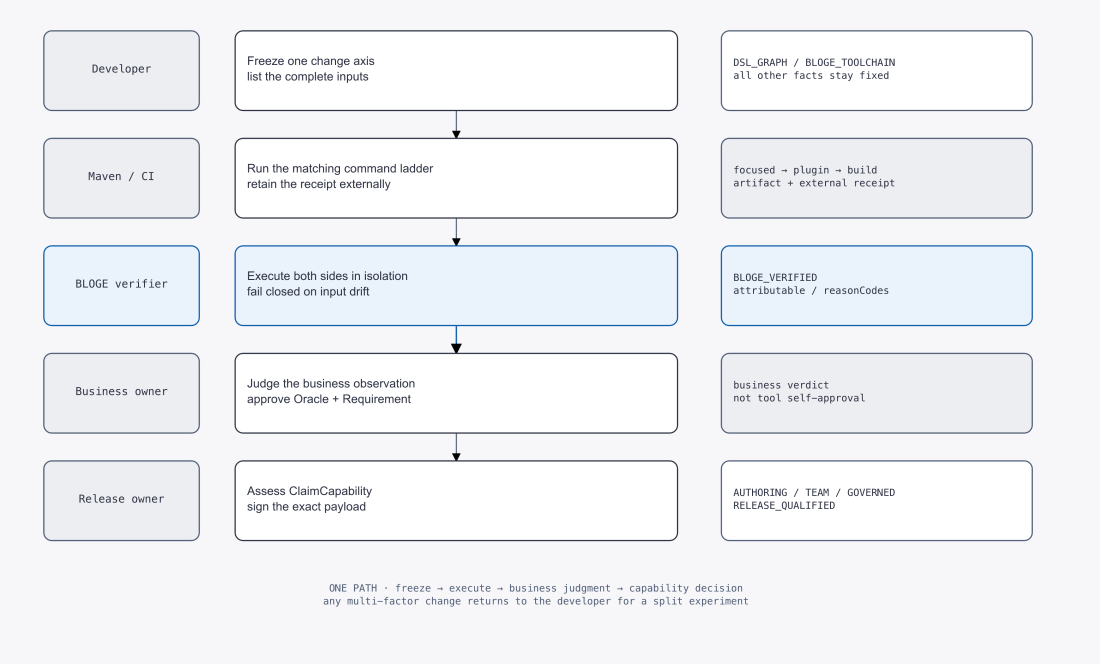
Chapter promise: *You will isolate one change axis, run the smallest honest Maven ladder, and decide whether the resulting evidence supports local diagnosis, a team gate, governed approval, or a release decision.*

32.1 Learning goals

1. Separate a DSL graph change, a BLOGE toolchain upgrade, and a business-code change.
2. Distinguish an observed difference from a difference that BLOGE can attribute.
3. Choose Maven commands by the claim you need, not by how green the console looks.
4. Preserve the receipts and external attestation needed for a release decision.

32.2 The upgrade changed the answer—but what caused it?

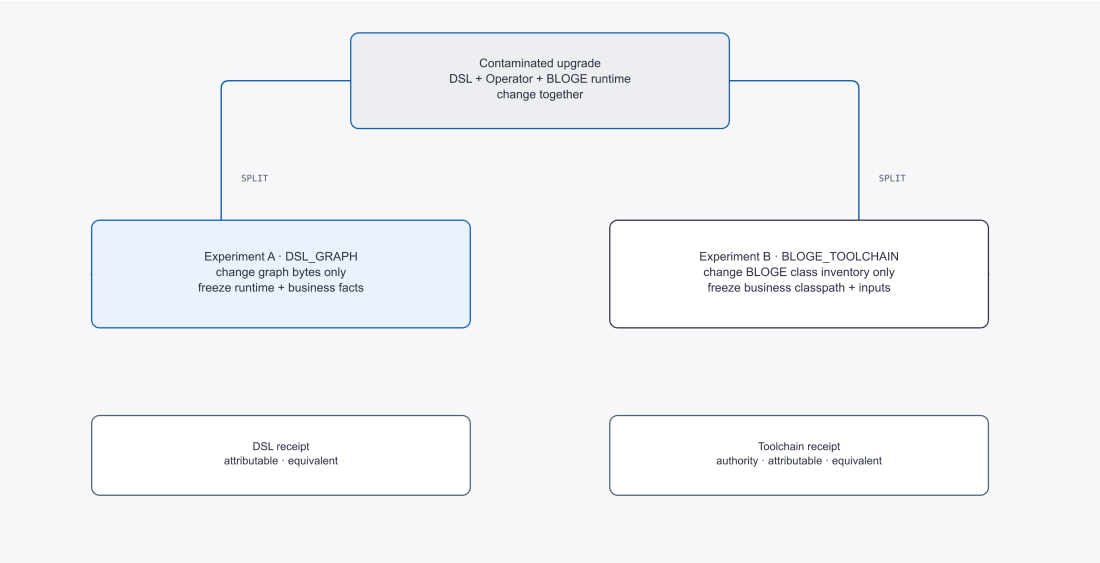
The loan team upgrades to BLOGE `0.9.8-RC1`. The boundary application now enters manual review instead of auto-approval. During the same branch, someone also edits the risk Operator. A before/after test can show a difference, but it cannot assign that difference to either change.



Attribution requires a controlled experiment: change one declared axis and freeze every material non-axis fact. If the Operator and runtime change together, stop at diagnosis, split the experiment, and run two comparisons.

32.3 Split the upgrade into two receipts

"Upgrade branch" is not a valid change axis. It mixes source edits, graph edits, dependency resolution, and runtime bytes. The loan team therefore creates two experiments from one common baseline:



Experiment	The only changed fact	Facts that must stay frozen	Honest result
A — `DSL_GRAPH`	baseline graph → candidate graph	BLOGE runtime, Scenario, Policy, Fixture, bootstrap, registry, profile	graph difference is attributable, or `COMPARISON_INPUT_DRIFT`
B — `BLOGE_TOOLCHAIN`	baseline BLOGE class inventory → RC1 inventory	business classpath, graph and declared inputs, Java launcher/protocol	attributable only with `BLOGE_VERIFIED` observations

Experiment A in `attributesAnObservedBusinessRegressionToTheDslGraphAxis` changes only the candidate graph's decision input. Both sides execute once; the candidate report is `FAIL`, the change code includes `OUTPUT_DIGEST_CHANGED:approve-low-risk/decision`, the comparison is attributable, and it is not equivalent.

Its control `attributesOnlyDslGraphChangeWhenEveryFrozenExecutionFactMatches` changes graph bytes while preserving the observed contract. It is both attributable and equivalent. Thus `attributable` and `equivalent` are independent questions.

Experiment B in `blogeOwnedChildExecutesTheFrozenScenarioBeforeAttributingTheToolchainAxis` keeps business classes and declared inputs fixed, changes the runtime inventory, and receives `BLOGE_VERIFIED` observations from two terminated child JVMs. The result is attributable, observationally equivalent, equivalent, and published as `SOURCE_BOUND`. Copying that artifact outside its bound project location produces `COMPARISON_SOURCE_BINDING_DRIFT`.

Now attack one frozen fact: `businessClasspathMutationIsReportedAsInputDrift` changes a business classpath marker during the toolchain experiment. The result becomes `COMPARISON_INPUT_DRIFT` and `attributable=false`. The runtime upgrade may still be interesting, but this run can no longer say it caused the observation.

32.4 Compare one DSL graph axis

For a graph-only change, both executions reuse one canonical Scenario and the verifier's source-bound environment:

```
ComparisonPlan plan = ComparisonPlan.dslGraph(
        "loan-risk-threshold-v3",
        scenarioPath,
        baselineGraphPath,
        candidateGraphPath);

VerificationAttributedComparisonEvidence evidence =
        verifier.compareEvidence(plan);
```

`compareEvidence(...)` publishes only when the `DSL_GRAPH` axis is attributable. It freezes the Scenario, Policy, Fixture, selected Cases, execution profile, bootstrap, Operator registry, service identity, governance inputs, and Git source binding. A drifted non-axis fact produces `COMPARISON_INPUT_DRIFT` and leaves only an in-memory diagnostic.

`attributable()` answers whether the observed change belongs to the declared axis. `equivalent()` is stricter: attribution must hold, both sides must pass, and their payload-free observations must match. Two equal outputs do not prove the candidate is better; they only report no observed regression within this contract.

32.5 Compare two BLOGE toolchains with frozen business code

A runtime upgrade needs two terminated child JVMs and one shared business runtime:

```
ToolchainBusinessRuntime business = ToolchainBusinessRuntime.of(
        businessClasspath,
        "com.acme.loan.LoanApprovalVerificationBootstrap");

ToolchainComparisonPlan plan = ToolchainComparisonPlan.blogeVerification(
        "bloge-0.9.7-to-0.9.8-rc1",
        inputs,
        business,
        ToolchainEndpoint.blogeVerifier("baseline", baselineToolchain),
        ToolchainEndpoint.blogeVerifier("candidate", candidateToolchain));

VerificationToolchainComparisonEvidence evidence =
        PairedExecutionCoordinator.builder()
                .projectRoot(projectRoot)
                .build()
                .compareEvidence(plan, outputRoot);
```

`inputs` must include `scenario`, `policy`, and every referenced DSL, Fixture, and Oracle file. Both children use the coordinator's Java launcher, the same ordered business classpath, and different BLOGE class inventories. The fixed BLOGE child reconstructs the project-relative files and runs a real `ScenarioVerifier`.

A custom `javaMain(...)` adapter is only `PROVIDER_ATTESTED`. It may report `observationallyEquivalent()`, but it cannot make `attributable()` or `equivalent()` true. Only `blogeVerifier(...)` gives the coordinator `BLOGE_VERIFIED` observations strong enough for attribution.

32.6 Run the Maven ladder that matches the question

1. **Focused story check — fast feedback.** Run the three loan starter tests. Eleven passing tests cover the selected story, not the whole Reactor.
   ```
   mvn -pl bloge-starter-loan-approval -am \
     -Dtest='LoanApprovalApplicationTest,LoanApprovalBusinessScenarioTest,LoanApprovalOperatorFlowTest' \
     -Dsurefire.failIfNoSpecifiedTests=false test
   ```
2. **Install the Preview plugin from a source checkout when needed.** If `0.9.8-RC1` is not available from your Maven repository, install it from the same source snapshot. `invoker.skip` skips only the plugin's own integration fixtures; it does not replace the complete Reactor check below.
   ```
   mvn -pl bloge-maven-plugin -am install \
     -DskipTests -Dinvoker.skip=true
   ```
3. **Publish customer-module verification artifacts — Preview.** In the customer module, keep compilation and the goal in one Maven invocation. The goal reads `.blogeverify.yaml` and writes the configured report directory.
   ```
   mvn test-compile \
     com.leanowtech.bloge:bloge-maven-plugin:0.9.8-RC1:verify
   ```
   A `PASS` here is still bounded by the config, source binding, receipt custody, and evidence capability. Keep the receipt outside the mutable artifact directory.

   > ***Known RC1 boundary:*** *this book's loan starter places its bootstrap under `com.leanowtech.bloge.starter.*`. The `0.9.8-RC1` plugin classloader mistakes that prefix for plugin-owned code, so the goal fails closed for this starter with `TEST_CLASSPATH_UNAVAILABLE`. This is not a Scenario failure; the runnable starter entry point is the JUnit check in step 1. Plugin Invoker fixtures cover Plain Java and Spring bootstraps in ordinary customer namespaces. Until the classloader boundary is fixed, do not present the starter as a successful artifact-publication example.*

4. **Check the complete Java build.**
   ```
   mvn clean install
   ```
   This checks the Reactor build; it does not create missing business Requirements or approve an Oracle. `verify-reactor` is a separate Preview path that requires schema-v2 per-module governance configuration and direct modules in one Git worktree. The loan starter configuration in this book does not claim that Reactor evidence.

32.7 Turn evidence into a decision

Read the highest supported `ClaimCapability`, then keep its boundary visible:

- **Local diagnosis — `AUTHORING`:** the comparison is not source-bound, one axis is not isolated, or required continuity evidence is absent. Fix the experiment before discussing causality.
- **Team gate — `TEAM_GATE`:** source binding, Policy continuity, replay assurance, and CI-safe execution are present. This supports team integration, not governed business approval.
- **Governed approval — `GOVERNED_GATE`:** trusted Project evidence also satisfies versioned Requirements and required Oracle provenance.
- **Release decision — `RELEASE_QUALIFIED`:** governed Project evidence, an exact attributable comparison receipt, and a trusted external release attestation bind the same change axis and Oracle closure.

An external signature proves that its signer endorsed exact digests. It does not prove undeclared behavior, production compatibility, or that the business policy itself is wise.

32.8 Lab: split a contaminated upgrade

1. Keep one Scenario, Policy, Fixture set, bootstrap, and source-bound verifier. Compare only `baseline-risk.bloge` with `candidate-risk.bloge`.
2. Record `changeAxis()`, `attributable()`, `equivalent()`, reason codes, and the external receipt digest.
3. Restore the DSL. Freeze the business classpath and compare two different BLOGE toolchain inventories with `blogeVerification(...)`.
4. Now change both the risk Operator and BLOGE runtime. Confirm this is not a valid single-axis plan; do not label the observed difference a runtime regression.
5. Split the contaminated change into a business-code experiment and a toolchain experiment. Ask the business owner to judge the changed business observation.
6. Assess the available governed and comparison evidence. State the highest supported capability and every remaining gap before a release owner signs anything.

Finish with two independent receipts and one gap card, not one green console:

```
DSL receipt       axis · baseline digest · candidate digest · attributable · equivalent
Toolchain receipt axis · runtime digests · business-classpath digest · authority · attributable
Release gap card  current capability · missing evidence · owner · next action
```

32.9 Transfer: payment-gateway upgrade attribution

When a payment gateway SDK and retry policy change in the same release branch, a higher success rate cannot be attributed to either one. First freeze business code and compare only SDK inventories; then restore the SDK and compare only the retry policy. Keep separate receipts and let the release owner decide from the two bounded results instead of one contaminated dashboard.

32.10 Chapter boundary

- A semantic difference is not automatically an attributable difference.
- `PROVIDER_ATTESTED` protocol correctness is not BLOGE-owned execution authority.
- `mvn clean install` is build evidence, not Requirement satisfaction.
- An attributable comparison alone remains `AUTHORING`; release qualification needs governed Project evidence and trusted external attestation.
- No comparison in this chapter proves correctness outside the declared Scenario, Oracle, inputs, and runtime boundary.

32.11 Experiment acceptance card

- **Expected and observed:** An upgrade becomes DSL_GRAPH and BLOGE_TOOLCHAIN single-axis receipts.
- **Failure and recovery:** Change classpath and toolchain together; freeze one axis and compare separately.
- **Proof boundary:** Proves attribution for fixed inputs, not release readiness.
- **Exercise contract:** Before/after versions; vary one axis at a time; deliver two receipts; stop when each difference has one candidate cause.

32.12 Summary

Freeze everything except one declared axis. Use `ScenarioVerifier.compareEvidence(...)` for a DSL graph and `PairedExecutionCoordinator.compareEvidence(...)` for a BLOGE-owned toolchain comparison. Then use the Maven ladder and `VerificationClaims` to report only the decision capability actually supported. Arc 6 ends where responsible release judgment begins: with evidence that is attributable, bounded, and independently reviewable.

Next: Arc 6 Recap — From Business Promise to Release Evidence.

CHAPTER

—

Arc 6 Recap — Business Evidence

Chapters 23–32 followed one promise: a loan decision must still match an owner-approved business contract after the graph, runtime, or environment changes.

0.1 The ten-chapter evidence path

1. **Frame the claim — Chapter 23.** Separate verdict, evidence trust, and decision capability.
2. **Run the first story — Chapter 24.** Map approved, rejected, and manual-review outcomes into Suite, Case, and Expectation.
3. **Control the inputs — Chapter 25.** Use Policy, Fixture, execution profile, and assurance to reject false greens.
4. **Keep time honest — Chapter 26.** Observe suspend, signal, resume, logical time, and temporal contracts.
5. **Kill weak mutants — Chapter 27.** Measure whether the suite notices selected, controlled business-rule changes.
6. **Search bounded domains — Chapter 28.** Turn properties into finite cases, retain counterexamples, and keep the Oracle boundary explicit.
7. **Bind the source — Chapter 29.** Connect artifacts to source, run cohort, seal, external receipt, and an independent reader.
8. **Contribute exactly — Chapter 30.** Decide whether a trusted Case fact contributes to one versioned Requirement.
9. **Aggregate one cohort — Chapter 31.** Derive Project/Reactor completeness and capability from an authoritative inventory, then name the remaining gaps.
10. **Attribute the change — Chapter 32.** Freeze one axis, compare, and report the highest supportable release capability.

0.2 Read every result on three axes

- `VerificationStatus`: what happened in this Case—`PASS`, `FAIL`, `INVALID`, or `INCOMPLETE`?
- `EvidenceTrust`: can an independent reader bind the result to the declared source and run inputs?
- `ClaimCapability`: does that evidence support only `AUTHORING`, or also `TEAM_GATE`, `GOVERNED_GATE`, or `RELEASE_QUALIFIED`?

Never collapse these axes into one green badge.

0.3 The release-owner scan

Before a release decision, point to each concrete item:

1. The active Requirement version, partition, binding, and approved Oracle closure.
2. A complete same-cohort governed Project aggregate.
3. Source-bound child receipts retained outside mutable artifact directories.
4. One attributable `DSL_GRAPH` or `BLOGE_TOOLCHAIN` comparison.
5. A trusted external attestation binding the exact governed receipt, comparison receipt, change axis, and Oracle closure.

6. The declared boundary: what the evidence still does not prove about production.

If one item is absent, keep the lower capability and name the gap. Do not negotiate the enum upward.

0.4 Final challenge: classify before you claim

A candidate build has passing loan Cases and a valid local seal. Its business Operator changed together with the BLOGE runtime. The report directory contains its receipt, no governed Project aggregate exists, and no release owner signed an attestation.

Classify it:

- **Verdict:** the declared Cases may be `PASS`.
- **Trust:** a local seal may establish byte integrity, but co-located custody and missing source/governance facts do not establish the release chain.
- **Attribution:** the two-factor change cannot identify a single cause.
- **Capability:** `AUTHORING`, with explicit source, governance, comparison, and attestation gaps.

Now split the two change axes, regenerate source-bound evidence, satisfy the governed Requirement, and retain the receipts externally. Only then ask the release owner to assess and sign the exact payload.

0.5 Keep the map

Return to Chapter 23 when claims blur, Chapter 24 for the runnable starter, Chapter 28 for counterexamples, Chapter 29 for trust boundaries, Chapter 30 for exact Requirement contribution, Chapter 31 for aggregation and capability limits, and Chapter 32 for comparison and release decisions. Then use Arc 7 to decide whether BLOGE belongs in the problem at all.

Learning Arc

Arc 7 — Selection and Capstone

Constraint-first selection and an operable graduation project

Goal: use BLOGE only when justified, then deliver a result others can challenge.

CHAPTER
33

Arc 7 — Selection and Capstone

Chapter 33 — Migration and Comparison

Constraint-first selection and an operable graduation project

Goal: use BLOGE only when justified, then deliver a result others can challenge.

Promise: *By the end of this chapter you will be able to state constraints before choosing BLOGE, name at least three cases where BLOGE should not be chosen, and complete a short ADR with evidence, a counterfactual, and a first migration probe.*

33.1 Learning Goals

1. Restate a tool-selection question as deployment, ownership, recovery, verification-governance, and ecosystem constraints.
2. Eliminate unsuitable candidates with hard stops instead of totaling feature check marks.
3. Separate "the feature exists," "the team can operate it," and "the evidence qualifies a release."
4. Design a reversible first migration step instead of using a full rewrite to test the choice.
5. Write a short ADR containing the choice, evidence, counterfactual, migration probe, and exit criteria.

33.2 Prerequisites

- Chapter 20 — Spring and Production Wiring — you need the `bloge-spring` wiring model to understand how BLOGE embeds into a running application.
- Chapter 13 — Durable Execution — checkpoint stores, suspend/resume, and recovery.
- Chapter 21 — Scheduling and Complexity in One JVM — engine statelessness, virtual-thread scheduling, and sharding.

33.3 Source Examples

File	What it shows
`BpmnToDslExample.java`	End-to-end: translate BPMN XML → `.bloge` DSL → compile → execute
`bloge-bpmn-transformer/README.md`	Translation pipeline, supported scope, diagnostics, and known limitations
`order-process.bloge`	Canonical order-processing graph — parallel fetch, branch, resilience

File	What it shows
`OrderProcessingDslExample.java`	DSL compilation, registry wiring, and execution from Java
`order-saga.bloge`	Saga-style compensation: `compensate` on each node
`payment-wait.bloge`	Long-running await with event correlation and timeout
`batch-order-processing.bloge`	`foreach` parallel fan-out over a collection
`SpringTicketTriageService.java`	Injecting `GraphEngine` and `List<Graph>` into a Spring service
ADR-001	Core zero-dependency decision
ADR-002	Virtual threads over CompletableFuture
ADR-004	HCL-inspired DSL over YAML/JSON

33.4 Why This Matters

You have spent sixteen chapters learning how to build, test, and run BLOGE workflows. Now a harder question arrives: **should you use BLOGE for this particular project?**

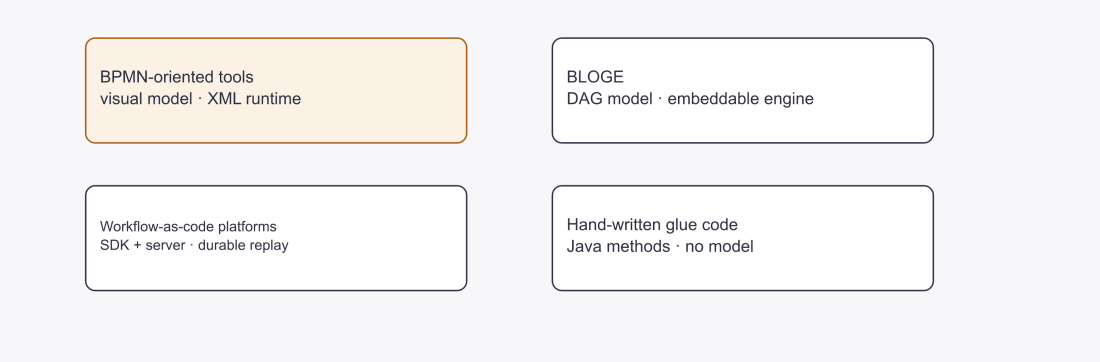
Teams rarely start with a blank slate. They have:

- Existing service methods with embedded orchestration logic.
- Spring Batch jobs handling step-oriented batch processing.
- BPMN diagrams authored in a visual modeller.
- Or they are evaluating workflow-as-code platforms that require a separate server cluster.

This chapter gives you a structured way to compare what BLOGE actually does against what those alternatives actually do — grounded in architectural properties you can point to in the repository, not marketing claims.

33.5 Mental Model

Think of orchestration tools as sitting on two axes:



Vertical axis — workflow shape visibility. The top half captures the workflow structure in a model (XML, DSL, or visual graph). The bottom half embeds structure entirely in code.

Horizontal axis — deployment topology. The left requires a separate server cluster. The right embeds into your existing application process.

BLOGE sits in the **upper-right quadrant**: a declarative DAG model that runs as a library inside your JVM, with no external infrastructure required by default.

This is not universally best — it is a specific set of trade-offs:

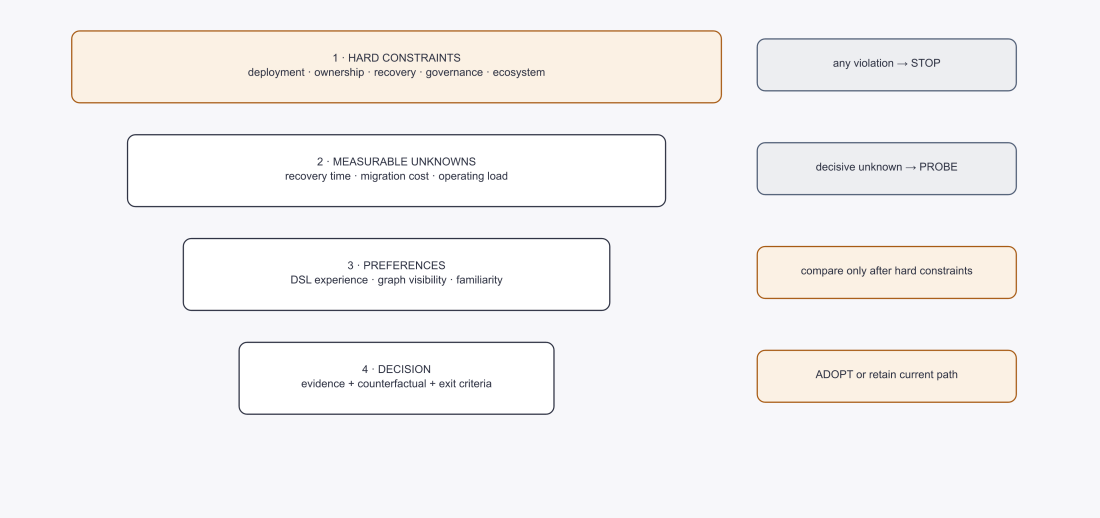
Property	BLOGE's choice	Where it is documented
Zero external dependencies in core	`bloge-core` depends on `java.base` alone	ADR-001
Virtual-thread execution	Each node runs in its own virtual thread	ADR-002
HCL-inspired DSL	Block-oriented, expression-capable, comment-friendly	ADR-004
Checkpoint-based durability	Opt-in via `bloge-durable`; stores are pluggable	Chapter 13, README
Embeddable engine	`GraphEngine` is a stateless library object	Chapter 21
Resilience per node	Retry → Timeout → Fallback chain, per node	ADR-007

Three orchestration modules expand the platform beyond graph-only orchestration:

Module	What it adds
`bloge-agent-ext`	LLM-driven agent loops with `agent` DSL, tool dispatch, memory strategies, and streaming
`bloge-session-durable`	`DurableSessionManager` — crash-safe multi-turn sessions with checkpoint persistence
`bloge-state-durable`	`DurableStateMachineManager` — crash-safe state machines with checkpoint persistence

33.6 Constraints Before Comparisons

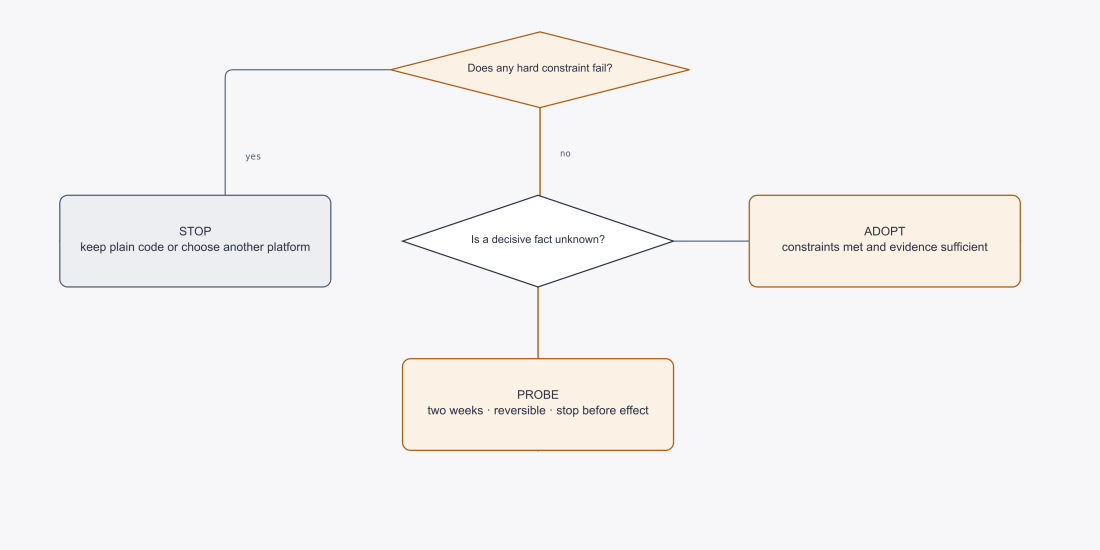
The loan platform has just completed business-correctness verification. The architecture board is not asking "how many features does BLOGE have?" It is asking: **under the project's non-negotiable constraints, can BLOGE become an operable, verifiable, and reversible choice?**



Write hard constraints first, evaluate preferences second, and design a migration probe last. A candidate that violates a hard constraint stops; other feature scores cannot average the conflict away.

Constraint	Loan-platform question	BLOGE fact	Decision meaning
Deployment topology	Can the deployment add a workflow cluster?	Engine embeds in the JVM; durable stores are optional	Still a candidate when new clusters are forbidden
Model ownership	Who reviews and changes the flow?	`.bloge` is an explicit, code-friendly DAG	A conflict when the business must own BPMN directly

Constraint	Loan-platform question	BLOGE fact	Decision meaning
Recovery semantics	Does recovery require checkpoints or full history replay?	BLOGE checkpoints; it is not a general history-replay platform	Stop when replay is mandatory
Verification governance	Must releases bind approved Scenarios, evidence, and source identity?	RC1 provides Scenario/Policy/Fixture, source-bound evidence, and claims	Supports the gate, but still needs organisational owners
Ecosystem boundary	Must workers run natively in several languages?	The primary runtime is JVM-based	Stop when polyglot workers are mandatory



This matrix does not produce a fake score such as "BLOGE: 82." It produces `STOP`, `PROBE`, or `ADOPT`. `PROBE` means a decisive fact is unknown and must be measured; it is not a polite pass.

33.7 Three Clear Reasons Not to Choose BLOGE

1. **A non-development team directly owns BPMN models.** If visual collaboration, human-task inboxes, and the existing process server are organisational contracts, migration breaks ownership; it does not merely replace syntax.
2. **The recovery contract requires complete history replay.** When audit or disaster recovery must reconstruct every step from event history, checkpoint recovery cannot substitute for replay merely because both can resume work.
3. **Polyglot workers are a hard constraint.** If one orchestration must natively dispatch Go, Python, and TypeScript workers, an embedded JVM model transfers the adapter cost to the team.

There is also a simpler stop: when the flow is three sequential calls with no waiting, branching, independent recovery, or shared governance need, a normal method is cheaper. Architectural capability must pay for itself by reducing real complexity.

33.8 A Completed Loan-Platform ADR

The loan platform forbids a new standalone cluster, runs on the JVM, and requires every release to bind approved Scenarios to source evidence. Full history replay and polyglot workers are not current requirements. The decision is therefore `PROBE BLOGE`, not immediate fleet-wide adoption.

ADR field	This decision
Choice	Run a two-week reversible probe on the `auto-approve` path

ADR field	This decision
Evidence	JVM embedding, explicit DAGs, RC1 verification, and source-bound evidence meet hard constraints
Counterfactual	If history replay or non-JVM workers become mandatory, choose a server-based workflow platform
First migration step	Move read-only risk checks into operators; leave the disbursement effect on the old path
Exit criteria	Stop expansion if the graph contract, business Scenario, or recovery drill fails

The ADR does not "approve BLOGE." It turns the next step into a falsifiable experiment. A green graph test cannot replace a business Scenario, and a `PASS` cannot replace source binding and release governance.

33.9 First Working Example

The most common migration scenario is replacing a hand-written Java orchestration method with a BLOGE graph. The migration below proceeds step by step.

33.9.1 Before: hand-written glue code

```java
// Typical procedural orchestration buried in a service method
public OrderResult processOrder(String userId, List<String> productIds) {
    // 1. Sequential — but these two calls are independent
    User user = userService.fetchUser(userId);
    List<Product> products = productService.fetchProducts(productIds);

    // 2. Retry logic tangled with business code
    BigDecimal total = null;
    for (int attempt = 0; attempt < 3; attempt++) {
        try {
            total = pricingService.calculate(user, products);
            break;
        } catch (Exception e) {
            if (attempt == 2) throw e;
            Thread.sleep(100 * (attempt + 1));
        }
    }

    // 3. Branch logic buried in an if-statement
    CreditResult credit = creditService.check(user.id(), total);
    if (credit.approved()) {
        return orderService.create(user, total);
    } else {
        return orderService.reject(user.id(), credit.reason());
    }
}
```

Problems:

- **Steps 1 and 2 are independent** but execute sequentially.
- **Retry logic is hand-rolled** and mixed into the business method.
- **No one can see the workflow shape** without reading every line.

33.9.2 After: BLOGE graph

The same workflow, expressed as `order-process.bloge`:

The 46-line graph stays whole because the comparison needs one copyable unit that exposes parallel fetch, fan-in, resilience, and both branch targets.

```
graph orderProcess {

  node fetchUser : FetchUserOperator {
    input { userId = ctx.userId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchProducts : FetchProductsOperator {
    input { productIds = ctx.productIds }
    timeout = 5s
  }

  node calcPrice : CalcPriceOperator {
    depends_on = [fetchUser, fetchProducts]
    input {
      user     = fetchUser.output
      products = fetchProducts.output
    }
  }

  node checkCredit : CreditCheckOperator {
    depends_on = [fetchUser, calcPrice]
    input {
      userId = fetchUser.output.id
      amount = calcPrice.output.total
    }
    retry = { attempts: 3, backoff: 100ms, strategy: jitter }
    fallback = { approved: false, reason: "credit service unavailable" }
  }

  branch on checkCredit.output.approved {
    true  -> createOrder
    false -> rejectOrder
  }

  node createOrder : CreateOrderOperator {
    depends_on = [calcPrice]
    input { user = fetchUser.output, price = calcPrice.output }
  }

  node rejectOrder : RejectOrderOperator {
    depends_on = [checkCredit]
    input { userId = fetchUser.output.id, reason = checkCredit.output.reason }
  }
}
```

What changed:

- `fetchUser` and `fetchProducts` **run in parallel** — the engine sees they have no dependency on each other.
- Retry and timeout are **declared on the node**, not coded in a loop.
- The branch is **a first-class routing decision**, not a hidden `if`.

- The shape is **visible, reviewable, and toolable**.

The same registry/engine wiring looks like this when the DSL lives in a file:

```
var registry = new DefaultOperatorRegistry();
registry.register("FetchUserOperator", fetchUserOp);
registry.register("FetchProductsOperator", fetchProductsOp);
// … register remaining operators …

Graph graph = new GraphLoader(registry).load(
        Path.of("bloge-examples/src/main/resources/bloge/order-process.bloge"));
GraphEngine engine = GraphEngine.builder().registry(registry).build();
GraphResult result = engine.execute(graph, new GraphContext(Map.of(
        "userId", "user-42",
        "productIds", List.of("prod-1", "prod-2", "prod-3")
)));
```

33.10 Break It Apart

33.10.1 BLOGE vs hand-written orchestration

Dimension	Hand-written glue code	BLOGE graph
Concurrency	You manage threads, pools, futures	Engine schedules virtual threads from the DAG shape (ADR-002)
Resilience	Hand-rolled retry loops, try/catch fallbacks	Declared per node: `retry`, `timeout`, `fallback` (ADR-007)
Visibility	Read the code; hope the comments are accurate	The `.bloge` file *is* the documentation; Studio renders it visually
Testability	Mock every service call in each test	`MockOperator`, `GraphTestRunner`, `DslTestHelper` from `bloge-test`
Durability	Build your own checkpoint/recovery	Opt-in via `bloge-durable` stores (Chapter 13)

When hand-written code is fine: If the workflow is three sequential calls with no retry, no branching, and no durability need, a plain method is simpler. BLOGE adds value when the orchestration shape becomes complex enough that you need concurrency, resilience, or visibility.

33.10.2 BLOGE vs step-oriented batch frameworks

Step-oriented batch frameworks (like Spring Batch) model work as a **linear pipeline of steps** with chunk-oriented reading, processing, and writing.

Dimension	Step-oriented batch	BLOGE graph
Execution model	Step 1 → Step 2 → Step 3, linear	DAG: independent nodes run in parallel automatically
Iteration	Chunk-oriented reader/processor/writer	`foreach` with parallel or sequential mode (`batch-order-processing.bloge`)
Resilience	Skip/retry policies at the step or chunk level	Per-node retry → timeout → fallback chain
Long-running waits	Not a primary concern	`await` with event correlation and timeout (`payment-wait.bloge`)
Dependencies	Typically requires the framework runtime	`bloge-core` has zero external dependencies (ADR-001)

When step-oriented batch is a better fit: If your workload is pure ETL — read chunks, transform, write — with no fan-out, no branching, and no external-event waits, a batch framework is purpose-built for that pattern. BLOGE adds value when batch steps need to fan out in parallel, wait for external signals, or share orchestration patterns with non-batch workflows.

33.10.3 BLOGE vs BPMN-oriented tools

BPMN-oriented tools model workflows in a visual XML-based notation, typically backed by a separate execution server.

Dimension	BPMN-oriented tools	BLOGE graph
Workflow notation	BPMN 2.0 XML (verbose, machine-oriented)	`.bloge` DSL (expression-capable, comment-friendly; ADR-004)
Gateways	Explicit fork/join gateway elements	Implicit from DAG shape: no gateway nodes needed
Deployment	Separate process engine server	Embedded library — `GraphEngine` runs in your JVM
Migration path	—	`bloge-bpmn-transformer` translates BPMN XML → `.bloge` DSL (README)

The `bloge-bpmn-transformer` module provides a concrete migration path. It reads BPMN 2.0 XML, lowers it through an intermediate representation, and outputs either `.bloge` DSL text or an executable `Graph` object.

Supported BPMN elements include: start/end events, service/script/user tasks, exclusive, parallel, and inclusive gateways, timer/message/signal events, call activities, and subprocess structures. Unsupported elements produce diagnostics instead of silently inventing semantics.

BPMN inclusive gateway support. The BPMN transformer supports **inclusive gateways** (translated to `branch mode=inclusive`), **user task metadata** preservation (`assignee`, `candidateGroups`, `formKey`), **extension element** parsing (`<inputOutput>` parameters and `<properties>`), and improved **UEL-to-BLOGE expression translation** (safe methods like `toString()`, `isEmpty()` no longer trigger `COMPLEX_EXPRESSION` diagnostics).

Translation example from `BpmnToDslExample.java`:

```
BpmnTranslator translator = new BpmnTranslator(OperatorMappingConfig.EMPTY);
try (InputStream bpmn = getClass().getResourceAsStream("/bpmn/order-process.bpmn")) {
    TranslationResult<String> dsl = translator.translateToDsl(bpmn);
    if (dsl.hasErrors()) {
        throw new IllegalStateException(dsl.diagnostics().toString());
    }
    // dsl.result() is valid .bloge DSL text
    Graph graph = new GraphLoader(registry).load(dsl.result());
}
```

Diagnostic code	Meaning
`UNSUPPORTED_ELEMENT`	BPMN construct the translator does not support yet
`UNMAPPED_OPERATOR`	No operator mapping matched a task
`COMPLEX_EXPRESSION`	UEL expression could not be translated one-to-one
`POTENTIAL_DATA_LOSS`	Translation possible, but some BPMN intent may not survive unchanged
`AMBIGUOUS_TIMER`	Timer definition uses a form that does not map cleanly to BLOGE `wait` syntax

When BPMN tools are a better fit: If your organisation's business analysts author and own the process models in BPMN, and the execution server is already deployed and operated, migrating to BLOGE may not be worth the disruption. BLOGE adds value when the developer team owns the workflow and wants a code-friendly authoring experience with an embeddable runtime.

33.10.4 BLOGE vs workflow-as-code platforms

Workflow-as-code platforms let you write workflows in a general-purpose language (Go, Java, TypeScript) with durability handled by a separate server cluster that replays function histories.

Dimension	Workflow-as-code platforms	BLOGE
Deployment	Requires a dedicated server cluster	Embeddable library; no external server required
Durability model	History replay / event sourcing	Checkpoint-based: includes node outputs, loop/foreach progress, suspend context, partial results, event correlation, and session snapshots (Chapter 14, built on Chapter 13)
Workflow shape	Implicit in code control flow	Explicit DAG model visible in `.bloge` files and Studio
Dependencies	Client SDK + server infrastructure	`bloge-core` is `java.base` only (ADR-001)
Concurrency	Server manages worker task dispatch	Engine uses virtual threads locally (ADR-002)
Compensation	Framework-specific saga patterns	Declarative `compensate` per node (`order-saga.bloge`)

When a workflow-as-code platform is a better fit: If you need built-in multi-language support across Go, TypeScript, and Python, or you already operate the server cluster and rely on its history-replay semantics, that platform's ecosystem advantage is real. BLOGE adds value when you want to avoid operating a separate cluster, when your team is JVM-native, or when the explicit DAG model and embeddable engine matter more than cross-language SDK support.

33.11 Common Trap

33.11.1 [X] Feature-list comparison instead of architecture-fit comparison

A common mistake is building a spreadsheet of features — "Does it have retry? Does it have branching?" — and picking the tool with the most checkmarks.

Every tool on this page supports some form of retry, branching, and persistence. The differences that matter are **architectural**:

- **Where does the engine run?** (embedded library vs. separate cluster)
- **How is the workflow shape expressed?** (explicit model vs. implicit in code)
- **How does durability work?** (checkpoints vs. replay)
- **What are the dependency and JDK requirements?** (zero-dep core vs. SDK + server)

If you compare at the wrong level, you'll pick a tool that has all the features but the wrong deployment model for your team.

33.12 Guided Rewrite

33.12.1 Migrating from hand-written code to BLOGE in four steps

Take any orchestration method in your codebase and apply this sequence:

Step 1 — Identify the independent calls.

Mark every pair of calls that have no data dependency on each other. In the hand-written example above, `fetchUser` and `fetchProducts` are independent.

Step 2 — Extract operators.

Each call becomes an `Operator<I, O>` implementation. Register them in a `DefaultOperatorRegistry`:

```
registry.register("FetchUserOperator", (input, ctx) -> {
    String userId = ((Map<String, Object>) input).get("userId").toString();
    return Map.of("id", userId, "name", userService.fetchUser(userId).name());
```

```
});
```

Step 3 — Declare the graph.

Write the `.bloge` file. Use `depends_on` for explicit fan-in. Use `branch on` for conditional routing. Add `retry`, `timeout`, and `fallback` on nodes that call external services.

Step 4 — Verify behaviour.

Use `GraphTestRunner` from `bloge-test` to assert node execution order, status, and outputs:

```
var runner = new GraphTestRunner(
        registry,
        Map.of(
                "FetchUserOperator", fetchUserOp,
                "FetchProductsOperator", fetchProductsOp
                // … remaining operators …
        )
);

var result = runner.execute(graph, new GraphContext(Map.of("userId", "u1")));
runner.assertNodeExecuted("fetchUser");
runner.assertNodeExecuted("fetchProducts");
runner.assertNodeSkipped("rejectOrder");   // credit was approved
```

33.12.2 Migrating from BPMN to BLOGE

Step 1 — Translate. Use `BpmnTranslator.translateToDsl()` with an operator mapping config.

Step 2 — Read diagnostics. Fix any `UNMAPPED_OPERATOR` or `COMPLEX_EXPRESSION` diagnostics. Map BPMN service task IDs to BLOGE operator names in `OperatorMappingConfig`.

Step 3 — Review the generated DSL. The translator generates `// Source: bpmn:serviceTask id="xxx"` comments for traceability (when `generateSourceComments` is enabled).

Step 4 — Compile and run.

```
Graph graph = new GraphLoader(registry).load(dsl.result());
GraphEngine engine = GraphEngine.builder().registry(registry).build();
engine.execute(graph, context);
```

33.13 Brain Check

1. What are the two axes in the mental model diagram, and where does BLOGE sit? *(Workflow shape visibility vs. deployment topology; BLOGE is in the upper-right: declarative model + embeddable library.)*
2. Name three architectural properties of BLOGE that are documented in ADRs and explain what each trades away. *(Zero-dep core → some convenience reimplemented in adapter modules. Virtual threads → requires modern JDK. HCL-inspired DSL → needs its own parser, not compatible with generic YAML/JSON tooling.)*
3. If you translate a BPMN file and get a `COMPLEX_EXPRESSION` diagnostic, what should you do? *(Manually rewrite the UEL expression as a BLOGE path expression or move the logic into an operator.)*
4. When would you choose a workflow-as-code platform over BLOGE? *(When you need multi-language SDK support across Go/TypeScript/Python, or you already operate the server cluster and rely on its replay semantics.)*

33.14 Lab

33.14.1 Lab — Write a Falsifiable Short ADR

Choose a flow in your own system with at least five steps. Deliver an ADR of no more than one page:

1. State three hard constraints and two preferences; each hard constraint must be decidable.
2. Compare at least a plain method, BLOGE, and one server-based candidate.
3. State one counterfactual under which BLOGE is not selected; "it depends" is not allowed.
4. Design a reversible, two-week migration probe with a bounded change scope and real-effect boundary.
5. Record a `STOP / PROBE / ADOPT` outcome and the evidence that would invalidate it.

Done means another reader can recover the choice, evidence, unknowns, first step, and stop condition from the ADR alone. A feature checklist is not sufficient. The reference answer and rubric are in `solutions/ch33/adr-review-guide.md`; do not open it first.

33.15 Bridge: from Change Attribution to Architecture Choice

This chapter consumes change-attribution evidence to make a choice; it does not turn design judgment into a controlled causal experiment. A comparison receipt can locate a changed result only when both runs share Scenario, Policy, Fixture, business runtime, and source cohort while changing exactly one of `DSL_GRAPH` or `BLOGE_TOOLCHAIN`. The ADR must still add organisational constraints, operating cost, and exit criteria. `compareEvidence(...)` cannot decide those for you.

33.16 Experiment acceptance card

- **Expected and observed:** The constraint funnel returns STOP, PROBE, or ADOPT without assuming BLOGE wins.
- **Failure and recovery:** Add a hard constraint that invalidates the choice; recover with a stop or reversible probe.
- **Proof boundary:** Proves a falsifiable selection, not universal fit.
- **Exercise contract:** Loan constraints; change one constraint; deliver a short ADR; stop when a counterfactual changes the decision and exit is explicit.

33.17 Recap

- BLOGE occupies a specific architectural position: **declarative DAG model
  - embeddable library engine** with zero external dependencies in the core.
- Compare tools on **architecture-fit** (deployment model, durability semantics, workflow visibility, dependency footprint), not feature checklists.
- **Hand-written code** is fine for simple sequences; BLOGE adds value when you need concurrency, resilience, or visibility in the orchestration shape.
- **Step-oriented batch** is purpose-built for ETL; BLOGE adds value when batch steps need fan-out, branching, or external-event waits.
- **BPMN tools** excel when business analysts own the model; BLOGE adds value when developers own the workflow and want code-friendly authoring. The `bloge-bpmn-transformer` provides a concrete migration path.
- **Workflow-as-code platforms** shine for multi-language, server-managed durability; BLOGE adds value when you want an embeddable, JVM-native engine without a separate cluster.
- Migration from hand-written code follows four steps: identify independence, extract operators, declare the graph, verify behaviour.
- A decision produces `STOP / PROBE / ADOPT`: stop on a violated hard constraint and probe when a decisive fact is unknown.
- A short ADR must contain the choice, evidence, counterfactual, first migration step, and exit criteria.

33.18 Next Step

In Chapter 34 — Labs you will work through end-to-end exercises that combine everything from the series — graphs, resilience, subgraphs, durability, observability, and the migration patterns from this chapter — into complete, runnable projects.

33.19 Reference Links

- Project README — module overview and quick start
- ADR-001 — Core Zero Dependencies
- ADR-002 — Virtual Threads over CompletableFuture
- ADR-004 — HCL-Inspired DSL
- ADR-007 — Resilience Order
- DSL Specification — full language grammar and compilation semantics
- Operator Design Specification — operator boundary and granularity rules
- `bloge-bpmn-transformer` README — BPMN translation pipeline and diagnostics
- `order-process.bloge` — canonical order-processing example
- `order-saga.bloge` — saga compensation example
- `payment-wait.bloge` — long-running await example
- `BpmnToDslExample.java` — BPMN → DSL → execute example
- `OrderProcessingDslExample.java` — DSL compilation and execution
- Getting Started — full setup guide

CHAPTER
34

Arc 7 — Selection and Capstone

Chapter 34 — Labs

Constraint-first selection and an operable graduation project

Goal: use BLOGE only when justified, then deliver a result others can challenge.

Promise: *By the end of this chapter you will have completed foundation, advanced, and graduation capstones. The final artifact includes not only a graph, but technical tests, a business Scenario/Policy, evidence interpretation, an architecture diagram, and a list of unproven claims.*

34.1 Learning Goals

1. Evolve a graph from a bounded starter instead of guessing syntax in an empty file.
2. Deliver separate evidence for the graph contract, business contract, and release boundary.
3. Run one-variable failure experiments for effects, waiting, recovery, and fallback.
4. Explain the runtime boundary in one architecture diagram and the evidence boundary in one receipt-reading diagram.
5. State what is proved, what is unproved, and what still needs human approval.

34.2 Prerequisites

All thirty-three earlier chapters. The labs assume you are comfortable with:

- Graph structure, dependencies, and data flow (Ch 1–Ch 4)
- Branches and resilience (Ch 5–Ch 6)
- Operator design and DSL authoring (Ch 7–Ch 8)
- Subgraphs, iteration, waiting, and durability (Ch 10–Ch 13)
- Orchestration patterns — sessions, state machines, and nested lifecycle ownership (Ch 14–Ch 16)
- Tooling and review feedback (Ch 9)
- Testing, production wiring, and scale (Ch 18–Ch 22)
- Business correctness, evidence, and release claims (Ch 23–Ch 32)
- Migration thinking (Ch 33)

34.3 Source Examples

File	What it shows
`order-process.bloge`	Foundation start — fan-out, fan-in, branch, resilience
`ticket-routing.bloge`	Advanced start — sentiment analysis and multi-way branch
`loan-approval.bloge`	Graduation start — loan business and verification assets
`batch-order-parallel.bloge`	Optional batch exercise — parallel `foreach`
`status-polling.bloge`	Optional loop exercise — `until` and `loopIteration`
`payment-wait.bloge`	Optional await reference — correlation and timeout
`order-saga.bloge`	Graduation reference — saga compensation
`GraphTestRunner.java`	Test harness used in every lab's verify step
`MockOperator.java`	Test doubles — `returning`, `throwing`, `delaying`, `recording`
`DslTestHelper.java`	DSL compile + `expectParseError` / `expectCompileError`
`missing-timeout.bloge`	Antipattern reference — nodes without a timeout budget
`over-broad-fallback.bloge`	Antipattern reference — fallback that masks real failure modes

34.4 Why This Matters

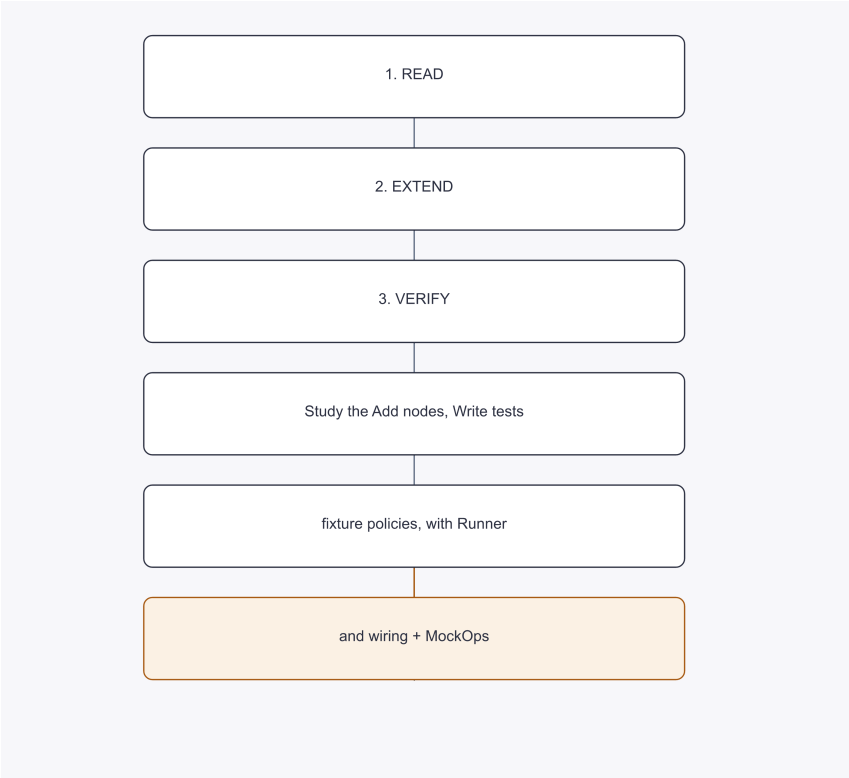
Reading about graphs is not the same as designing them. Every earlier chapter introduced one concept at a time. Real workflows combine all of them at once:

- An order pipeline needs fan-out *and* resilience *and* branching.
- A ticket router needs sentiment analysis *and* a three-way branch *and* a fallback for a flaky ML service.
- A batch processor needs `foreach` *and* downstream aggregation *and* tests that assert on each item's outcome.

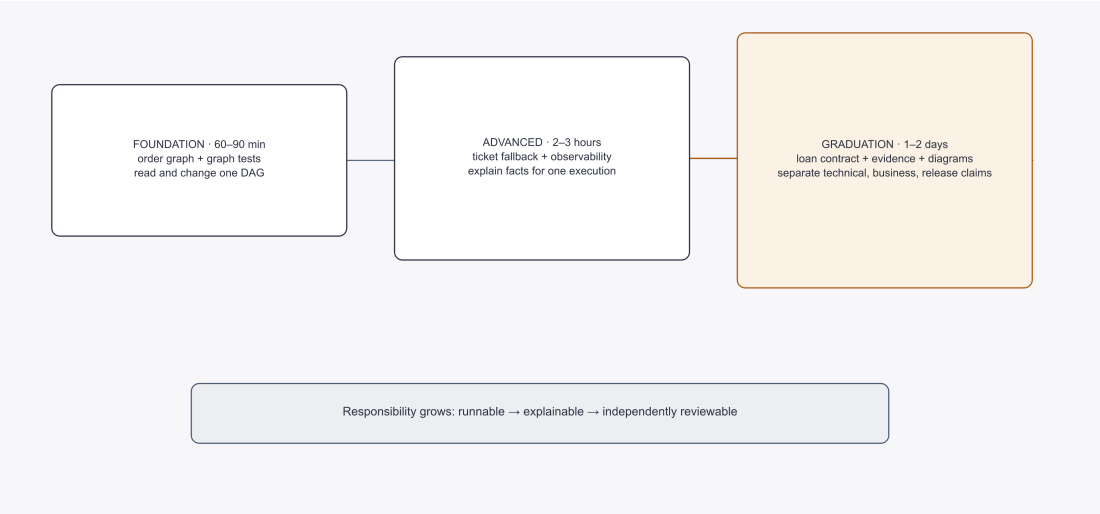
These labs test whether the concepts compose in practice. Each lab starts from a real conformance fixture already in the repository, so you are never inventing syntax. You are reading, extending, and testing.

34.5 Mental Model

Think of the three capstones as progressively wider responsibility:



Read — Open the conformance fixture. Trace every dependency. Identify which nodes run in parallel and which wait.
Extend — Add the features the lab asks for. Wire inputs from upstream outputs. Choose resilience policies deliberately.
Verify — Use `GraphTestRunner` and `MockOperator` to execute the graph, assert on `NodeStatus` values, and check execution order.



Level	Expected effort	Starting file	Work contract	Completion standard
Foundation	60–90 minutes	`order-process.bloge`	One DSL copy + graph tests; explicit steps	Success and rejection paths are reviewable
Advanced	2–3 hours	`ticket-routing.bloge`	DSL, mocks, listener test; boundary hints only	Fallback and observation facts stay distinct
Graduation	1–2 days	Loan starter + verification assets	Graph, tests, Scenario/Policy/Fixture, diagrams, report; acceptance contract only	Runnable, explainable, reviewable, with unproven claims

Copy each read-only starter into your own work directory. Do not modify `submodule/examples` or `submodule/bloge`; they are the remote example baseline and the RC1 source of truth respectively.

34.6 First Working Example

Before you start the labs, make sure you can run the simplest possible test-driven graph cycle. This is your warm-up.

```
// 1. Compile DSL
var helper = new DslTestHelper(registry);
Graph graph = helper.compile("""
    graph warmup {
      node greet : GreetOperator {
        input { name = ctx.name }
      }
    }
    """);

// 2. Wire a mock operator
var runner = new GraphTestRunner(Map.of(
    "GreetOperator", MockOperator.returning(Map.of("message", "Hello"))
));

// 3. Execute
var ctx = new GraphContext(Map.of("name", "Lab Runner"));
GraphResult result = runner.execute(graph, ctx);

// 4. Assert
assert result.isSuccess();
runner.assertNodeExecuted("greet");
```

If that compiles and passes, you are ready.

34.7 Break It Apart

Every lab follows the same anatomy. Here is what each piece teaches:

Phase	Skill exercised	Chapter callback
Read the fixture	Dependency tracing, parallel identification	Ch 3
Add a node	Operator binding, input wiring, `depends_on`	Ch 2, Ch 4
Add resilience	`retry`, `timeout`, `fallback`, `compensate`	Ch 6
Add a branch	`branch on`, `otherwise`, skipped-node reasoning	Ch 5
Add iteration	`foreach`, `loop`, `carry`, `until`	Ch 11
Add suspension	`await`, event correlation, `on_timeout`	Ch 12
Write tests	`GraphTestRunner`, `MockOperator`, `DslTestHelper`	Ch 18

34.8 Common Trap

34.8.1 [X] Adding resilience to every node

It is tempting to slap `retry + timeout + fallback` on every node in the graph. Don't.

Study the antipattern in `over-broad-fallback.bloge`: a blanket fallback on `authorizePayment` hides validation errors behind the same "manual review" as transient timeouts. Downstream nodes cannot distinguish a legitimate degradation from a permanent bug.

And study `missing-timeout.bloge`: a node calling an external service with *no* timeout at all can stall the entire graph indefinitely.

Rule of thumb for the labs:

- External calls → always `timeout`.
- Transiently flaky calls → `retry` with `timeout` per attempt.
- Only add `fallback` when partial data is genuinely acceptable to every downstream consumer.
- Pure local computation (transforms, aggregation) → no resilience needed.

34.9 Guided Rewrite

Before diving into the full labs, practice the extend-and-verify cycle on a small change.

Open `order-process.bloge`. Currently `createOrder` depends only on `calcPrice`:

```
node createOrder : CreateOrderOperator {
    depends_on = [calcPrice]
    input {
      user  = fetchUser.output
      price = calcPrice.output
    }
  }
```

Rewrite task: The business now requires that `createOrder` also logs the credit-check result for audit purposes — but *only* when the order is approved. Add a new input binding:

```
node createOrder : CreateOrderOperator {
    depends_on = [calcPrice, checkCredit]
    input {
      user        = fetchUser.output
      price       = calcPrice.output
      creditCheck = checkCredit.output
    }
  }
```

Now verify the change is safe:

```
var runner = new GraphTestRunner(Map.of(
    "FetchUserOperator",     MockOperator.returning(Map.of("id", "u1")),
    "FetchProductsOperator", MockOperator.returning(Map.of("items", List.of())),
    "CalcPriceOperator",     MockOperator.returning(Map.of("total", 99.0)),
    "CreditCheckOperator",   MockOperator.returning(Map.of("approved", true)),
    "CreateOrderOperator",   MockOperator.recording(),
    "RejectOrderOperator",   MockOperator.recording()
));

var ctx = new GraphContext(Map.of(
    "userId", "u1",
    "productIds", List.of("p1")
));

GraphResult result = runner.execute(graph, ctx);
```

```
assert result.isSuccess();
runner.assertNodeExecuted("createOrder");
runner.assertNodeSkipped("rejectOrder");
```

This pattern — extend the graph, then write a test that proves the new wiring — is the core muscle these labs build.

34.10 Brain Check

Before you start the labs, answer these without looking back:

1. In `ticket-routing.bloge`, which nodes run in parallel? *(Answer: `fetchCustomer` and `fetchTicketHistory` — neither depends on the other.)*
2. In `loan-approval.bloge`, what is the maximum parallelism after `fetchApplication` completes? *(Answer: four — `checkCredit`, `detectFraud`, `verifyIncome`, and `checkBlacklist` all depend only on `fetchApplication`.)*
3. In `batch-order-parallel.bloge`, can `summarize` start before every foreach iteration has finished? *(Answer: no — `summarize` depends on `processOrders`, which completes only when all iterations are done.)*
4. What `MockOperator` factory would you use to simulate a node that takes 500 ms and then succeeds? *(Answer: `MockOperator.delaying(Duration.ofMillis(500), output)`.)*
5. What `GraphTestRunner` method proves a branch target was not chosen? *(Answer: `assertNodeSkipped("nodeId")`.)*

34.11 Lab

34.11.1 Foundation Capstone — Read and Extend an Order Graph

Expected effort: 60–90 minutes **Starting file:** `order-process.bloge`, copied before editing **Allowed scope:** one DSL copy and one `GraphTestRunner` suite; do not edit operator implementations **Hint level:** explicit steps

Requirements:

1. Add a `fetchInventory` node that runs **in parallel** with `fetchUser` and `fetchProducts`. Input: `ctx.productIds`. Timeout: `2s`.
2. Wire `fetchInventory.output` into `calcPrice` as a new `inventory` input. Update `depends_on` accordingly.
3. Add a `compensate` block to `createOrder` that calls `ReleaseInventoryOperator` — modelled after the pattern in `order-saga.bloge`.
4. **Verify:** Write a test that asserts `fetchInventory`, `fetchUser`, and `fetchProducts` all complete; that `createOrder` executes when credit is approved; and that `rejectOrder` is skipped.

Completion standard: the real compiler accepts the DSL; approved and rejected paths separately assert `EXECUTED`/`SKIPPED`; an inventory timeout changes only the expected node; include one sentence saying this lab did not prove the business rule correct.

34.11.2 Advanced Capstone — Make Ticket Degradation Observable

Expected effort: 2–3 hours **Starting file:** `ticket-routing.bloge`, copied before editing **Allowed scope:** DSL, mock operators, and one listener test; do not call a real ticket system **Hint level:** boundaries, not complete DSL

Requirements:

1. `analyzeSentiment` already has a fallback. Add `retry = { attempts: 2, backoff: 300ms, strategy: jitter }` to `fetchTicketHistory` so it can survive transient history-service failures.
2. Add an `escalateToManager` node as a fourth branch case for `classifyPriority.output.priority == "critical"`. Input: `customerId` from `fetchCustomer.output.id` and `sentiment` from `analyzeSentiment.output`.
3. **Verify:** Write one test where `classifyPriority` returns `{ priority: "critical" }` and assert that `escalate`

`ToManager` executes while `assignVipAgent`, `assignNormalAgent`, and `autoResolve` are all skipped.

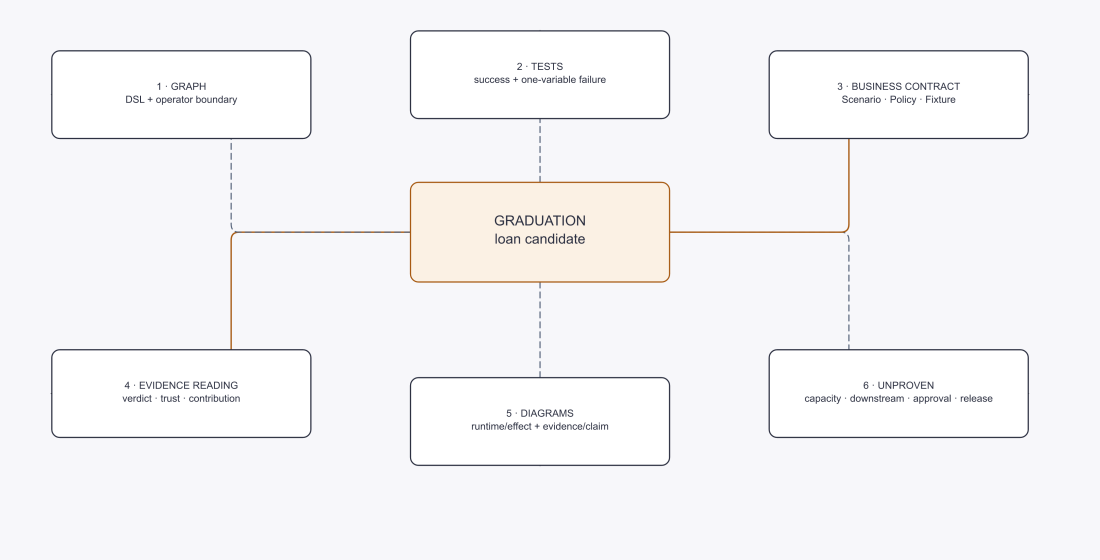
Completion standard: critical, normal, and fallback paths are reviewable; metric, log, and node status are distinct under one `executionId`; fallback success is not described as business success.

34.11.3 Graduation Capstone — Deliver a Reviewable Loan Change

Expected effort: 1–2 days **Starting files:** `loan-approval.bloge` plus the Scenario/Policy/Fixture test assets under that module's `src/test`, copied before editing **Allowed scope:** graph, controlled operator doubles, Scenario, Policy, Fixture, evidence interpretation, and diagrams; do not approve GOLDEN on behalf of a business owner or call a real disbursement effect **Hint level:** acceptance contract only

Requirements:

1. Add a `reserveFunds` node between `aggregateRisk` and `makeDecision`. `reserveFunds` depends on `aggregateRisk` and takes `application = fetchApplication.output` and `risk = aggregateRisk.output` as input.
2. Give `reserveFunds` an `output { reservationId: String }` declaration and a `compensate` block calling `ReleaseFundsOperator` — the same saga pattern used in `order-saga.bloge`.
3. Update `makeDecision` so it also `depends_on reserveFunds`.
4. Add `timeout = 3s` to `reserveFunds` and `fallback = { reservationId: "NONE", status: "degraded" }` so a slow reserve doesn't block the pipeline.
5. **Technical verification:** Write a test with a `MockOperator.throwing(...)` for `ReserveFundsOperator` that proves the fallback fires and the graph still reaches `makeDecision`.
6. **Business verification:** Cover at least `auto-approve`, `manual-review`, and `reject`; forbid real effects in Policy and fix credit, fraud, and income facts in Fixture.
7. **Evidence interpretation:** Write separate conclusions for verdict, evidence trust/source binding, Requirement contribution, and release claim.
8. **Architecture:** Submit one graph/runtime/effect boundary diagram and one evidence-to-claim diagram, naming the business owner and release owner.
9. **Unproven claims:** Include production capacity, real downstream health, GOLDEN approval, and release authorisation.



Completion standard: all six asset classes exist; a one-variable failure is reproducible; every report statement points to a Scenario, source test, or human responsibility; no `PASS` is expanded into release approval.

34.11.4 Optional Extension Bank

The next three exercises do not count toward the three-capstone completion standard. They are workshop extensions. Answers and grading notes live in the teacher guide so readers do not mistake six exercises for six graduation gates.

34.11.4.1 Batch Orders: Add Per-Item Resilience and Summary Branching Fixture: `batch-order-parallel.bloge`

Requirements:

1. Add `retry = { attempts: 1, backoff: 100ms, strategy: exponential }` and `timeout = 3s` to the `deductStock` node inside the `foreach` body.
2. Add a `fallback = { deducted: false, reason: "stock service unavailable" }` to `deductStock` so a single stock failure doesn't break the whole batch.
3. After `summarize`, add a `branch on summarize.output.allSucceeded`:
   - `true` → `notifySuccess` (a new `NotifySuccessOperator` node)
   - `false` → `notifyPartialFailure` (a new `NotifyPartialFailureOperator` node)
4. **Verify:** Wire `deductStock` as `MockOperator.throwing(...)` for one item to confirm the fallback fires, then assert `notifyPartialFailure` executes and `notifySuccess` is skipped.

Concepts exercised: `foreach`, per-item resilience, fallback inside iteration, downstream branching on aggregate output (Ch 5, 6, 10).

Stretch task — global throttling and per-item failure modes. On top of the requirements above, set `batch_size = 10` on the `foreach` so at most 10 items run concurrently, and set `on_item_failure = continue` so the batch never aborts on a single bad item. Use `MockOperator.throwing(...)` to make 3 of 20 items fail, then assert that:

1. `deductStock` is invoked exactly 20 times (each item attempted).
2. `summarize.output.failedCount == 3` and `allSucceeded == false`.
3. `notifyPartialFailure` runs once; `notifySuccess` is skipped.
4. With `batch_size = 10`, the test observes at most 10 concurrent `deductStock` invocations (wrap `MockOperator.of(...)` with an `AtomicInteger active` counter and update a second `peak` counter on entry).

This combination — `batch_size` + `on_item_failure = continue` + downstream branch on the aggregate — is the production pattern for resilient batch processing.

34.11.4.2 Status Polling to Event-Driven: Replace Loop with Await Fixtures: `status-polling.bloge` (start) and `payment-wait.bloge` (reference)

Requirements:

This lab has two parts.

Part A — Extend the polling loop:

1. Add a `notifyReady` node after `fetchResult` that sends a notification when the result is available. Input: `result = fetchResult.output`.
2. Add `timeout = 5s` to `checkStatus` inside the loop body.
3. **Verify:** Assert `notifyReady` executes after `fetchResult`.

Part B — Replace the loop with `await`:

1. Rewrite the graph to use `await` instead of a polling loop. After `submitJob`, declare:

```
await awaitJobReady {
  event "job.ready" where jobId = submitJob.output.jobId
  timeout = 60s
  on_timeout {
    status = "timeout"
    reason = "Job did not become ready within 60 seconds"
```

```
    }
  }
```

2. Add a `branch on awaitJobReady.output.status`:
   - `"ready"` → `fetchResult`
   - `otherwise` → `handleTimeout` (a new node that logs the failure)
3. **Verify:** Write two tests — one that simulates the event arriving (assert `fetchResult` executes, `handleTimeout` skipped) and one that simulates a timeout (assert `handleTimeout` executes, `fetchResult` skipped).

Concepts exercised: `loop`, `await`, event correlation, `on_timeout`, suspension branching, contrasting polling vs. event-driven design (Ch 11, 11, 12).

34.11.4.3 Credit Approval Decision Table

Goal: Refactor a branch-based credit-tier classifier into a `decision_table` node, then explore multi-policy tables and runtime error paths.

Step 1 — Baseline: the branch stack. Open (or create) `ch22/credit-approval-baseline.bloge`. The graph uses a `branch on applicant.output.score` with three explicit cases and an `otherwise`. Confirm it passes all existing tests.

Step 2 — Refactor to `hit=unique` with named output. Replace the branch with:

```
decision_table credit_tier(
  score = applicant.output.score
) hit=unique -> { tier: String, rate: Decimal } {
  rule (score: score >= 750) -> { tier: "platinum", rate: 3.25 }
  rule (score: 680 <= score < 750) -> { tier: "gold", rate: 4.50 }
  otherwise -> { tier: "rejected", rate: 0.0 }
}
```

Update downstream wiring to use `credit_tier.output.tier` and `credit_tier.output.rate`. Run tests — all should still pass.

Step 3 — Add `hit=collect` for discount accumulation. Add a second table that collects all applicable discount labels:

```
decision_table applicable_discounts(
  score = applicant.output.score,
  amount = loan.output.amount
) hit=collect -> String {
  rule (score: score >= 700) -> "loyalty"
  rule (score: score >= 760) -> "premium"
  rule (amount: amount <= 100000) -> "small-loan"
}
```

Wire `applicable_discounts.output.items` to the notification node. Write a test asserting that `score=780, amount=80000` returns three discount labels.

Step 4 — Trigger error paths. Write two negative test cases:

1. Remove `otherwise` from the `hit=unique` table, then pass `score=500`. Assert that the graph throws `DecisionTableViolationException` with code `RUNTIME_DECISION_TABLE_NO_MATCH`.
2. Add an overlapping rule `score >= 700` alongside the existing `score >= 750` rule (both resolve for `score=800`). Assert that the engine throws `RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH` with `hit=unique`. Restore the table after verifying.

Step 5 — BLOGE lint demo. Remove the `otherwise` clause from the `hit=unique` table and run `bloge-lint` (or observe the editor warning). Confirm the `decision-table/missing-otherwise` WARNING appears. Re-add `oth-`

`erwise` to make the lint pass clean.

Exit Criteria:

- `hit=unique` + named output: all tier + rate assertions pass.
- `hit=collect`: discount accumulation test passes (3 labels for `score=780, amount=80000`).
- Both error-path tests assert the correct `code` string.
- `bloge-lint` reports zero WARNINGs after `otherwise` is restored.

Concepts exercised: `decision_table`, hit policies (`unique`, `collect`), named-field output, `RUNTIME_DECISION_TABLE_NO_MATCH`, `RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH`, `decision-table/missing-otherwise` lint rule, `DecisionTableViolationException` assertion (Ch 5, Appendix F).

34.12 Graduation Run and Evidence Check

The graduation capstone must finish with a story defined by a business owner. From the pinned BLOGE `0.9.8-RC1` source tree, run the baseline:

```
mvn -pl bloge-starter-loan-approval -am \
  -Dtest='LoanApprovalApplicationTest,LoanApprovalBusinessScenarioTest,LoanApprovalOperatorFlowTest' \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

The expected scope is three test classes and eleven tests: three decision paths plus funding success, retry, fallback, and compensation behavior. Treat that result as focused starter evidence only. Then run your graduation variant and record the test count, Scenario verdict, evidence trust, and unproven claims. Use Chapter 24 to inspect the business mapping and Chapter 23 to state what the green run still does not prove.

The complete reference decisions, common failures, and 20-point rubric are in `solutions/ch34/teacher-guide.md`. Open it only after the learner submits all six artifact classes; answers are not placed beside the tasks.

34.13 Experiment acceptance card

- **Expected and observed:** The capstone moves from parser to loan cases to evidence governance and six deliverables.
- **Failure and recovery:** Fail a baseline, case, or evidence binding; repair that layer only and do not advance while red.
- **Proof boundary:** Proves a learner variant meets the graduation contract, not release approval.
- **Exercise contract:** First 33 chapters and fixed RC1; add one capability class per level; deliver six assets; stop when 91 baseline tests and the level rule pass.

34.14 Coding practices workshop

Follow one loan application: credit 760, fraud 10, strong income, and a blacklist hit. The starter's existing teaching policy requires `auto_reject`. Move policy from a domain method into a decision table while preserving routing, seven risk-output fields, and the absence of unselected actions.

This chapter owns the complete assets:

- Maven project and bilingual instructions.
- Before graph and after graph.
- Twelve fixed expected examples and executable verification.
- Application typed function and function projection graph.

34.14.1 Act one: predict without changing expected values

Find `blacklist-priority` in the CSV. Rejection and approval predicates can both match. Explain rejection priority and identify the action that must not occur. These fixed answers describe the teaching policy; the refactored implemen-

tation does not generate them. Real policy changes need a new business-approved Oracle.

34.14.2 Act two: choose representation and ownership

Use existing `trim` for name cleanup and application function `loanText.normalizePolicyCode` for locale-independent policy codes. The function does not read policy or approve loans. Keep an independently controlled credit query in an Operator, calculate policy in the decision table, and let the branch select actions.

Complete the Appendix J choice card: expression, execution responsibility, maintenance scope, rejected alternatives, and verification. Repeated code alone does not justify a new built-in or publishing bank policy in core.

34.14.3 Act three: replace one boundary and preserve the contract

From the repository root, using Java 25 and Maven 3.9+:

```
# Install the candidate source dependencies needed by this workshop; upstream tests are skipped.
mvn -q -f submodule/bloge/pom.xml -pl bloge-dsl,bloge-durable,bloge-session-durable -am \
  -DskipTests -Dinvoker.skip=true install

# Compile and execute this chapter's workshop.
mvn -f chapters/34-labs/assets/coding-practices/pom.xml test
```

The BLOGE checkout must match this edition's candidate source. Passing results from older artifacts are not interchangeable. This standalone Maven project needs no real credit provider, database, or funding account.

Twelve parameterized examples execute both graphs and compare all seven risk fields, including decision, priority, reasons, and review form. The refactor does not delete the aggregation responsibility that supplies metadata. Simulated action counts are observed independently of node status: a rejected case requires approve SKIPPED and zero approval actions.

34.14.4 Act four: attack the choice with counterexamples

The execution stage is in FundingBoundaryTest. A visible credit query completes before funding. The provider commits once, loses its first response, and receives a retry with the same business key; the second call returns the original receipt. One configured graph retry produces two total calls in this API.

Another test commits a reservation and withholds the response until timeout. The graph fails while the provider ledger still holds the amount: reconcile by the original key first. A third test rejects reuse of the key with a different amount. These are local provider-contract examples, not real-bank guarantees.

The paired execution refactor uses the CSV's approved applicant. Before refactoring, a credit read is hidden in input evaluation and graph plus SDK each allow three attempts: nine underlying funding calls. Afterward, credit has its own node and graph owns a three-attempt budget: three calls. Both versions deliberately lose all funding responses, fail the funding node, and retain one committed effect under the same business key. This changes the operational budget explicitly; it does not pretend the external action never happened.

Exercise recovery using the fixed source's actual test suites:

```
bash chapters/34-labs/assets/coding-practices/run-runtime-seams.sh
```

The script checks the exact commit and rejects zero tests, skips, and failures in all four selected suites. Lease coordination revokes permission from an old claim, checkpoint writes reject its stale epoch, and dispatch tests retain a permit until actual worker exit. Session tests distinguish restoring old state from new progress. A separate workshop test creates three successive managers over the same stores: recovery counts remain 1, 2, and 3 with the old error retained, then a newly committed round resets the count to zero and clears the error. These fixtures use in-memory stores and controlled workers, not production databases.

Experiment	Required observation	Practices
Move approval ahead of blacklist	The high-score blacklisted applicant is incorrectly approved; a simulated action occurs	BP-06, BP-07, BP-14
Replace the frozen expected answer with the mutant's output	The forged answer accepts the regression; the unchanged CSV answer rejects it	BP-14

Experiment	Required observation	Practices
Replace first directly with unique	Two matching rules conflict; this is not equivalent rejection	BP-07
Add otherwise to collect	Both high-score and default items appear	BP-07
Convert a policy code under Turkish locale	The application function still returns `POLICY-I`	BP-02
Invoke EffectPort twice with one key	The real method count is two; the declaration did not deduplicate	BP-09

The counterexample tests demonstrate controlled mistakes by asserting their divergence from fixed answers. A passing counterexample demonstration does not approve its deliberately incorrect implementation.

34.14.5 Act five: explain the evidence boundary

Expect 23 workshop tests: twelve standard examples, six decision/function/effect-key tests, four credit/funding tests, and one repeated-takeover test, with zero skipped. The separate runtime command reports its actual count by suite. These results cover teaching inputs and controls, not real-provider deduplication, every lending policy, or production capacity.

Transfer the policy to the verification Arc's Scenarios while retaining separate business-answer, path, and effect observations. The runtime command covers declared recovery contracts; in-memory tests do not establish multi-process database or production qualification.

Deliverables and stop rule: Supply the choice card, both graphs, fixed Oracle, full-output comparison, and one detected mutation. Fewer lines or nodes are insufficient. Stop expanding claims when the approved answer is absent or an unknown external result remains unresolved.

34.15 Recap

- The **foundation capstone** proves you can read, extend, and verify an order DAG against a graph contract.
- The **advanced capstone** proves you can keep fallback, branching, and observation facts explainable.
- The **graduation capstone** combines graph, tests, Scenario/Policy/Fixture, evidence interpretation, architecture diagrams, and unproven claims into a reviewable artifact.
- Three optional extensions cover batch, await, and decision tables without changing the graduation standard.
- Technical tests, business verification, and release approval are three ownership layers; passing one cannot substitute for the next.

34.16 Next Step

You have completed Head First BLOGE. The final artifact is not “one more DSL file”; it is a body of work that another person can run, explain, and challenge. From here:

- **Ship a real graph** — pick one workflow in your system, write the DSL, wire the operators, and test it with `Graph TestRunner`.
- **Explore advanced features** — try wiring `DurableSessionManager` and `DurableStateMachineManager` into your test harness to practice crash-safe orchestration patterns. If your use case involves LLM tool-use, explore the `agent` DSL in `bloge-agent-ext` (covered in Chapter 17).
- **Consult the reference layer** when you need exact syntax or edge-case behaviour:
  - DSL Specification
  - Operator Design Specification
  - Core Architecture
  - Getting Started
  - Appendix F — Decision Table
- **Explore advanced examples** in `bloge-examples/` — especially `food-order.bloge`, `claim-processing.bloge`,

and `shipment-planning.bloge` for graphs that combine everything you have learned.
- **Set up tooling** — install the VS Code extension or IntelliJ plugin for real-time diagnostics as you author.

34.17 Reference Links

- Project README — module overview and quick start
- Getting Started — full setup guide
- DSL Specification — formal grammar reference
- Operator Design Specification — operator contract and lifecycle
- Core Architecture — engine internals
- `GraphTestRunner.java` — test harness source
- `MockOperator.java` — test double source
- `DslTestHelper.java` — DSL compile helper source
- `order-process.bloge` — foundation start
- `ticket-routing.bloge` — advanced start
- `loan-approval.bloge` — graduation start
- `batch-order-parallel.bloge` — optional batch exercise
- `status-polling.bloge` — optional loop exercise
- `payment-wait.bloge` — optional await reference
- `order-saga.bloge` — saga compensation reference

CHAPTER

—

Arc 7 Recap — Graduation

Chapters 33–34 end the book with one standard: graduation is not "using every BLOGE feature." It is choosing the smallest model that makes the important behavior visible—and knowing when to stop.

0.1 The final decision loop

Step	Question	Output
STOP	Is this only sequential code, one remote call, or CRUD?	A documented reason not to use BLOGE
PROBE	Is orchestration risk real but still uncertain?	One thin vertical slice and a comparison baseline
ADOPT	Do dependencies, waiting, recovery, or evidence justify a graph?	An ADR with constraints, costs, and exit conditions
GRADUATE	Can another team operate and challenge the result?	Runnable system, tests, diagrams, evidence, and limits

Chapter 33 starts from constraints rather than features. Chapter 34 then offers three capstone levels: foundation, advanced, and graduation. The final level requires six artifact classes—story, executable flow, behavior tests, architecture view, operational evidence, and a limits statement. Optional features earn no credit unless they resolve a stated constraint.

0.2 Try It: Take One Feature Away

Choose your capstone design and remove its most impressive BLOGE feature. If the business story, recovery boundary, and evidence remain clear, leave the feature out. If clarity collapses, write the exact constraint that justifies putting it back. That sentence is the heart of your final ADR.

0.3 The book's last claim

You now have a method, not a catalog: model dependencies, name lifecycle owners, control time and inputs, bind claims to evidence, and stop at the strongest conclusion the evidence supports.

CHAPTER

—

Appendix A — Built-in Functions

Every guard, constraint, and expression in bloge DSL has the same toolbox of built-in functions available. This appendix is the single reference for that toolbox: every name, every signature, every gotcha. Bookmark this page once you start writing real DSL.

0.1 When You Need This Reference

Question	Section
What does `coalesce` / `ifNull` / `isNull` do?	Null & Default Handling
Can I do regex inside a `when:` guard?	String & Regex
How do I sum or average a list field?	Math & Aggregation
How do I read a value out of a nested map?	Map & Object
How do I render a timestamp or compute a duration?	Date & Time
How do I sign or hash a payload?	Optional Crypto Modules
How do I check whether a function call is allowed at compile time?	Purity & Compile-Time Evaluation

0.1.1 Find a Function by Task

I need to…	Start with
clean, split, match, or format text	String functions
total, compare, round, or aggregate numbers	Math and aggregation
filter, map, flatten, or inspect a list	Collection functions
read nested object or map fields safely	Map/object plus null handling
convert a value before schema validation	Type conversion
calculate a deadline or duration	Date/time functions
create a digest or signature	Optional crypto module

Chapter 5 (Branches that Decide) introduces these functions inside `when:` guards. Chapter 7 (Designing Good Operators) uses them inside operator `constraints` and `usageExample`. Chapter 15 (State Machines) uses them inside transition guards. Wherever you see an expression in bloge DSL, this list is what you can call.

0.2 Function Call Syntax

All functions use **standard call syntax**:

```
when: concat("Hello, ", ctx.name) == "Hello, Alice"
when: contains(split(ctx.tags, ","), "premium")
when: padLeft(toString(ctx.id), 10, "0")
```

bloge does **not** use a Unix-style pipe operator (`|`) for function composition. The `|` token is reserved by the lexer for the logical OR operator (`||`). Compose functions by nesting calls.

0.2.1 Null Handling Convention

Core built-ins are **null-safe by default**. Passing `null` either:

- returns `null` (safe-nav semantics), or
- returns a sensible default — `size(null) → 0`, `isEmpty(null) → true`, `concat(null, "x") → "x"`, etc.

You can rely on this convention to write fewer defensive guards. The optional crypto module is the one exception — see its section below.

0.2.2 Purity & Compile-Time Evaluation

Most functions are **pure** — given the same input they always produce the same output and have no side effects. The DSL compiler is allowed to fold pure-function calls at compile time when arguments are constant.

A small number of functions are **impure** and may not be folded:

- `now()` — current timestamp from the engine's time source
- `today()` — current date from the engine's time source
- `uuid()` — random UUID
- `secret(name)` — secret lookup (crypto module)

In strict compile-time-only contexts (constants, schema constraints) the compiler will reject impure calls. Use them inside runtime guards instead.

0.3 1. String Functions (17)

Function	Signature	Returns	Description
`concat`	`concat(arg1, arg2, …)`	`String`	Concatenates all arguments as strings. `null` arguments are treated as empty.
`substring`	`substring(str, start [, end])`	`String`	Substring from `start` (inclusive) to `end` (exclusive); clamps to bounds.
`uppercase` / `upper`	`uppercase(str)`	`String`	Upper-case the string.
`lowercase` / `lower`	`lowercase(str)`	`String`	Lower-case the string.
`trim`	`trim(str)`	`String`	Strip leading and trailing whitespace.
`replace`	`replace(str, oldStr, newStr)`	`String`	Replace every literal occurrence of `oldStr`.
`startsWith`	`startsWith(str, prefix)`	`Boolean`	Prefix test; `false` on null.
`endsWith`	`endsWith(str, suffix)`	`Boolean`	Suffix test; `false` on null.
`indexOf`	`indexOf(str, target)`	`Number`	Index of first occurrence, or `-1`.
`length` / `len`	`length(str)`	`Number`	String length. Also works on lists and maps. `0` on null.
`matches`	`matches(str, regex)`	`Boolean`	Regex match. Returns `false` on null or invalid pattern (no exception).
`replaceAll`	`replaceAll(str, regex, replacement)`	`String`	Regex replace-all. Returns original on null or bad pattern.
`split`	`split(str, delimiter)`	`List`	Split by literal delimiter (not regex). Empty list on null.

Function	Signature	Returns	Description
join	join(list, delimiter)	String	Join list elements with delimiter. Null elements become empty strings.
padLeft	padLeft(str, length, padChar)	String	Left-pad to length using a single pad character.
padRight	padRight(str, length, padChar)	String	Right-pad to length using a single pad character.

```
when: startsWith(ctx.email, "admin@") and contains(ctx.email, "@")
when: matches(ctx.phone, "^\\+?[0-9 ]{6,}$")
when: padLeft(toString(ctx.orderId), 8, "0") == "00012345"
```

0.4 2. Math & Aggregation (10)

Function	Signature	Returns	Description
abs	abs(n)	Number	Absolute value.
min	min(a, b)	Number	Minimum of two numbers.
max	max(a, b)	Number	Maximum of two numbers.
round	round(n)	Number	Round to nearest integer (Math.round).
ceil	ceil(n)	Number	Round up to nearest integer.
floor	floor(n)	Number	Round down to nearest integer.
clamp	clamp(n, min, max)	Number	Constrain n to [min, max].
pow	pow(base, exponent)	Number	base ^ exponent.
sum	sum(list)	Number	Sum of numeric elements. Null elements skipped. Empty list → 0.
avg	avg(list)	Number	Average of numeric elements. Empty/null → null.

```
when: sum(ctx.items.*.price) > 1000
when: clamp(ctx.retries, 0, 5) == ctx.retries
```

0.5 3. Collection Functions (13)

Function	Signature	Returns	Description
size	size(collection)	Number	Size of list, map, or string. 0 on null.
contains	contains(collection, item)	Boolean	Membership. Lists → exact match; strings → substring; maps → key lookup.
first	first(list)	any	First element, or null.
last	last(list)	any	Last element, or null.
isEmpty	isEmpty(collection)	Boolean	True if empty or null.
distinct	distinct(list)	List	Remove duplicates, preserve order.
flatten	flatten(listOfLists)	List	Flatten one level. Non-list elements pass through.

Function	Signature	Returns	Description
sort	sort(list)	List	Sort using natural order (falls back to toString); nulls first.
take	take(list, n)	List	First n elements.
drop	drop(list, n)	List	Skip first n elements.
reverse	reverse(list)	List	Reverse list order.
any	any(list, value)	Boolean	True if any element equals value.
all	all(list, value)	Boolean	True if every element equals value. Empty list → true.

```
when: contains(ctx.tags, "premium") and size(ctx.items) > 0
when: any(ctx.errors, "TIMEOUT")
when: all(distinct(ctx.statuses), "OK")
```

0.6 4. Map & Object (5)

Function	Signature	Returns	Description
keys	keys(map)	List	All keys as a list.
values	values(map)	List	All values as a list.
entries	entries(map)	List	List of {key, value} maps.
merge	merge(map1, map2)	Map	Shallow merge; map2 overwrites map1.
has	has(map, key)	Boolean	Whether the map contains the key.

```
when: has(ctx.headers, "X-Trace-Id")
when: size(keys(ctx.attributes)) > 3
```

0.7 5. Null & Default Handling (3)

Function	Signature	Returns	Description
coalesce	coalesce(v1, v2, …)	any	First non-null argument, or null if all are null.
isNull	isNull(value)	Boolean	Value is null.
isNotNull	isNotNull(value)	Boolean	Value is not null.

coalesce is the workhorse for default values. ifNull (see Advanced) is a two-argument variant that reads more naturally for a single fallback.

```
when: isNotNull(ctx.userId)
let region = coalesce(ctx.region, ctx.country, "GLOBAL")
```

0.8 6. Type Conversion (5)

Function	Signature	Returns	Description
`toString`	`toString(value)`	`String`	Convert any value to its string form.
`toNumber`	`toNumber(value)`	`Number`	Parse number from string, or return the numeric value. Invalid strings coerce to `0`.
`toBoolean`	`toBoolean(value)`	`Boolean`	Parse `"true"` / `"false"` or pass through boolean.
`toInt`	`toInt(value)`	`Number`	Convert to integer; truncates decimals.
`typeOf`	`typeOf(value)`	`String`	`"String"`, `"Number"`, `"Boolean"`, `"List"`, `"Map"`, `"Object"`, or `"Null"`.

```
when: toNumber(ctx.amountStr) > 100
when: typeOf(ctx.payload) == "Map"
```

0.9 7. Date/Time (6)

All dates flow as **ISO-8601 strings** (`yyyy-MM-ddTHH:mm:ssZ`). Use these helpers to format, parse, add, and diff them.

Function	Signature	Returns	Pure?	Description
`now`	`now()`	`String`	[X]	Current timestamp from `GraphEngine.currentTimeSource()`.
`today`	`today()`	`String`	[X]	Current date `yyyy-MM-dd` (UTC).
`formatDate`	`formatDate(isoStr, pattern)`	`String`	[OK]	Format ISO instant with a `DateTimeFormatter` pattern (e.g. `"yyyy-MM-dd HH:mm"`).
`parseDate`	`parseDate(str, pattern)`	`String`	[OK]	Parse with a pattern; returns an ISO-8601 instant.
`addDuration`	`addDuration(isoStr, durationStr)`	`String`	[OK]	Add a duration. Accepts `"2h"`, `"30m"`, `"1d"`, `"500ms"`, or ISO-8601 form `"PT2H"`.
`diffDuration`	`diffDuration(iso1, iso2, unit)`	`Number`	[OK]	Distance between two instants. `unit`: `"s"`/`"seconds"`, `"m"`/`"minutes"`, `"h"`/`"hours"`, `"d"`/`"days"`.

```
when: diffDuration(ctx.createdAt, now(), "h") > 24
let expiresAt = addDuration(now(), "30m")
```

> ***Determinism tip.*** *Inside replay-sensitive contexts (durable sessions, deterministic tests) prefer `parseDate`/`addDuration` driven by stored timestamps over `now()` / `today()`.*

0.10 8. Utility / ID (2)

Function	Signature	Returns	Pure?	Description
`uuid`	`uuid()`	`String`	[X]	Random UUID v4.
`format`	`format(template, arg1, arg2, …)`	`String`	[OK]	Java `String.format` style placeholders (`%s`, `%d`, …). Returns template on format error.

```
let traceId = uuid()
let message = format("Order %s failed after %d retries", ctx.orderId, ctx.retries)
```

0.11 9. Advanced Utility (8)

Function	Signature	Returns	Description
`regexExtract`	`regexExtract(str, pattern [, group])`	`String`	First match for a regex group (default 1); `null` if no match.
`regexExtractAll`	`regexExtractAll(str, pattern)`	`List`	All non-overlapping matches.
`getField`	`getField(obj, fieldName)`	`any`	Read a field from a map or bean (reflection); null-safe.
`setField`	`setField(obj, fieldName, value)`	`Map`	Return a **shallow copy** with the field updated (immutable semantics).
`template`	`template(str, bindings)`	`String`	Replace `${key}` placeholders using a map of bindings.
`range`	`range(start, end)`	`List`	Integer sequence `[start, end)`.
`mapOf`	`mapOf(k1, v1, k2, v2, …)`	`Map`	Build a map from alternating key/value arguments.
`listOf`	`listOf(e1, e2, …)`	`List`	Build a list from arguments.
`ifNull`	`ifNull(value, default)`	`any`	`value` if non-null, otherwise `default`.

```
let host = regexExtract(ctx.url, "https?://([^/]+)/")
let updated = setField(ctx.user, "lastSeenAt", now())
let greeting = template("Hello, ${name}!", mapOf("name", ctx.name))
```

0.12 10. JSON Functions (2)

JSON helpers are registered when a `JsonCodec` is provided to the DSL compiler. They are part of `BuiltInFunctions.registerAll(registry, jsonCodec)`.

Function	Signature	Returns	Description
`toJson`	`toJson(obj)`	`String`	Serialize to JSON via the configured codec.
`fromJson`	`fromJson(str)`	`any`	Deserialize JSON to `Map` / `List` / primitive.

```
let payload = toJson(mapOf("orderId", ctx.orderId, "total", ctx.total))
let parsed = fromJson(ctx.responseBody)
```

If no `JsonCodec` is configured, calls to these names will fail at DSL compilation with an unknown-function error.

0.13 Optional Modules (`bloge-functions-crypto`)

The `bloge-functions-crypto` module is loaded automatically via Java `ServiceLoader` when its JAR is on the class-path. It exposes three families of functions. Crypto functions that resolve secrets require a `SecretProvider` to be

configured on the engine.

0.13.1 Crypto (5)

Function	Signature	Returns	Pure?	Description
`hmacSha256`	`hmacSha256(data, keyRef)`	`String`	[OK]	HMAC-SHA256 hex digest. `keyRef` is resolved via `SecretProvider`.
`hmacSha512`	`hmacSha512(data, keyRef)`	`String`	[OK]	HMAC-SHA512 hex digest.
`aesEncrypt`	`aesEncrypt(plaintext, keyRef [, ivRef])`	`String`	[OK]	AES-256-CBC; base64 output. Random IV when not supplied.
`aesDecrypt`	`aesDecrypt(ciphertext, keyRef [, ivRef])`	`String`	[OK]	AES-256-CBC; expects base64 input.
`secret`	`secret(name)`	`String`	[X]	Resolve a secret by name from the `SecretProvider`.

0.13.2 Hash (4)

Function	Signature	Returns	Description
`md5`	`md5(str)`	`String`	MD5 hex digest.
`sha1`	`sha1(str)`	`String`	SHA-1 hex digest.
`sha256`	`sha256(str)`	`String`	SHA-256 hex digest.
`sha512`	`sha512(str)`	`String`	SHA-512 hex digest.

0.13.3 Encoding (6)

Function	Signature	Returns	Description
`base64Encode`	`base64Encode(str)`	`String`	Base64 encode UTF-8 string.
`base64Decode`	`base64Decode(str)`	`String`	Base64 decode; `null` on invalid input.
`hexEncode`	`hexEncode(str)`	`String`	Hex encode UTF-8 string.
`hexDecode`	`hexDecode(str)`	`String`	Hex decode; `null` on invalid input.
`urlEncode`	`urlEncode(str)`	`String`	URL encode (UTF-8).
`urlDecode`	`urlDecode(str)`	`String`	URL decode (UTF-8); `null` on invalid input.

```
let signature = hmacSha256(ctx.requestBody, "webhook-secret")
let fingerprint = sha256(ctx.userId)
let safeQuery = urlEncode(ctx.search)
```

> ***Crypto module exception.*** *Unlike core functions, crypto/hash functions may throw on invalid input (bad keys, malformed ciphertext). Validate inputs at the boundary or wrap calls inside operators that handle the failure mode you want.*

0.14 Common Mistakes

- **Pipe operator.** `value | uppercase` is **not** valid bloge DSL. Write `uppercase(value)`. The lexer reads `|` as the start of `||`.

- **Regex split.** `split(s, ",")` splits on the literal string `","`, not a regex. Use `replaceAll` first if you need pattern-based splitting.
- **Date math without ISO strings.** `addDuration("2024-01-01", "1d")` will fail — the first argument must be an ISO-8601 instant. Use `addDuration(parseDate("2024-01-01", "yyyy-MM-dd"), "1d")` if you have a bare date.
- **Mutating maps.** `setField` does not mutate in place; it returns a copy. Bind the result: `let updated = setField(ctx.user, "name", "Alice")`.
- **Impure calls in constants.** `now()` and `uuid()` are rejected in compile-time-only positions (constant folding, schema constants). Move them into runtime guards or operator outputs.
- **Crypto without `SecretProvider`.** `hmacSha256` / `aesEncrypt` will fail if `SecretProvider` is not configured on the engine. Wire it in your Spring or manual bootstrap (see Chapter 20).
- **`toBoolean` on numbers.** `toBoolean(1)` returns `null`, not `true`. Only `"true"` / `"false"` strings and actual booleans are accepted.

0.15 Relationship to Main Chapters

- **Chapter 5 — Branches that Decide.** Where `when:` guards are introduced — the primary place you compose these functions.
- **Chapter 7 — Designing Good Operators.** Uses `constraints` and `usageExample` expressions; the same function set applies.
- **Chapter 14 — Multi-Turn Sessions** and **Chapter 15 — State Machines.** Use guard expressions for session phase transitions and state-machine transitions; both compile through the same `ExpressionEvaluator`.
- **Chapter 9 — Tooling Workflow.** The `operator-metadata.json` exporter surfaces these function names to IDE tooling so completion and validation match what the compiler will accept.
- **Chapter 20 — Spring and Production Wiring.** Where `JsonCodec` and `SecretProvider` are configured, enabling the optional JSON and crypto helpers above.

Verification. The 72-function core surface matches `BuiltInFunctionsTest.registerAll_populatesExpectedNumberOfEntries()` in `bloge-core`. Add the two JSON helpers and the eleven crypto/hash/encoding helpers to reach the **85** functions documented here.

CHAPTER

—

Appendix B — Session Cheat Sheet

Read Chapter 14 first. Return here when authoring or diagnosing a multi-turn session.

0.1 Choose the Boundary

Need	Use	Do not use
One external pause and resume	Graph `await`	A session with one artificial round
Repeated outside turns with similar processing	`session`	A hand-built loop around HTTP requests
Named business states and backward transitions	`state_machine`	Phase names pretending to be states

0.2 Phase Card

Item	Question
`ONCE` or `ROUND`	Does this phase run once, or once per incoming signal?
`max_rounds`	What finite budget stops a runaway conversation?
`until`	What observable result ends the phase?
signal schema	Which payload is legal at this boundary?
timeout / idle timeout	How does waiting end without a signal?
owner	Which outer runtime owns identity, recovery, and termination?

0.3 Runtime Moves

Move	Meaning	Evidence to retain
start	Create one stable session identity	Definition version and initial context
signal	Deliver one validated round payload	Signal type, correlation identity, accepted/rejected outcome
restore	Rehydrate from the last durable snapshot	Snapshot version and original deadlines
terminate	End by policy rather than normal `until`	Stable reason and final status

0.4 Failure Scan

- Reject an invalid signal before it mutates round state.

- Keep phase and session round limits finite.
- Test late, duplicate, and concurrent signals.
- For crash recovery, use `bloge-session-durable`; an in-memory session is not a durability claim.
- In nested orchestration, send all external signals through the declared outer owner.

Mainline: Chapter 14, Chapter 16. Exact contracts: `docs/session-phase-round-specification.md`, `bloge-session-ext`, and `bloge-session-durable` in the BLOGE source tree.

CHAPTER

—

Appendix C — Streaming

A graph usually materialises all node outputs before downstream nodes run. Streaming changes that contract: data flows item-by-item through a channel while the upstream node is still producing.

0.1 When You Need Streaming

Most BLOGE graphs are ”batch-complete”: every node finishes, its output is stored, and only then do downstream nodes see that output. This works well when datasets are bounded and fit comfortably in memory.

Streaming is for the cases where that model breaks:

- **The upstream produces data incrementally** — audio chunks, database cursor pages, live sensor readings — and you want to process items as they arrive.
- **The dataset is large enough that materialising it all at once is wasteful** — you would rather pipeline through a bounded buffer.
- **Latency matters more than throughput** — showing partial results to a user while the rest are still being computed.

If none of these apply, regular nodes and `foreach` are simpler and easier to test. If the hard part of your workflow is not data volume but lifecycle — multiple external turns, named states, or signal ownership — continue with Chapter 14 through Chapter 16 instead of reaching for streaming primitives.

0.2 The Three Streaming Primitives

BLOGE offers three constructs that produce streaming output. Each serves a different pattern.

0.2.1 1. `stream node` — a streaming source or transform

A `stream node` declares an operator whose output is emitted **item by item** through an internal `NodeChannel` rather than returned as a single value.

```
stream node audioCapture : AudioCapture {
  buffer = 64
}
```

The operator pushes items into the channel at its own pace. Downstream nodes can consume the channel live or wait for it to close and receive the materialised list.

Use when: a single operator produces an unbounded or long-running sequence of items (audio capture, event stream, paginated API).

0.2.2 2. `stream foreach` — iterate a collection with streaming output

`stream foreach` is the streaming variant of `foreach`. It iterates over a collection and **emits each item's result downstream as soon as that item completes**, rather than waiting for every item to finish.

```
stream foreach processOrders : item in loadOrders.output.orders {
  buffer = 16
  node processItem : OrderProcessor {
    input {
      order = item
    }
  }
}
```

Use when: you have a bounded collection but want downstream nodes to start consuming results before the last item is processed.

0.2.3 3. `stream loop` — poll with streaming output

`stream loop` is the streaming variant of `loop`. Each iteration's output is emitted downstream immediately, and the loop continues until `until` is satisfied or `max_iterations` is reached.

```
stream loop checkStatus {
  max_iterations = 100
  delay = 2s
  depends_on = [initMonitor]
  buffer = 8
  node pollStatus : StatusPoller {
    input {
      serviceId = initMonitor.output.serviceId
      iteration = loopIteration
    }
  }
  until pollStatus.output.status == "ready"
}
```

Use when: you are polling or producing items over time and want to forward each result immediately rather than collecting all iterations first.

0.3 `.stream` vs `.output` — the Two Edge Types

When a downstream node references a streaming node's result, the **edge type** determines when and how data arrives.

Syntax	Edge Type	Behavior
`upstream.stream`	**StreamEdge**	Downstream receives items **live**, one by one, as the channel produces them. The downstream operator must be able to process incremental input.
`upstream.output`	**DirectEdge**	Downstream **waits** until the stream closes, then receives the complete materialised `List<T>`. Use this when the downstream operator needs the whole dataset.

0.3.1 Choosing the right edge

Ask: **does the downstream operator need all the data before it can start?**

- **Yes** → use `.output`. Example: a report generator that needs every processed order to compute totals.

- **No** → use `.stream`. Example: a speech-to-text node that transcribes audio chunks as they arrive.

The voice pipeline example shows both edges in a single graph:

```
/// Receives audio chunks live via StreamEdge
stream node speechToText : SpeechToText {
  input {
    audio = audioCapture.stream       // StreamEdge — live forwarding
  }
  buffer = 16
}

/// Waits for the full transcript via DirectEdge
node textAnalysis : TextAnalyzer {
  depends_on = [speechToText]
  input {
    transcript = speechToText.output   // DirectEdge — materialised List<T>
  }
}
```

0.4 Buffer Sizing

Every streaming construct has a `buffer` parameter that sets the capacity of the internal `NodeChannel` ring buffer.

```
stream node audioCapture : AudioCapture {
  buffer = 64    // ring buffer holds up to 64 items
}
```

Default: 16 items.

0.4.1 How to think about buffer size

The buffer decouples the producer's rate from the consumer's rate. When the buffer is full, the producer blocks (back-pressure) until the consumer catches up.

Scenario	Guidance
Producer is much faster than consumer	Increase the buffer to absorb bursts; back-pressure will still kick in when the gap is too large.
Producer and consumer run at similar speeds	Default of 16 is usually fine.
Memory is constrained or items are large	Reduce the buffer to limit memory usage; accept more frequent back-pressure pauses.
Latency-sensitive pipeline	Keep the buffer small so items are not queued longer than necessary.

There is no formula — start with the default, measure, and adjust. The engine logs back-pressure events when observability listeners are active, which makes tuning straightforward.

0.5 Back-pressure, Cancellation, and Durability Boundary

Event	Contract	What to test
Buffer becomes full	Producer waits; items must not be silently dropped	Slow the consumer and observe bounded memory plus eventual progress
Downstream cancels	Cancellation must propagate upstream and close the channel	Assert producer termination and no post-cancel emissions
Producer fails	Channel closes with failure; materialized `.output` must not look complete	Assert downstream failure and partial-output policy
Process crashes	In-memory channel contents are not a durable checkpoint	Restart and prove what is replayed from source or store

Streaming and durable execution solve different problems. A bounded channel controls live flow inside a running process; it does not make in-flight items survive a crash. If replay is required, name a durable source offset or checkpoint and define duplicate handling. Do not claim exactly-once delivery from buffer size alone.

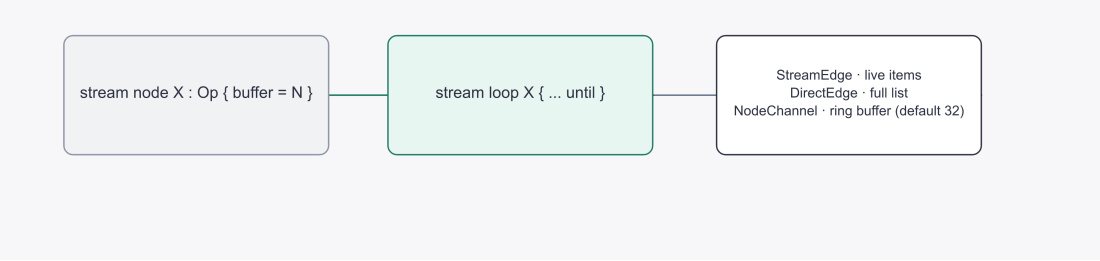
0.6 Putting It Together — Real Examples

The following files in the repository demonstrate the three streaming primitives in realistic scenarios.

File	Primitive	What it shows
`streaming-batch.bloge`	`stream foreach`	Processes customer orders with streaming output; downstream report uses `.output` (DirectEdge) to materialise all results.
`streaming-status-monitor.bloge`	`stream loop`	Polls a service at 2-second intervals, streaming each status update downstream; loop exits early when the service reports ”ready”.
`voice-pipeline.bloge`	`stream node`	Audio capture → speech-to-text → text analysis. Shows both `.stream` (live) and `.output` (materialised) edges in the same graph.

Open these files, read the doc comments, and trace how data flows from the streaming source through the buffer to downstream consumers. Modify `buffer` values and observe how back-pressure behavior changes.

0.7 Quick Reference



0.8 Common Mistakes

Mistake	Why It Happens	Fix
Using `.stream` when the operator needs all data	Habit of choosing ”the fast path”	If the operator cannot produce output until it has seen every item, use `.output`.
Setting buffer too large ”just in case”	Trying to avoid back-pressure entirely	A huge buffer delays back-pressure signals and wastes memory. Start with the default.
Forgetting that `stream foreach` still collects for `.output`	Assuming streaming means nothing is materialised	Items are streamed *and* collected; `.output` gives you the full list after the stream closes.
Not testing the back-pressure path	Only testing with small datasets where the buffer never fills	Use `MockOperator.delaying(...)` in tests to simulate a slow consumer and verify the graph behaves correctly under back-pressure.

0.9 Relationship to the Main Chapters

Streaming is not covered in a single chapter because it builds on several concepts:

- **Nodes and dependencies** — Chapter 2 and Chapter 3
- **Batch and iteration** — Chapter 11 introduces `foreach` and `loop`; streaming variants add the channel.
- **Testing** — Chapter 18 shows how to assert on streaming graph results.
- **Observability** — Chapter 19 covers listeners that log back-pressure events.

This appendix collects the streaming mental model in one place so you can reference it whenever a graph needs incremental data flow.

CHAPTER

—

Appendix D — State Machines

*Chapter 15 teaches state machines as a narrative. Chapter 16 shows how they compose with sessions. This appendix is the **lookup reference** — the operator table, configuration knobs, listener callbacks, and checkpoint shape you keep coming back to when authoring a non-trivial machine.*

0.1 When You Need This Appendix

Question	Go to
”How do I model a lifecycle with named states?”	Chapter 15
”How do state machines and sessions nest?”	Chapter 16
”What operators are legal in a `when` guard?”	This appendix — Guards
”What does the engine persist when a checkpoint fires?”	This appendix — Checkpoint shape
”Which listener callback gives me state-enter vs state-exit timing?”	This appendix — Listener SPI
”Should this workflow be an SM, a session, or a plain graph?”	This appendix — Decision table

If you have not yet read Chapter 15, start there. This appendix assumes you already know what `[initial]`, `on EVENT -> target`, and `[terminal]` mean.

0.2 DSL Keyword Reference

Every keyword Chapter 15 introduces, in one table. Each appears inside a `state machine` block. Names are case-sensitive.

Keyword	Where it goes	What it does
`[initial]`	State header marker	Exactly one state per machine carries this marker; execution begins there.
`[terminal]`	State header marker	Any number of states may be terminal; reaching one completes the machine with status `COMPLETED`.
`on EVENT -> target`	Inside a state	Wait for a signal whose `event` name matches; transition to `target` on receipt.
`on EVENT when GUARD -> target`	Inside a state	Same, but only fire when the guard expression evaluates truthy. Multiple `when` clauses are tried in declaration order.

Keyword	Where it goes	What it does
`on *`	Inside a state	Automatic transition — fires the moment the state is entered, before the executor parks. Use for fan-out into a sub-graph that always runs.
`timeout = <duration>`	Inside a state	State-local timeout. Combined with `on_timeout` to declare where to go when it fires.
`on_timeout -> target`	Inside a state	Target for the state-local `timeout`.
`global_timeout = <duration>`	Top of machine	Whole-machine deadline. When it fires without a matching `global_transition`, status becomes `FAILED`.
`global_transitions { … }`	Top of machine	Transitions that apply to every non-terminal state. Useful for `CANCEL` / `ERROR` events that can arrive in any state.
`max_transitions = N`	Top of machine	Hard cap on transition count. Default `100`. Exceeding it sets status `FAILED`.
`max_state_visits = N`	Top of machine	Hard cap on how many times any single state may be entered. Default `10`.

Duration syntax *matches the rest of BLOGE:* `30s`*,* `5m`*,* `2h`*,* `24h`*,* `PT15M` *(ISO-8601). Use the form your DSL lint allows.*

0.3 Guard Expression Catalog

Guard expressions are parsed by the same evaluator the DSL uses for `when` clauses and conditional edges. The grammar is verified against `ExpressionEvaluator` in `bloge-dsl`.

0.3.1 Operators

Category	Operator	Notes
Arithmetic	`+ - * / %`	Numeric; `+` does **not** concatenate strings. Use string interpolation instead.
Comparison	`== != < <= > >=`	`==` / `!=` work on any comparable types; numeric promotion follows Java widening rules.
Logical	`&& || !`	Short-circuit. `!` is the only logical unary.
Unary	`-` (negate)	Numeric only.
Null-coalesce	`??`	`a ?? b` returns `a` unless `a` is `null`, in which case `b`.
Ternary	`cond ? a : b`	Right-associative.
Path lookup	`.` and `[…]`	Dotted paths resolve against the context map; index access works for lists.

0.3.2 Worked examples

```
# Numeric guard against a node output
on submit when ctx.scoreNode.output.score >= 80 -> approved

# String comparison
on classify when ctx.input.tier == "platinum" -> fastTrack

# Null-coalesce + ternary
on review when (ctx.audit.output.flags ?? []) == [] ? true : false -> noManualReview
```

```
# Composite logical
on dispatch when ctx.geo.region == "EU" && ctx.user.consent.marketing -> sendEmail
```

0.3.3 Resolution rules

- **Missing path** — a path that does not resolve evaluates to `null`. A guard whose top-level expression is `null` is treated as **falsy**, so the transition is skipped.
- **Type mismatch** — comparing a number with a string is `false`; it does **not** raise. The lint catches obvious mismatches at compile time.
- **Boolean coercion** — `null`, `false`, `0`, `""`, and empty collections are falsy. Everything else is truthy.
- **Evaluation order** — when multiple `on EVENT when …` clauses exist for the same event, they are tried top-to-bottom; the first guard that evaluates truthy wins. Put the most specific guard first.

0.4 Configuration Knobs

Knob	Default	Raise when	Lower when
`max_transitions`	`100`	The machine has many automatic transitions or a long-running retry loop with bounded back-off.	The machine should be short; a runaway loop indicates a missing terminal state.
`max_state_visits`	`10`	A state legitimately re-enters (e.g. retry budget) and you have an external counter for the wider attempt.	A state is supposed to be visited at most once; lower it to fail loud.
`global_timeout`	unset	The machine has no natural deadline and you want crash-safe expiry.	Each state already has its own `timeout` and a global cap would only mask state-level bugs.
State `timeout`	unset	The state can wait indefinitely for an external event and you need a fallback.	The state should complete promptly from `on *` — a state timeout adds noise.

> ***What happens when a cap is hit:*** *the executor transitions to status `FAILED`, fires `onStateMachineComplete` with `FAILED`, and writes a final checkpoint. Downstream signals are rejected.*

0.5 Failure and Rejection Outcomes

The state-machine API exposes typed exceptions rather than one universal error-code enum. Preserve the type at service boundaries; do not parse messages.

Outcome	Typed signal	Meaning
Event has no legal transition	`UnhandledEventException`	Current state cannot consume that event
Signal arrives after completion or at an illegal point	`StateMachineSignalRejectedException`	Instance state rejects the signal
State graph fails	`StateGraphExecutionException`	Work inside the selected state failed
Transition or visit budget is exhausted	`StateMachineTransitionLimitExceededException` / `StateVisitLimitExceededException`	Safety cap stopped a runaway machine
Deadline expires	`StateMachineTimeoutExceededException`	State or global timeout ended execution

Outcome	Typed signal	Meaning
Durable definition changed	`StateMachineVersionMismatchException`	Checkpoint hash does not match current definition
Migration is invalid	`StateMachineMigrationException`	Declared state mapping cannot be applied

If the state machine is backed by a generic durable store, storage failures additionally carry `DurableErrorCode`; that enum describes store semantics, not state-machine business transitions.

0.6 Listener SPI Cheat Sheet

`StateMachineListener` is the single hook for audit, metrics, and debugging. All methods are `default`-no-op; override only what you need.

Callback	Fires when	Typical use
`onStateMachineStart(StateMachineStartEvent)`	`execute(def, ctx)` is invoked, before the initial state is entered.	Audit log row, span open.
`onStateEnter(StateEnterEvent)`	After a transition lands, before the state's graph runs.	State-duration timer start, MDC enrichment.
`onStateExit(StateExitEvent)`	After the state's graph finishes, before the chosen transition fires.	State-duration timer stop.
`onTransition(TransitionEvent)`	A transition is selected — covers `on EVENT`, `on *`, `on_timeout`, and global transitions.	Transition counter, structured log.
`onWaitingForEvent(WaitingForEventEvent)`	Executor parks because no automatic transition matched.	"Awaiting signal" gauge, dashboard hint.
`onSignalReceived(SignalReceivedEvent)`	A `signal(event, payload)` call lands while the machine waits.	Signal-rate counter, payload audit.
`onStateTimeout(StateTimeoutEvent)`	A state-local `timeout` fires before any matching event arrives.	Per-state timeout metric.
`onGlobalTimeout(GlobalTimeoutEvent)`	The machine-wide `global_timeout` fires.	Alert / page.
`onCheckpointSaved(CheckpointSavedEvent)`	A checkpoint has been written to the configured store.	Replication watermark, lag check.
`onCheckpointRestored(CheckpointRestoredEvent)`	`resumeFromCheckpoint()` rehydrated an instance.	"Recovered after crash" log line.
`onStateMachineComplete(StateMachineCompleteEvent)`	Terminal state reached **or** machine entered `FAILED`.	Audit close, span close.

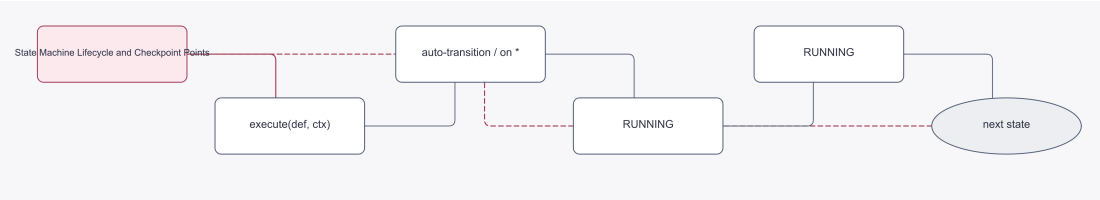
Register listeners via `StateMachineExecutor.Builder.listeners(...)` or, for the durable path, via `DurableStateMachineManager.Builder.listeners(...)`.

0.7 Checkpoint Shape

`StateMachineCheckpoint` is what the persistence SPI stores. Knowing the shape helps when you debug a partial recovery or design a custom `ExecutionCheckpointStateMachineStore`.

Field	Type	Notes
`instanceId`	`String`	Stable identity assigned by `execute(def, ctx, instanceId)` or generated.
`stateMachineName`	`String`	The `name` declared on the `state machine` block.
`currentStateId`	`String`	The state the machine is in **or** the last terminal state if completed.
`status`	`StateMachineStatus`	`RUNNING` / `WAITING_EVENT` / `COMPLETED` / `FAILED`.
`totalTransitions`	`int`	Increments on every transition; bounded by `max_transitions`.
`stateVisitCount`	`Map<String, Integer>`	Per-state entry count; bounded by `max_state_visits`.
`stateOutputs`	`Map<String, Map<String, Object>>`	The `ctx.stateName.output.nodeId` tree, namespaced per state.
`sharedContext`	`Map<String, Object>`	Cross-state context the operators write to.
`history`	`List<StateExecutionRecord>`	One record per entered-and-exited state, oldest first.
`startedAt`	`Instant`	Wall-clock at `execute(...)`.
`lastTransitionAt`	`Instant`	Wall-clock at the most recent transition.
`checkpointedAt`	`Instant`	Wall-clock at the most recent write.
`stateTimeoutDeadline`	`Instant?`	Set when the current state has a `timeout`; `null` otherwise.
`globalTimeoutDeadline`	`Instant?`	Set when the machine has a `global_timeout`.

Both deadline fields persist so that after a crash, the recovery loop can reschedule timers against the **original** wall-clock deadlines — not a new "now + timeout". This is what makes durable state machines correct across restarts.



0.8 SM / Session / Plain Graph

Use this table to pick the right primitive before you write any DSL.

Question	If yes → use
Is there a finite set of named lifecycle phases that the workload sits in for variable amounts of time, possibly looping?	**State machine** (Chapter 15)
Does the workload have a back-and-forth dialogue shape with the outside world, where each turn is processed by similar logic?	**Session** (Chapter 14)
Does the workload have **both** — phases that contain conversations?	**Composition** (Chapter 16). Pick the outer owner by signal origin (see Ch15).
Is the dependency shape a pure DAG that runs to completion in one pass?	**Plain graph** (Chapter 2 onward)
Is there one long-running operator that produces incremental output?	**Streaming** (Appendix C)

> *A state machine is the right tool when **what changes is the phase**, not the data shape. If the schema of the work item is what drives branching, a DAG is simpler.*

0.9 Real Examples

The repository ships verified fixtures you can read end-to-end.

File	What it shows
`order-lifecycle-state-machine.bloge`	4 states with a backward transition, 24h global timeout. The walk-through in Chapter 15.
`ticket-state-machine.bloge`	6 states, escalation paths, multiple event flows.
`order-session-with-state-machine.bloge`	Pattern A — session-outermost.
`review-state-machine-with-session.bloge`	Pattern B — state-machine-outermost (with the signal-deadlock constraint).

0.10 Common Mistakes

Mistake	Why it happens	Fix
Two `[initial]` markers in one machine.	Copy-pasted a draft and forgot to clear the marker.	The compiler will reject this. Run `bloge lint` before commit.
`max_transitions` exceeded in a retry loop.	Retry state visits itself with no exit condition.	Either lower the visit cap (`max_state_visits`) on the retry state, or add an `on * when ctx.retry.output.count > 3 -> giveUp` transition.
`signal(event, payload)` returns "no transition matched" silently.	The event name matched, but every `when` guard evaluated falsy.	Add a final `on EVENT -> fallback` after all guarded variants, or audit the guard's path with `MockOperator.invocations()` in a test.
Nested session in Pattern B tries to wait for an external signal.	Treating an inner session like an outer one.	The executor throws `SM_NESTED_SESSION_AWAITS_SIGNAL`. Restructure so the session's external touch-points live in the outer machine (Pattern A) — see Chapter 16.
State `timeout` fires but `on_timeout` is missing.	The state was added without a fallback.	The machine transitions to `FAILED`. Always pair `timeout` with `on_timeout` unless `FAILED` is the desired outcome.
Checkpoint restore lands in the wrong state.	Custom `ExecutionCheckpointState MachineStore` returned a stale row.	The store contract requires read-after-write for `instanceId`. Verify with `Durable StateMachineManager`'s built-in MyBatis store as a baseline.

0.11 Relationship to the Main Chapters

State machines do not live in one chapter; this appendix collects what is scattered:

- **Authoring** — Chapter 15 introduces the DSL.
- **Composition** — Chapter 16 covers the two nesting patterns and the signal-deadlock rule.
- **Durability** — Chapter 13 explains the store SPIs; this appendix names the checkpoint fields they carry.
- **Testing** — Chapter 18 covers deterministic time control and snapshot assertions.
- **Operations** — Chapter 19 shows the listener-driven metrics; this appendix gives you the callback catalog.

CHAPTER

—

Appendix E — Agent Extension

Chapter 17 teaches how to think about agents on top of bloge. This appendix is the lookup table you keep open while writing real ones: every DSL keyword, every record field, every SPI callback, every default value, and every finish-reason string the runtime can emit.

0.1 When You Need This Reference

Question	Section
What DSL keywords can I use inside `agent { … }`?	DSL Keyword Reference
How does a `tool { … }` block become an LLM tool schema?	Tool Block Reference
Which memory strategies exist and what do they cost?	Memory Strategies
What context variables can I use inside `exit_condition`?	Exit Condition
What can `AgentOutput.finishReason` actually contain?	Finish Reasons and Exceptions
Which callbacks fire and in what order?	AgentExecutionListener Cheat Sheet
How do I build the agent from Java instead of DSL?	Java Builder API
What chat-model SPIs do I have to satisfy?	LLM Provider Surface
What Spring auto-configuration kicks in when this module is on the classpath?	Spring Wiring
When should I reach for an agent versus a state machine or session?	Decision Table

This is the reference layer for **Chapter 17 — Agent Orchestration**. Start there for the conceptual walk-through.

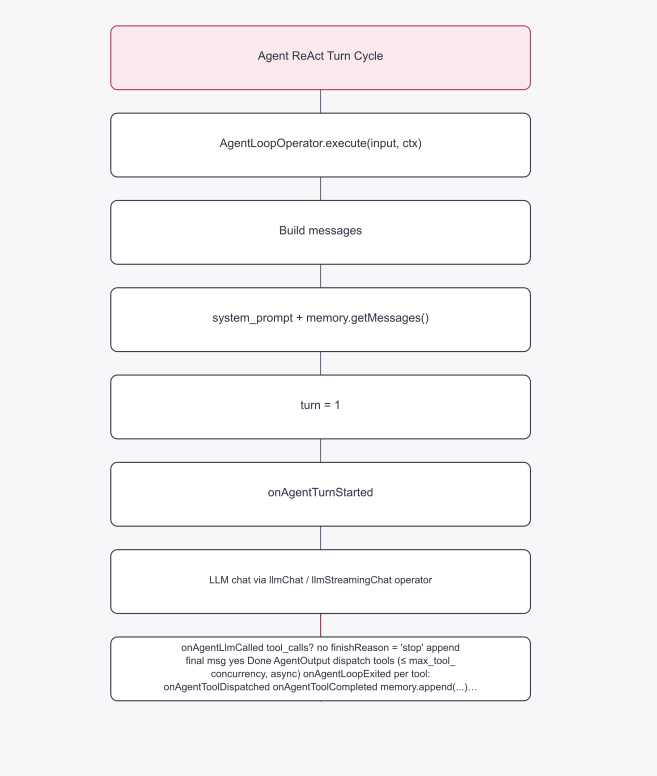
0.2 Module Surface

Property	Value
Maven coordinates	`com.leanowtech.bloge:bloge-agent-ext:${version}`
DSL extension kind	`"agent"`
Registration	`AgentExtensionProvider` discovered via `DslExtensionProvider` SPI
Hard dependencies	`bloge-core`, `bloge-dsl`
Runtime dependencies	`bloge-common-operators` (for `LlmProvider` / `llmChat` / `llmStreamingChat` / `TokenEstimator`); optional `JsonCodec` for tool argument deserialization

When the JAR is on the classpath the DSL compiler picks up the `agent` block syntax automatically. No manual wiring is required for compilation. At runtime, the agent loop expects a registered LLM chat operator (sync: `"llmChat"`; streaming: `"llmStreamingChat"`).

0.3 ReAct Turn Cycle

The loop, the listener callback points, and the streaming chunk types in one picture:



0.4 DSL Keyword Reference

Keyword	Scope	Type / value	Default	Purpose
`agent`	root, graph body, phase body, state body	block prefix	—	Declares an agent extension block.
`stream agent`	same as `agent`	block prefix	—	Shorthand that sets `streaming = true`.
`model`	agent body	`String`	**required**	LLM model identifier passed to `llmChat`.
`system_prompt`	agent body	`String`	`null`	Leading system message added to every LLM call.
`max_turns`	agent body	`int`	`10`	Hard cap on think-act-observe rounds. Exceeding it throws `MaxTurnsExceededException`.
`max_tool_concurrency`	agent body	`Integer`	`null` (unlimited)	Concurrent tool dispatch cap **per turn**.
`temperature`	agent body	`Double`	`null` (provider default)	Sampling temperature passed to `llmChat`.
`memory`	agent body	strategy expression	`full()`	Conversation memory strategy — see below.
`exit_condition`	agent body	DSL expression	`null` (never early-exit)	Evaluated after each tool round; truthy result exits the loop.
`streaming`	agent body	`boolean`	`false`	Materializes a `StreamingAgentLoopOperator` instead of `AgentLoopOperator`.
`streaming_buffer_capacity`	agent body	`Integer`	`16`	Internal LLM-token buffer size used by the streaming variant.

Keyword	Scope	Type / value	Default	Purpose
`tool <name> : <OperatorRef>`	agent body, repeatable	block	—	Declares one tool. The operator referenced becomes its implementation.
`description`	tool body	`String`	`""`	Tool description sent to the LLM.
`input { … }`	tool body	block of assignments	—	Maps tool-call arguments into operator inputs.
`tool_args`	inside `input { … }`	context variable	—	The arguments object provided by the LLM for a tool call.

0.4.1 Memory Strategies

`memory = …` accepts these forms (all backed by `AgentMemoryStrategy`):

DSL form	Behavior	Notes
`full()`	Keeps every message ever appended.	Default. Cheapest in CPU, most expensive in tokens.
`sliding_window(n)`	Keeps **all** system messages plus the last `n` non-system messages.	`n` must be ≥ 1.
`token_budget(n)`	Prunes oldest non-system messages until the `TokenEstimator` reports ≤ `n` total tokens.	Needs `TokenEstimator` (registered as an SPI; a heuristic fallback exists).
`summary(n)`	Keeps the last `n` non-system messages verbatim and replaces older ones with an LLM-generated summary.	Triggers a separate LLM call to summarize — measure cost.
`AgentMemoryStrategy.Custom(factory)`	Java only — supply your own `ConversationMemory.Factory`.	Not expressible in DSL.

0.4.2 Exit Condition

`exit_condition` is a DSL expression evaluated against this context after each tool round:

Variable	Type	Meaning
`finish_reason`	`String`	Finish reason from the most recent LLM response.
`content`	`String`	Assistant content string from the most recent LLM response.
`turn`	`int`	1-based turn number that just completed.
`tool_calls`	`List<Map>`	Tool calls produced this turn — each has `id`, `name`, `arguments`.
`tool_results`	`Map<String,Object>`	Results from the current tool batch, keyed by tool name.
`messages`	`List<LlmMessage>`	Full conversation state including newly-appended tool messages.

A truthy result records `"exit_condition"` in the listener's `LoopExitedPayload.exitReason` and stops the loop without another model call. The returned `AgentOutput.finishReason` still carries the latest provider reason.

0.5 Tool Block Reference

A tool block compiles into an `AgentToolRef`. Three things flow out of it:

1. The LLM-facing **`ToolDefinition`** — `name` (the DSL label), `description` (your `description = …`), and a JSON-schema `parameters` object derived from the `input { … }` block.
2. A **subgraph** that wraps the referenced operator. The compiler builds it so the agent loop can invoke the operator like any other bloge node.

3. The **`resultNodeId`** the loop uses to extract the tool's output from that subgraph.

0.5.1 Tool Schema Inference

The compiler walks each assignment in `input { … }`. Every path of the form `tool_args.<field>` becomes a **required** parameter in the JSON-schema. Other assignments (constants or references to outer context) do not appear in the schema — they are filled in at dispatch time.

```
tool searchKnowledgeBase : KBSearchOperator {
  description = "Search the knowledge base"
  input {
    query  = tool_args.query
    locale = ctx.user.locale     // not part of the LLM-visible schema
  }
}
```

The LLM sees a tool named `searchKnowledgeBase` with one required string parameter `query`. `locale` is filled from the surrounding graph context. If your tool inputs include nested objects, configure a `JsonCodec` on the DSL compiler so the arguments JSON can be deserialized into structured values.

0.6 Worked Example

The shortest end-to-end agent — DSL plus the Java bootstrap needed to run it.

```
agent customerSupport {
  model = "gpt-4o"
  system_prompt = "You are a helpful support agent."
  max_turns = 15
  temperature = 0.7
  memory = sliding_window(20)

  tool searchKnowledgeBase : KBSearchOperator {
    description = "Search the knowledge base"
    input { query = tool_args.query }
  }

  tool escalateToHuman : EscalateOperator {
    description = "Escalate to a human agent"
    input { reason = tool_args.reason }
  }

  exit_condition = finish_reason == "stop" || tool_call("escalateToHuman")
}
```

```
OperatorRegistry registry = OperatorRegistry.builder()
    .register("KBSearchOperator", new KBSearchOperator())
    .register("EscalateOperator", new EscalateOperator())
    .register("llmChat", new LlmChatOperator(myLlmProvider))
    .build();

AgentDef def = new AgentDslCompiler(registry).compile(dslSource);
AgentLoopOperator agent = new AgentLoopOperator("supportAgent", def, registry);

AgentOutput result = agent.execute("My account is locked", OperatorContext.root());
System.out.println(result.content());
System.out.println(result.finishReason());   // provider reason, e.g. "stop" / "tool_calls"
```

For the streaming variant call `toStreamingOperator(...)` on `AgentBuilder` or set `streaming = true` in DSL; the

runtime emits `AgentStreamChunk` values through a `NodeChannel`.

0.7 Java Builder API

`AgentBuilder` mirrors every DSL keyword:

Method	Purpose
`AgentBuilder.create(id)`	Start a fresh builder.
`AgentBuilder.from(def)`	Start from an existing `AgentDef`.
`.model(String)`	Sets the LLM model id.
`.systemPrompt(String)`	Sets the system prompt.
`.maxTurns(int)`	Sets the turn cap.
`.maxToolConcurrency(int)`	Sets the per-turn tool concurrency cap.
`.temperature(Double)`	Sets the sampling temperature.
`.memory(AgentMemoryStrategy)`	Picks a memory strategy.
`.tool(AgentToolRef)`	Adds one tool definition.
`.exitCondition(Expression | String)`	Compiles or attaches an exit-condition expression.
`.streaming(boolean)`	Toggles the streaming variant.
`.streamingBufferCapacity(int)`	Sets the streaming buffer size.
`.build()`	Produces an immutable `AgentDef`.
`.toOperator(ref [, registry])`	Materializes an `AgentLoopOperator`.
`.toStreamingOperator(ref [, registry])`	Materializes a `StreamingAgentLoopOperator`.

0.8 Record Schemas

Every record is immutable. Fields are listed in declaration order.

0.8.1 `AgentDef`

Field	Type	Notes
`id`	`String`	Identifier from the DSL.
`model`	`String`	LLM model id; required.
`systemPrompt`	`String`	May be `null`.
`maxTurns`	`int`	Default `10`.
`maxToolConcurrency`	`Integer`	`null` => unlimited.
`temperature`	`Double`	`null` => provider default.
`memoryStrategy`	`AgentMemoryStrategy`	Defaults to `Full`.
`tools`	`List<AgentToolRef>`	Order preserved from DSL.
`exitCondition`	`Expression`	May be `null`.
`streaming`	`boolean`	`true` => use streaming operator.
`streamingBufferCapacity`	`Integer`	Buffer for streaming variant, default `16`.

0.8.2 `AgentInput`

Field	Type	Notes
`messages`	`List<LlmMessage>`	Normalized form. `AgentInput.from(Object)` coerces strings, single messages, and lists.

0.8.3 `AgentOutput`

Field	Type	Notes
`content`	`String`	Final assistant content.
`finishReason`	`String`	See Finish Reasons.
`turnsUsed`	`int`	Turns the loop actually consumed.
`messages`	`List<LlmMessage>`	Full conversation, including system / assistant / tool messages.
`toolResults`	`Map<String,Object>`	Latest-turn tool results, keyed by tool-call id.
`checkpoint`	`AgentCheckpoint`	Snapshot for resume; never `null`.

0.8.4 `AgentCheckpoint`

Field	Type	Notes
`conversation`	`List<LlmMessage>`	Messages up to and including the snapshot point.
`turnCount`	`int`	Turn at the snapshot.
`toolResults`	`Map<String,Object>`	Tool results at the snapshot.

Persist this object alongside your own session/state-machine checkpoint to resume an agent run after restart.

0.8.5 `AgentToolRef`

Field	Type	Notes
`name`	`String`	Tool name exposed to the LLM.
`operatorRef`	`String`	Operator registered in `OperatorRegistry`.
`description`	`String`	Same string the LLM sees.
`graph`	`Graph`	Compiled subgraph wrapping the operator.
`resultNodeId`	`String`	Node whose output becomes the tool result.
`toolDefinition`	`LlmProvider.ToolDefinition`	LLM-facing schema (name + description + parameters).

0.8.6 `AgentStreamChunk` (sealed)

Variant	Fields	When
`Token`	`text: String, turnNumber: int`	One token (or token group) from the LLM.
`ToolStart`	`toolName: String, toolCallId: String, turnNumber: int`	Just before a tool is dispatched.
`ToolEnd`	`toolName: String, toolCallId: String, success: boolean, turnNumber: int`	Right after a tool completes.
`Done`	`finalOutput: AgentOutput`	Last chunk; the loop has exited.

0.9 Finish Reasons and Exceptions

Surface	Value and timing
`AgentOutput.finishReason`	Latest provider reason such as `"stop"`, `"length"`, or `"tool_calls"`; preserved unchanged.
`LoopExitedPayload.exitReason`	`"stop"`, `"exit_condition"`, or `"max_turns"`, describing why the loop exited.
`MaxTurnsExceededException`	Thrown directly at `max_turns` with a checkpoint; there is no normal output whose finish reason is `"max_turns"`.

`MaxTurnsExceededException` is **thrown** from `AgentLoopOperator.execute` — the boundary code is responsible for catching it and surfacing a meaningful error to the caller.

0.10 `AgentExecutionListener` Cheat Sheet

Callback	Payload (record)	Fields
`onAgentTurnStarted`	`TurnStartedPayload`	`turnNumber`, `messageCount`
`onAgentLlmCalled`	`LlmCalledPayload`	`turnNumber`, `model`, `promptTokens`, `completionTokens`, `latencyMs`, `finishReason`
`onAgentToolDispatched`	`ToolDispatchedPayload`	`turnNumber`, `toolCallId`, `toolName`, `argumentsJson`
`onAgentToolCompleted`	`ToolCompletedPayload`	`turnNumber`, `toolCallId`, `toolName`, `success`, `latencyMs`, `resultSummary`
`onAgentLoopExited`	`LoopExitedPayload`	`totalTurns`, `totalLlmCalls`, `totalLlmTokens`, `totalLatencyMs`, `exitReason`

Every callback receives an `AgentCallbackContext` with `executionId`, `graphName`, `nodeId`, `operatorRef`. All callbacks have default no-op implementations; override only the ones you care about.

The order in a typical turn is: `onAgentTurnStarted` → `onAgentLlmCalled` → (per tool call) `onAgentToolDispatched` → (per tool call) `onAgentToolCompleted` → next turn or `onAgentLoopExited`.

0.11 `ConversationMemory` SPI

`ConversationMemory` is a sealed interface (`FullConversationMemory`, `SlidingWindowConversationMemory`, `TokenBudgetConversationMemory`, `SummaryConversationMemory`). Construct an instance with one of the static factory methods:

Factory	Notes
`ConversationMemory.full(initialMessages)`	No pruning.
`ConversationMemory.slidingWindow(initialMessages, windowSize)`	Keeps all system messages plus the last `windowSize` non-system.
`ConversationMemory.tokenBudget(initialMessages, maxTokens, tokenEstimator)`	Prunes from the oldest non-system message until total tokens fit.
`ConversationMemory.summary(initialMessages, summarizer, summaryThreshold)`	Calls `summarizer` once the non-system count exceeds `summaryThreshold`.

Two functional SPIs let you plug in custom behavior:

- `ConversationMemory.Factory` — `create(List<LlmMessage> initial)` lets you return any `ConversationMemory` instance and is the hook for `AgentMemoryStrategy.Custom(factory)`.
- `ConversationMemory.Summarizer` — `summarize(List<LlmMessage>)` is what the `Summary` strategy calls. The bloge runtime does not include a default summarizer; wire your own (typically an `llmChat` call against the same model with a "summarize the conversation" prompt).

0.12 LLM Provider Surface (bloge-common-operators)

The agent loop never talks to OpenAI / Anthropic / Azure directly. It calls operators registered in `OperatorRegistry` under standard names. The contract lives in `bloge-common-operators`:

Type	FQN	Purpose
`LlmProvider`	`com.leanowtech.bloge.operators.spi.LlmProvider`	SPI for the underlying chat-model client.
`LlmProvider.LlmMessage`	nested record	`role`, `content`, `parts`, `toolCalls`, `toolCallId`.
`LlmProvider.ToolDefinition`	nested record	`name`, `description`, `parameters` (JSON-schema map).
`LlmChatOperator`	`com.leanowtech.bloge.operators.ai.LlmChatOperator`	Synchronous chat operator; default registration name `"llmChat"`.
Streaming chat operator	(same module)	Registered as `"llmStreamingChat"` for the streaming agent variant.
`TokenEstimator`	`com.leanowtech.bloge.operators.spi.TokenEstimator`	Estimates message token count for `token_budget` memory. A heuristic default ships in the module.

If you write a brand-new provider, implement `LlmProvider` and pass it into `LlmChatOperator` when you register the operator. The agent loop will route through it unchanged.

0.13 Spring Wiring

`bloge-agent-ext` ships no Spring auto-configuration of its own. Two conditional hooks in `bloge-spring` cover everything Spring users need:

Bean	Defined in	Activation
`AgentEventJournalBridge`	`BlogeEventJournalAutoConfiguration`	Registered when `bloge-agent-ext` is on the classpath, `bloge-event-journal` is on the classpath, and `spring.bloge.event-journal.enabled=true`. Pipes every `AgentExecutionListener` callback into the engine's event journal.
Optional listener wiring	`OptionalAgentListenerSupport`	Soft class-loads `AgentExecutionListener` so the rest of `bloge-spring` works without the agent-ext JAR. Any `AgentExecutionListener` beans in the context are picked up automatically.

There is exactly one property:

```yaml
spring.bloge:
  event-journal:
    enabled: true
```

Beyond that, expose your provider, your operators, and your listeners as ordinary Spring beans — they will be picked up via the usual `OperatorRegistryAutoConfiguration` path from Chapter 20.

0.14 Decision Table

Where does the agent loop fit alongside other bloge concepts?

Question	Reach for
Do I have a fixed pipeline of operators that always runs in the same order?	Plain bloge graph. No agent.
Do I need an LLM to choose **which** operators to call, in **what** order, with arguments **it** synthesizes?	`agent { … }`.
Do I need to drive a deterministic workflow through phases with guard-based transitions?	**State machine** (Appendix D).
Do I need crash-safe multi-turn conversation with the user (not the LLM picking tools)?	**Session** (Chapter 14).
Do I need streaming tokens out to a UI?	Streaming variant — `streaming = true` or `stream agent`. See Appendix C for the broader streaming model.
Do I need fine-grained traces for LLM calls, tool dispatches, and timings?	`AgentExecutionListener` + `AgentEventJournalBridge`.

Agents and state machines compose: an agent can be a single state's operator, and a state machine can route through several specialized agents.

0.15 Common Mistakes

- **Forgetting `model = "…"`.** Compilation fails. The model id is required and passed verbatim to the chat operator.
- **No `llmChat` operator registered.** The DSL compiles, but the loop fails at the first turn with an "unknown operator" error. Wire a `LlmChatOperator` (or `llmStreamingChat` for streaming) when you build the registry.
- **`tool_args.x` outside an `input { … }` block.** `tool_args` is only in scope inside a tool's `input` block. Using it in `exit_condition` or outside any tool is a compile error.
- **Unbounded `full()` memory on long conversations.** Every turn re-sends the entire history. Switch to `sliding_window(n)` or `token_budget(n)` before going to production.
- **`summary()` strategy with no `Summarizer` wired.** The default `Summary` strategy expects a summarizer callback (typically an LLM call). Without one configured, the strategy degrades to keeping the threshold windowed view only.
- **`exit_condition` that can never be true.** If `exit_condition` references a tool result key that no tool ever produces, the loop runs to `max_turns` every time. Test with both the happy-path and the cap path.
- **JSON-schema mismatch for nested-object tool inputs.** If your tool needs structured input, supply a `JsonCodec` to the DSL compiler so `tool_args.foo.bar` resolves through the deserialized object. Without a codec, only top-level string fields work cleanly.
- **`MaxTurnsExceededException` not caught at the boundary.** The exception propagates up through `execute()`. Operator code calling the agent must catch it and surface a meaningful failure (otherwise the surrounding graph marks the node as crashed).

0.16 Relationship to Main Chapters

- **Chapter 7 — Designing Good Operators.** The operator metadata fields (`promptHint`, `usageExample`, `constraintsDescription`) are what the agent loop reads to populate `description` and the JSON-schema hints surfaced to the LLM as tool definitions.
- **Chapter 11 — Batch and Iteration** and **Appendix C — Streaming.** The streaming variant emits `AgentStreamChunk` values through the same `NodeChannel` mechanism Appendix C documents.
- **Chapter 19 — Observability in Production.** `AgentEventJournalBridge` hooks every callback into the event journal Chapter 19 describes.
- **Chapter 9 — Tooling Workflow.** The `operator-metadata.json` exporter describes each registered operator in the same shape the agent loop sends to the model as a `ToolDefinition`.
- **Chapter 20 — Spring and Production Wiring.** Conditional bean registration described above; `OperatorRegistryAutoConfiguration` is the standard path for exposing operators and listeners.
- **Chapter 17 — Agent Orchestration.** The conceptual chapter and worked walk-through this appendix backs.

CHAPTER

—

Appendix F — Decision Table

This appendix is the authoritative quick-reference for the `decision_table` node type. Every syntax rule, hit-policy guarantee, output shape, and error code listed here is derived directly from bloge source code and conformance fixtures.

Use this appendix when you need to look up exact semantics without re-reading Chapter 5. For a guided walk-through, start with Chapter 5 — Branches That Decide. For a hands-on exercise, see Lab 6 in Chapter 34.

0.1 Syntax Cheat Sheet

```
decision_table <id>(
  <param1> = <expr1>,
  <param2> = <expr2>,
  ...
) [hit=<first|unique|any|collect>] [-> <OutputType>] {
  rule (<param>: <predicate>, ...) -> <output>
  ...
  [otherwise -> <output>]
}
```

- `hit=` defaults to `first` when omitted.
- `-> <OutputType>` is optional; the compiler infers the type from the first rule's output when absent.
- Every `rule` names the parameters it tests inside `(...)`.
- `otherwise` is evaluated only after all explicit `rule` entries fail (or, for `collect`, it is always appended as an additional item — see §Otherwise Semantics).

Minimal example — `hit=first` with scalar output:

```
graph creditScreening {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

0.2 Hit Policy Matrix

Policy	Multiple matches	No match, no `otherwise`	Exception thrown
`first`	Takes the **first** match in declaration order	`RUNTIME_DECISION_TABLE_NO_MATCH`	`CODE_NO_MATCH`
`unique`	Throws `RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH`	`RUNTIME_DECISION_TABLE_NO_MATCH`	`CODE_AMBIGUOUS_MATCH` / `CODE_NO_MATCH`
`any`	Allowed **if all matching outputs are equal** (`Objects.equals`); otherwise `RUNTIME_DECISION_TABLE_CONFLICTING_MATCH`	`RUNTIME_DECISION_TABLE_NO_MATCH`	`CODE_CONFLICTING_MATCH` / `CODE_NO_MATCH`
`collect`	Collects **all** matches into `output.items`	Returns `{ items: [] }` — no exception	—

`first` example — first rule that fires wins:

```
graph decisionBasicFirst {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

`unique` example — at most one rule may fire; multiple matches are an error:

```
graph decisionBasicUnique {
  decision_table credit_band(score = applicant.output.score) hit=unique -> String {
    rule (score: score >= 750) -> "prime"
    rule (score: 650 <= score < 750) -> "standard"
    otherwise -> "manual"
  }
}
```

`any` example — multiple rules may fire provided their outputs agree:

```
graph decisionBasicAny {
  decision_table approval(score = ctx.score) hit=any -> String {
    rule (score: score >= 700) -> "approved"
    rule (score: score >= 750) -> "approved"
    otherwise -> "manual"
  }
}
```

`collect` example — all matching rules contribute to `output.items`:

```
graph decisionBasicCollect {
  decision_table applicable_discounts(score = applicant.output.score) hit=collect -> String {
    rule (score: score >= 700) -> "loyalty"
    rule (score: score >= 760) -> "premium"
  }
}
```

0.3 Otherwise Semantics

Policy	otherwise present	otherwise absent
first	Fires when no explicit rule matches	No match → RUNTIME_DECISION_TABLE_NO_MATCH
unique	Fires when no explicit rule matches	No match → RUNTIME_DECISION_TABLE_NO_MATCH
any	Fires when no explicit rule matches	No match → RUNTIME_DECISION_TABLE_NO_MATCH
collect	**Always appended** as an additional item, regardless of explicit matches	Empty match → returns { items: [] }, no exception

> ***The collect asymmetry**: For first, unique, and any, otherwise is a fallback that fires only when nothing else matched. For collect, it is an unconditional rule that always appends its output to items. This distinction is flagged by the decision-table/collect-otherwise lint rule (INFO severity) so you can decide whether the behaviour is intentional.*

0.4 Output Shape Reference

Three shapes are available. All rule bodies (including otherwise) in a single table **must use the same shape** — mixing shapes is a compile error (COMPILER_DECISION_TABLE_OUTPUT_SHAPE_MISMATCH).

Shape	Declaration syntax	Downstream access
Scalar	-> "value" / -> 3.5	<node>.output.value
Named fields	-> { rate: 3.5, maxTerm: 30 }	<node>.output.rate, <node>.output.maxTerm
Collect items	hit=collect + any output	<node>.output.items (a List)

Multi-output (named fields) example:

```
graph decisionMultiOutput {
  decision_table loan_terms(
    score = applicant.output.score,
    amount = loan.output.amount
  ) hit=unique -> { rate: Decimal, maxTerm: Int } {
    rule (score: score >= 750, amount: amount <= 500000) -> { rate: 3.5, maxTerm: 30 }
    otherwise -> { rate: 6.0, maxTerm: 5 }
  }
}
```

0.5 The in Operator

in tests set membership inside a rule predicate. There are two forms.

0.5.1 Form A — Literal array

```
rule (type: type in ["vip", "enterprise", "gov"]) -> "priority"
```

- The right-hand side is a **literal array**: [lit1, lit2, ...].
- Each element must be a **String, Number, or Boolean literal** — no expressions, variables, or node paths are allowed.
- Violating this causes a compile error: COMPILER_DECISION_TABLE_IN_LITERAL_ONLY.

- An **empty array** (`type in []`) never matches any input.

Example:

```
graph decisionTableInStatic {
  decision_table customer_tier(type = ctx.type) hit=first -> String {
    rule (type: type in ["vip", "enterprise", "gov"]) -> "priority"
    otherwise -> "standard"
  }
}
```

0.5.2 Form B — Dynamic parameter reference

```
rule (type: type in allowed) -> "priority"
```

- The right-hand side is a **single declared parameter name** that resolves to a `Collection` at runtime.
- Deep paths like `ctx.allowed`, `obj.list`, or `a[0]` are **not** permitted — use the parameter binding in the `decision_table(...)` head to forward the collection.
- If the parameter resolves to a non-`Collection` type at runtime, bloge throws `DecisionTableViolationException` with code `RUNTIME_DECISION_TABLE_INVALID_COLLECTION_PARAM`. The exception message includes the line and column of the `in` expression and the actual runtime type.
- A `null` left-hand value always evaluates to `false` (short-circuit, no exception).

Example:

```
graph decisionTableInDynamic {
  decision_table customer_tier(type = ctx.type, allowed = ctx.allowed) hit=first -> String {
    rule (type: type in allowed) -> "priority"
    otherwise -> "standard"
  }
}
```

0.5.3 Equality semantics — both forms

Both Form A and Form B use **numeric-aware equality** internally (`evaluateBinaryOp(EQ_EQ)`). This means `Integer(700)` and `BigDecimal(700.0)` are considered equal. You will never get a silent mismatch due to numeric type differences from different graph nodes.

0.6 Common Pitfalls

0.6.1 1. Scope violation — referencing a non-parameter path in a rule

```
// [X] COMPILE ERROR: COMPILER_DECISION_TABLE_RULE_SCOPE_VIOLATION
rule (score: applicant.output.score >= 750) -> "prime"

// [OK] bind the value as a parameter first
decision_table credit_band(score = applicant.output.score) hit=unique -> String {
  rule (score: score >= 750) -> "prime"
  ...
}
```

0.6.2 2. Output shape mismatch

```
// [X] COMPILE ERROR: COMPILER_DECISION_TABLE_OUTPUT_SHAPE_MISMATCH
```

```
decision_table tier(score = ctx.score) hit=first {
  rule (score: score >= 750) -> "prime"          // scalar
  otherwise -> { label: "manual", flag: true }   // named fields — shape mismatch!
}
```

0.6.3 3. `in` right-hand side is not a plain parameter name

```
// [X] COMPILE ERROR: COMPILER_DECISION_TABLE_IN_INVALID_RIGHT
rule (type: type in ctx.allowed) -> "priority"   // deep path not allowed

// [OK] bind it in the parameter list
decision_table tier(type = ctx.type, allowed = ctx.allowed) hit=first -> String {
  rule (type: type in allowed) -> "priority"
  ...
}
```

0.6.4 4. `in` right-hand side is not a `Collection` at runtime

If the parameter bound to the right side of `in` resolves to a non-`Collection` value (e.g., a `String` or `Integer`), bloge throws:

`RUNTIME_DECISION_TABLE_INVALID_COLLECTION_PARAM`

The exception message identifies the exact line, column, and the actual runtime type. Fix the upstream node or graph expression that produces the wrong type.

0.6.5 5. `collect` + `otherwise` — always appends, never just falls back

```
// [INFO] decision-table/collect-otherwise
// The otherwise clause here appends "baseline" to items on EVERY run,
// even when other rules also matched. This may be intentional, but verify.
decision_table discounts(score = ctx.score) hit=collect -> String {
  rule (score: score >= 700) -> "loyalty"
  otherwise -> "baseline"   // always appended
}
```

0.7 Runtime Error Codes

All runtime violations are thrown as `DecisionTableViolationException`. The `code` field carries one of these frozen wire strings:

Wire string	When thrown
`RUNTIME_DECISION_TABLE_NO_MATCH`	`first`, `unique`, or `any` — no rule matched and no `otherwise` is declared
`RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH`	`unique` — more than one rule matched
`RUNTIME_DECISION_TABLE_CONFLICTING_MATCH`	`any` — multiple rules matched with different output values
`RUNTIME_DECISION_TABLE_INVALID_COLLECTION_PARAM`	`in <param>` — the parameter does not resolve to a `Collection`

> *These strings are **frozen contracts**. Copy them verbatim when matching in application error handlers or test assertions.*

0.8 Compiler Error Codes

The bloge compiler rejects invalid decision table definitions at compile time. Key error identifiers (used in diagnostics and CI lint output):

Identifier	Cause
`COMPILER_DECISION_TABLE_IN_LITERAL_ONLY`	`in [...]` array contains a non-literal element
`COMPILER_DECISION_TABLE_IN_INVALID_RIGHT`	`in <expr>` right side is not a single parameter name
`COMPILER_DECISION_TABLE_RULE_SCOPE_VIOLATION`	A rule predicate references something outside the declared parameters
`COMPILER_DECISION_TABLE_OUTPUT_SHAPE_MISMATCH`	Rules use inconsistent output shapes (scalar vs. named)
`COMPILER_DECISION_TABLE_OUTPUT_TYPE_INVALID`	A named output type annotation contains a field without the required `name: Type` form
`COMPILER_DECISION_TABLE_UNKNOWN_PARAMETER`	A rule condition names a parameter not declared in the table head
`COMPILER_DECISION_TABLE_DUPLICATE_CONDITION`	Two rules have identical conditions
`COMPILER_DECISION_TABLE_PARAM_RESERVED`	A parameter name collides with a DSL keyword
`COMPILER_DECISION_TABLE_PARAM_DUPLICATE`	Duplicate parameter name in the table head
`COMPILER_DECISION_TABLE_PARAM_NODE_CONFLICT`	A parameter name shadows a graph node name

0.9 Minimal Runnable Examples

0.9.1 Example 1 — `hit=first` with `otherwise`

```
graph ex1First {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

Expected: `score=720` → `credit_tier.output.value == "gold"`.

0.9.2 Example 2 — `hit=unique` — ambiguous-match error path

```
graph ex2Unique {
  decision_table risk_band(score = ctx.score) hit=unique -> String {
    rule (score: score >= 650) -> "eligible"
    rule (score: score >= 700) -> "preferred"
    // No otherwise — if score >= 700 both rules fire → RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH
  }
}
```

Expected: `score=720` → `DecisionTableViolationException` with `RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH`.

0.9.3 Example 3 — `hit=collect` with named-field output

```
graph ex3Collect {
  decision_table loan_badges(
    score = applicant.output.score,
    amount = loan.output.amount
  ) hit=collect -> { label: String, discount: Decimal } {
    rule (score: score >= 700) -> { label: "loyalty", discount: 0.02 }
```

```
    rule (score: score >= 760) -> { label: "premium", discount: 0.05 }
    rule (amount: amount <= 100000) -> { label: "small-loan", discount: 0.01 }
  }
}
```

Expected: `score=780, amount=80000` → `loan_badges.output.items` contains three entries: `loyalty`, `premium`, `small-loan`.

0.10 See Also

- Chapter 5 — Branches That Decide — conceptual introduction, branch-vs-decision-table selection guide
- Chapter 9 — Tooling Workflow — lint recipes for `decision-table/missing-otherwise` and `decision-table/collect-otherwise`
- Chapter 34 — Labs, Lab 6 — hands-on exercise covering `unique`, `collect`, and exception paths
- Chapter 17 — Putting a Nondeterministic Agent Inside a Deterministic Lifecycle — composing an agent signal with a decision table for auditable policy enforcement

CHAPTER

—

Appendix G — DSL Engineering Reference

Chapter 8 teaches how a personal draft becomes a team asset. This appendix keeps engineering lookup detail out of that first-reading path.

0.1 Imports and aliases

Prefer an explicit alias:

```
import "./payment-flow" as paymentFlow
```

Without one, the compiler may derive a name from the final path segment. Team-owned files should still declare it so renaming a file does not silently rename every reference. `FileSystemGraphResolver` resolves relative paths from the current file. Missing files, cycles, and duplicate aliases are graph-construction errors.

0.2 Schema lookup

Fact to express	Form
Required scalar	`field: String`
Optional scalar	`field: String?`
Reused shape in one file	named `schema`
One-off local shape	inline `{ ... }`
Homogeneous collection	`List<T>`
Key/value shape	`Map<K,V>`

RC1 path validation can diagnose each segment of an inline nested shape. The ch08 probe therefore uses an inline `customer`; do not turn this observation into a promise about every future release.

0.3 `ValidatedSchema` is not DSL syntax

The Java runtime can combine an expected schema with `STRICT`, `COERCE`, or `LENIENT` policy through `ValidatedSchema`. The DSL compiler reads the expected schema for structural checks; runtime policy decides how values are validated or converted. Keep the claims separate: a correct compile-time path does not prove that a runtime value satisfies policy.

0.4 Resolver extension boundary

A custom `GraphResolver` may load graphs from a controlled repository, classpath, or another versioned source. It should provide deterministic identity, explicit failure, and cycle protection. Remote fetching, cache invalidation, and signature validation remain host-system responsibilities; import syntax does not provide them automatically.

0.5 Source entry points

- `DslCompiler.java`
- `FileSystemGraphResolver.java`
- `ValidatedSchema.java`
- `DslFeedbackLoopProbeTest.java`

CHAPTER

—

Appendix H — Durable and Scale Runtime Reference

Chapters 21 and 22 keep mechanisms and decisions in the reading path. This appendix keeps deployment checks and measurement parameters.

0.1 Lease and recovery configuration

Record owner identity, claim timeout, scan interval, batch size, maximum recovery attempts, and logical/wall-clock source. A shorter timeout is not automatically safer: long pauses and store jitter increase false takeover, so timeout must be paired with fencing epochs.

0.2 Routing and migration

`ShardResolver` chooses a location for a new business key; existing keys read `RoutingStore`. Migration order is: stop writes, copy complete state, validate, atomically update the binding, resume traffic. A new hash after scaling is not migration.

0.3 Remote dispatch parameters

Parameter	Record with it
topic / queue	worker capability and version
heartbeat	false-positive window and redelivery policy
batch size	global in-flight ceiling
retry	effect idempotency and duplicate delivery
result timeout	cancellation, late result, and DLQ handling

0.4 Store migration check

Use the Flyway scripts packaged with the selected `bloge-durable-mybatis` artifact as authority; an old chapter's V25 name is not a permanent contract. Per environment, record schema history, table/index presence, application artifact checksum, migration time, and rollback decision.

0.5 Benchmark record template

```
commit / artifact:
JDK / CPU quota / heap:
```

```
topology / operator behavior:
arrival model / concurrency:
warmup / measurement / forks:
external stubs and store:
raw result path:
single changed axis:
stop rule:
claim and non-claim:
```

`benchmarks/run.sh` and `compare.py` provide run and comparison entry points. Read thresholds from the current script or team baseline; copying a number from the book does not make it current.

0.6 Exact entry points

- `benchmarks/run.sh`
- `benchmarks/compare.py`
- `RecoveryLoopConfig.java`
- `RuntimeRoutingShardingTest.java`

CHAPTER

—

Appendix I — Spring Production Reference

Use this appendix after Chapter 20 works. It is a lookup surface for production assembly, not a second tutorial path.

Return to Chapter 20 when the ownership model is unclear.

0.1 Break It Apart

0.1.1 What the starter auto-configures

The auto-configuration is registered via the Spring Boot 3.x `AutoConfiguration.imports` file, which lists ten top-level auto-configuration entries. The table also shows `BlogeCoreAutoConfiguration`, which `BlogeAutoConfiguration` imports internally:

Configuration class	What it provides	Gate condition
`BlogeAutoConfiguration`	Entry point — imports `BlogeCoreAutoConfiguration`	Always active
`BlogeCoreAutoConfiguration`	`OperatorRegistry`, `GraphLoader`, `List<Graph>`, tenant resolver, tenant filter	Always active
`BlogeDurableMigrationConfiguration`	Explicit encrypted-checkpoint migration tooling	`bloge.decrypt-checkpoints=true` + migration classes on classpath
`BlogeGraphEngineAutoConfiguration`	`GraphEngine` via builder with all `ObjectProvider` collaborators	Active unless checkpoint decryption migration mode is enabled
`BlogeDurableAutoConfiguration`	All durable stores, `RecoveryConfig`, `RoutingStore`, Flyway migrations	`bloge-durable` on classpath + `DataSource` bean
`BlogeAuditAutoConfiguration`	`AuditJournalListener`	`spring.bloge.audit.enabled=true` + `AuditJournalStore` bean present
`SessionExecutorAutoConfiguration`	Pure `SessionExecutor`; optional durable session adapter and manager	`bloge-session-ext` on classpath; durable beans require durable classes plus stores and definition lookup
`BlogeStateMachineAutoConfiguration`	Durable state-machine manager and recovery wiring	durable state-machine class on classpath + `spring.bloge.state-machine.enabled=true`
`BlogeEndpointAutoConfiguration`	`/actuator/bloge`, `/actuator/blogeStats`	`spring-boot-actuator` on classpath
`BlogeObservabilityAutoConfiguration`	Metrics, tracing, MDC, logging, health indicator	`bloge-metrics-otel` on classpath
`BlogeEventJournalAutoConfiguration`	Event journal retention and lifecycle	`bloge-event-journal` on classpath

If you're wiring multi-turn session flows, Chapter 14 — Multi-Turn Sessions explains the runtime model that `Session`

`ExecutorAutoConfiguration` is bootstrapping: session graphs, phase/round execution, idle timeouts, and optional durable recovery.

0.1.2 The `@BlogeOperator` annotation

Defined in `BlogeOperator.java`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface BlogeOperator {
    @AliasFor(annotation = Component.class, attribute = "value")
    String value() default "";
    String description() default "";
    String owner() default "";
    String[] tags() default {};
    String promptHint() default "";
    String usageExample() default "";
    String constraintsDescription() default "";
}
```

It is `@Component` underneath, so Spring discovers it. The `BlogeAutoConfiguration.operatorRegistry()` bean method then scans all beans annotated with `@BlogeOperator` and registers each one in a `DefaultOperatorRegistry` keyed by the annotation's `value()`.

> ***For agent/LLM integration**, three additional attributes describe the operator to an AI agent: `promptHint` tells the agent when to choose this operator; `usageExample` gives a DSL snippet showing the operator in context; `constraintsDescription` documents constraints not obvious from the schema. All three are optional.*

0.1.3 How `GraphEngine` is assembled

The engine bean is created by `BlogeGraphEngineAutoConfiguration`, which accepts `tenantContextResolver` and `tenantResourcePolicy` for multi-tenant deployments and exposes a `GraphEngineCustomizer` callback hook so you don't have to replace the entire engine bean to tweak it.

The `graphEngine()` bean method in `BlogeGraphEngineAutoConfiguration` uses `ObjectProvider` for every optional collaborator:

```
@Bean
@ConditionalOnMissingBean
public GraphEngine graphEngine(
        OperatorRegistry registry,
        ObjectProvider<OperatorInterceptor> interceptors,
        ObjectProvider<ExecutionListener> listeners,
        ObjectProvider<CallerContextCarrier> contextCarriers,
        ObjectProvider<CheckpointCodec> checkpointCodec,
        ObjectProvider<VersionRoutingPolicy> versionRoutingPolicy,
        ObjectProvider<TenantContextResolver> tenantContextResolver,
        ObjectProvider<TenantResourcePolicy> tenantResourcePolicy,
        ObjectProvider<GraphEngineCustomizer> customizers,
        /* ... more ObjectProviders ... */) {
    GraphEngine.Builder builder = GraphEngine.builder()
            .registry(registry)
            .interceptors(interceptors.orderedStream().toList())
            .listeners(listeners.orderedStream().toList())
            .contextCarriers(contextCarriers.orderedStream().toList())
            .checkpointCodec(checkpointCodec.getIfAvailable())
            .versionRoutingPolicy(versionRoutingPolicy.getIfAvailable())
```

```
            .tenantContextResolver(tenantContextResolver.getIfAvailable())
            .tenantResourcePolicy(tenantResourcePolicy.getIfAvailable());

    if (properties.getEngineMode() == BlogeProperties.EngineMode.REQUEST_RESPONSE) {
        builder.requestResponseDefaults();
    }

    customizers.orderedStream().forEach(c -> c.customize(builder));
    return builder.build();
}
```

This means **any** `OperatorInterceptor`, `ExecutionListener`, `CallerContextCarrier`, `TenantContextResolver`, or `TenantResourcePolicy` you declare as a Spring bean is automatically picked up by the engine. You never need to touch the engine builder.

`GraphEngineCustomizer` is a callback interface that lets you fine-tune the `GraphEngine.Builder` after all auto-configured defaults and engine-mode presets have been applied — without replacing the entire engine bean:

```
@Bean
public GraphEngineCustomizer myCustomizer() {
    return builder -> builder
            .schedulerTimerSupport(myTimerSupport);
}
```

0.1.4 Core configuration properties

Declared in `BlogeProperties.java` under the `spring.bloge` prefix:

```
spring:
  bloge:
    dsl-locations: classpath:bloge/    # where .bloge files are scanned
    hot-reload: false                  # file-watch and recompile on change
    default-timeout: 30s               # default node timeout
    accessor-mode: method-handle       # expression compilation mode
    engine-mode: auto                  # engine preset (auto | request-response)
    streaming-default-buffer-size: 16  # streaming buffer capacity
```

`engine-mode` and `streaming-default-buffer-size`. *`engine-mode` selects a high-level preset for the engine builder. `AUTO` (the default) uses standard settings; `REQUEST_RESPONSE` applies optimised defaults for short-lived, request-scoped graph executions. `streaming-default-buffer-size` controls the default back-pressure buffer for streaming graph executions.*

0.1.5 Tenant properties

When `spring.bloge.tenant.enabled=true`, the starter creates a `SpringTenantContextResolver` and `SpringTenantResourcePolicy` bean that feed into the engine's multi-tenant admission control:

```
spring:
  bloge:
    tenant:
      enabled: false                    # master switch
      mode: header                      # header | security-context | static
      header-name: X-Tenant-Id          # HTTP header for tenant resolution
      namespace-header-name: X-Namespace # HTTP header for namespace resolution
      default-tenant: default           # fallback tenant ID
      default-namespace: default        # fallback namespace
      max-concurrent-executions: 100    # per-tenant concurrency cap
      max-starts-per-minute: 0          # per-tenant rate limit (0 = unlimited)
```

The `mode` property selects how the tenant is resolved:

Mode	How it resolves the tenant
`HEADER`	Reads `X-Tenant-Id` and `X-Namespace` from the HTTP request headers
`SECURITY_CONTEXT`	Extracts `tenantId` / `namespace` from the Spring Security `Authentication.getDetails()`
`STATIC`	Always returns `default-tenant` / `default-namespace`

When tenant support is enabled, the starter also registers a `TenantContextFilter` — a servlet filter that binds the resolved `TenantContext` to the request thread via `TenantContextHolder.callWith()`, so store queries and default `GraphContext` construction see the same tenant for the entire HTTP request.

0.1.6 Version-routing properties

```yaml
spring:
  bloge:
    version-routing:
      policy: latest                  # latest | pinned | canary
      pinned-version: null            # required when policy=pinned
      canary-primary: null            # primary version for canary
      canary-candidate: null          # candidate version for canary
      canary-percentage: 10           # traffic % routed to candidate
```

0.1.7 Durable routing properties

```yaml
spring:
  bloge:
    durable:
      routing:
        mode: single                  # single | hash | tenant
        tenant-strategy: null         # shared-table | separate-schema | separate-database
        default-shard-id: default
        fallback-shard-id: null
        shard-ids: []
        tenant-shards: {}             # map of tenantId → shardId
```

0.1.8 Durable store wiring

When `bloge-durable` is on the classpath and a `DataSource` bean exists, the starter creates a `durableSqlSessionFactory` bean, runs Flyway migrations (if `spring.bloge.durable.flyway.enabled=true`, the default), and registers ten+ runtime store beans — each with `@ConditionalOnMissingBean` so you can override any of them. The full list includes `ExecutionStore`, `ExecutionCheckpointStore`, `WaitStore`, `WorkItemStore`, `GraphRegistryStore`, `RoutingStore`, `TaskInboxStore`, `EventMatcherStore`, `EventDeduplicationStore`, `ArchiveService`, and `DurableControlPlaneService`.

0.1.9 Health indicator

`BlogeHealthIndicator` reports `UP` when at least one graph is loaded and `DOWN` with `"reason": "No graphs loaded"` when the list is empty. It appears at `/actuator/health` under the `bloge` component.

0.2 Error Responses — RFC 9457 ProblemDetail

When `bloge-spring-web` is on the classpath, every BLOGE exception is mapped to a **RFC 9457 `ProblemDetail`** so clients see a uniform machine-readable error format, not stack traces.

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/problem+json
```

```
{
  "type":   "https://bloge.dev/problems/graph-version-mismatch",
  "title":  "Graph version mismatch",
  "status": 422,
  "detail": "Execution started against orderProcess@v3 but live registry has @v4.",
  "instance": "/executions/exec-12af9",
  "executionId": "exec-12af9",
  "graphId":     "orderProcess",
  "expectedHash": "sha256:1f3a…",
  "actualHash":   "sha256:9c0e…"
}
```

Built-in mappings (configured by `BlogeProblemDetailAdvice`):

Exception	HTTP status	type URI suffix
`GraphDefinitionException`	400	`/problems/graph-definition`
`GraphValidationException`	422	`/problems/graph-validation`
`GraphVersionMismatchException`	422	`/problems/graph-version-mismatch`
`SessionAccessDeniedException`	403	`/problems/session-access-denied`
`StaleLeaseException`	409	`/problems/stale-lease`
`OperatorTimeoutException`	504	`/problems/operator-timeout`
Anything else	500	`/problems/internal` (no exception detail leaked)

Disable or override the advice by declaring your own `@RestControllerAdvice` bean — the auto-configuration is gated on `@ConditionalOnMissingBean(BlogeProblemDetailAdvice.class)`.

0.3 Guided Rewrite

Start from the ticket-triage example and add production features one layer at a time.

0.3.1 1. Add observability

Add `bloge-metrics-otel` to your POM. No configuration changes needed — the starter detects `MetricsExecutionListener` on the classpath and registers it with your existing `MeterRegistry`.

Verify:

```
GET /actuator/bloge
```

You should see:

```
{
  "graphCount": 1,
  "graphs": [
    {
      "name": "springTicketTriage",
      "nodeCount": 2,
      "edgeCount": 1,
      "sourceNodes": ["classifyTicket"],
      "terminalNodes": ["draftReply"]
    }
  ]
}
```

0.3.2 2. Enable audit journaling

Add to `application.yml`:

```
spring:
  bloge:
    audit:
      enabled: true
      capture-input: true
      capture-output: false
      async-flush: true
      flush-interval: 100ms
      batch-size: 64
```

This requires a durable `DataSource` so that `AuditJournalStore` is available. The `BlogeAuditAutoConfiguration` creates an `AuditJournalListener` that the engine picks up automatically.

0.3.3 3. Enable crash recovery

```
spring:
  bloge:
    recovery:
      enabled: true
      scan-interval: 30s
      batch-size: 10
      claim-timeout: 5m
      max-recovery-attempts: 3
```

The starter creates a `RecoveryConfig` bean and wires it into `GraphEngine`. The engine manages execution leases in the background and scans for expired RUNNING executions to resume.

> *Recovery requires `bloge-durable` on the classpath and Flyway migrations at version V11 or later.*

0.3.4 4. Add a custom interceptor

```
@Bean
public OperatorInterceptor loggingInterceptor() {
    return invocation -> {
        log.info("Executing node: {}", invocation.nodeId());
        return invocation.proceed();
    };
}
```

The `graphEngine()` bean collects all `OperatorInterceptor` beans via `ObjectProvider<OperatorInterceptor>` and injects them in Spring ordering. No registration code needed.

0.3.5 5. Production deployment patterns

Three patterns recur in production BLOGE deployments. None of them require custom engine code — they are configuration and convention.

Graph versioning. The `GraphRegistryStore` (wired automatically with `bloge-durable`) persists each compiled graph by name + content hash. When you deploy a new `.bloge` file version, the starter compiles and registers it on startup. In-flight executions continue on the old graph definition; new executions use the new one. The starter exposes version-routing policies via `spring.bloge.version-routing.policy`:

- `latest` (default) — always use the newest graph definition.
- `pinned` — lock to a specific version with `pinned-version`.
- `canary` — split traffic between `canary-primary` and `canary-candidate` at the configured `canary-percentage`.

For example, to route 10 % of executions to a new graph version:

```yaml
spring:
  bloge:
    version-routing:
      policy: canary
      canary-primary: "1.0.0"
      canary-candidate: "1.1.0"
      canary-percentage: 10
```

Environment-specific operator bindings. Different environments may need different operator implementations (e.g., a sandbox payment gateway in staging vs. a real one in production). Use Spring profiles:

```
@BlogeOperator(value = "ChargePaymentOperator")
@Profile("production")
public class RealChargePaymentOperator { … }

@BlogeOperator(value = "ChargePaymentOperator")
@Profile("staging")
public class SandboxChargePaymentOperator { … }
```

The DSL references `ChargePaymentOperator` without knowing which implementation is active. The Spring profile selects the correct bean.

Blue-green deployment with durable graphs. During a rolling update, both old and new JVM instances run simultaneously. Because `GraphEngine` is stateless and execution leases are in the database, this is safe:

1. New instances start and begin handling new requests.
2. Old instances finish in-flight executions.
3. If an old instance is killed before finishing, its lease expires and a new instance recovers the execution.

The only constraint: ensure the new graph version's operator output schemas are backward-compatible with checkpoints written by the old version (see Chapter 13).

0.4 Production Readiness Scan

Boundary	Question to answer before release
Component ownership	Which bean is auto-configured, which is overridden, and who owns the override?
Operator discovery	Can the application prove every required operator is registered exactly once?
Persistence	Are execution, wait, and work-item stores durable where restart recovery is claimed?
Routing	Are tenant, version, and durable-routing policies explicit and tested?
Error contract	Do clients receive stable ProblemDetail types without internal exception leakage?
Observability	Can one execution identity join logs, metrics, traces, and business events?
Recovery	Has a cold restart been tested against the configured store and version?

Record the answer and the evidence location. A property value without an owner, test, and operational consequence is configuration trivia—not readiness.

Appendix J — BLOGE Coding and Refactoring Handbook

CHAPTER

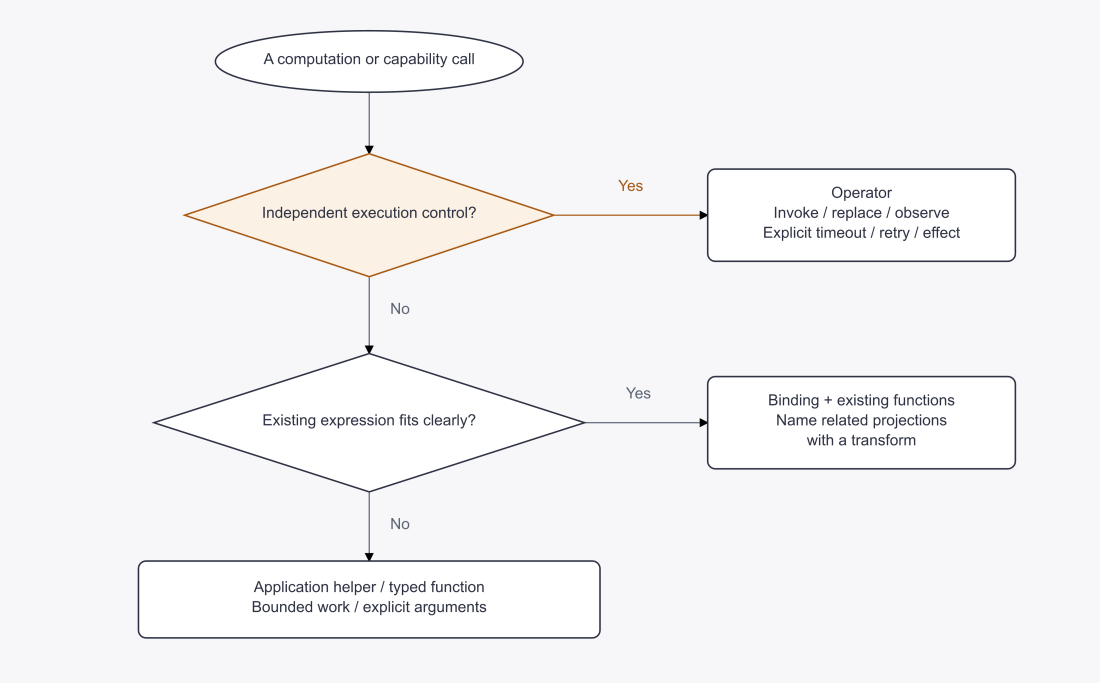
—

Appendix J — BLOGE Coding and Refactoring Handbook

Short code can hide an important decision: a binding queries a remote service, a fallback turns a timeout into a low score, or one approval call also funds and notifies. This handbook starts with those symptoms and explains which BLOGE structure fits and how to verify a behavior-preserving change.

The methods and runnable workshop use the BLOGE `0.9.9-RC1` candidate source. They are design guidance, not universal rules enforced by the engine or evidence that candidate artifacts have been published. Read the chapters in order, look up a BP identifier when making a design choice, and use Chapter 34 to combine the methods.

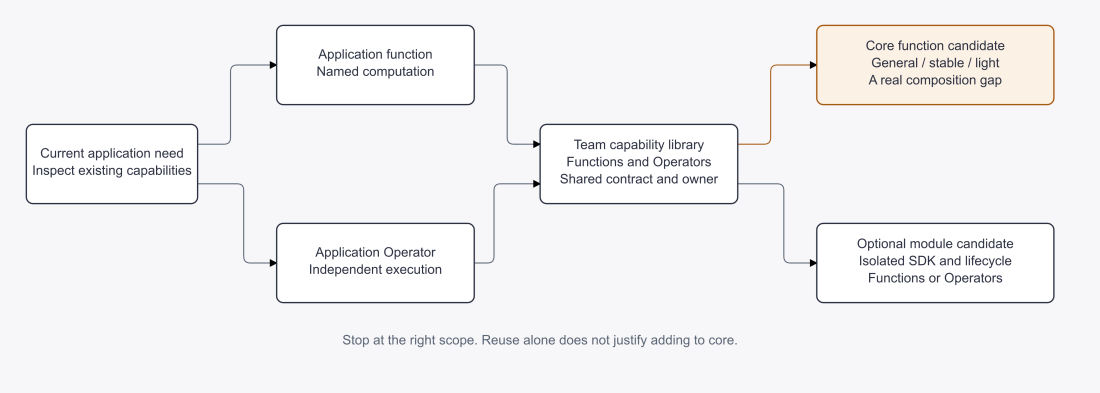
0.1 Choose the expression, then its maintenance scope



Main responsibility	Preferred expression	Boundary to preserve
Read or rename an existing field	Input binding	Explicit data source
General, lightweight value computation	Registered built-in function	Type, null, numeric, and locale semantics
Named projection of several fields	Transform	No hidden external action

Main responsibility	Preferred expression	Boundary to preserve
Bounded computation needs a domain name	Helper; typed function when DSL reuse is needed	Explicit arguments and isolated tests
Several facts determine one result	Decision table or domain method	Overlap, gaps, and priority
Select the next execution path	Branch	Consume a decision without recomputing policy
A capability needs independent execution control	Operator	Input/output, failure, cost, and effects
Reuse a sequence of business steps	Subgraph	Explicit input/output and one lifecycle owner

Pure does not mean cheap. An expensive algorithm can need an Operator boundary; a helper used only inside Java does not automatically need a published DSL function.



Maintenance scope	Suitable content	Insufficient reason to expand scope
Application	Company policy and business adapters	Another team might need it
Team or domain library	Consumers sharing one contract	Many copies with different policy owners
Optional module	Protocols, SDKs, specialized wiring	Installing a provider dependency in core for convenience
BLOGE core	Stable cross-domain primitives	Saving one expression line

0.2 BP-01

0.2.1 Check existing functions for general value computation

Symptom: A dedicated Operator trims text, removes duplicates, or concatenates fields.

Change and reason: Inspect the deployed registry/catalog, then check semantics. Bind a single field directly; name related projections with a transform. Existing `trim` and `distinct` often remove the need for an independently retryable node. Projection still has evaluation cost.

Verify: Check null, empty collections, order, and invalid types. `concat` treats null as empty text; that does not satisfy a required-name contract. General `round` does not define a currency policy.

Exception: Built-ins include environment-dependent `now()`, `today()`, and `uuid()`. Time and IDs use execution services, but are not automatically constant across retries or executions. Functions are not all pure or deterministic.

0.3 BP-02

0.3.1 Name bounded computation and keep business arguments explicit

Symptom: Nested expressions obscure evaluation intent or one calculation must be fixed in several places.

Change and reason: Extract an ordinary domain method first. Publish a typed function with `@BlogeFunction` and `@BlogeFunctionLibrary` only when graph authors need it. Describe arguments, results, and errors in JavaDoc; declare effects and determinism in metadata. Declarations do not prove behavior.

The workshop's `loanText.normalizePolicyCode` is an **application-defined function**, not a BLOGE built-in. It rejects blank codes and uses `Locale.ROOT`; that contract differs from a conversion that depends on the default locale.

Verify: The workshop provides the complete class, registration, and DSL. Test Turkish default locale, null, blank input, the catalog name, and the projected result.

Exception: Do not extract a function when one `trim` already communicates intent. Internal Operator helpers need not be published. The registry allows replacement by name, so use an explicit prefix and check collisions.

0.4 BP-03

0.4.1 Use an Operator for independent execution responsibility

Symptom: A binding calls `customer.currentCredit(id)` remotely, a function queries and writes, or expensive computation is hidden in binding evaluation.

Change and reason: Prefer an Operator when the call needs its own timeout, retry, replacement, observations, or capacity policy. A visible `CreditScore` node lets policy consume `credit.output.score`. Keep approval, funding, and notification out of one undifferentiated failure boundary.

Verify: Distinguish function invocation, Operator invocation, and real external effects. Call an `EffectPort` through `invoke` and test the actual REAL/DOUBLE/DENY wiring. Creating a node does not intercept every internal HTTP or JDBC call.

Exception: Instance function libraries support repositories, clients, and `READ_EXTERNAL`; using an Operator for I/O is default design guidance, not a syntax restriction. If a read remains a function, explain repeated evaluation, consistency, and control requirements. Conversely, expensive pure computation can justify an Operator.

0.5 BP-04

0.5.1 Prove a semantic gap before proposing a public function

Symptom: Company policy is proposed for core, or a public helper only shortens a simple composition.

Change and reason: Application registration, team modules, and default built-ins have different costs. A public proposal needs the same semantics in different business contexts, a concrete limitation in existing compositions, type/null/error/order/numeric contracts, bounded cost, an owner, and a compatibility plan. Provider SDKs and credentials usually belong in optional modules.

Example: `distinct` exists. Keeping the first item by an explicit business key can be explored locally, but `distinctBy` here is only a candidate name. Define null keys, first versus last, equality, and order before authoring new syntax.

Verify: Use distinct real consumers and regression-test existing calls. Fixing an implementation that violates a contract differs from changing the contract; null, rounding, or output-shape changes need an explicit migration.

Exception: A bank's eligibility score has a policy owner and version. Repeated use does not turn it into a general built-in; keep it in domain code or a decision table.

0.6 BP-05

0.6.1 Separate configuration, adaptation, and new Operator capabilities

Change	Preferred action	Order example
Endpoint, credentials, or wiring	Configuration or instance assembly	One payment API in different environments
Provider protocol	Narrow port/adapter, optional dependencies	Replace the provider, preserve authorization semantics
Business guarantee or result	New capability or explicit version	Model authorization and charging separately

Change	Preferred action	Order example
Several ordered capabilities	Graph/subgraph	Charge, book, notify
Field shape only	Binding/transform	Adapt the provider response

Symptom: `UniversalBusinessOperator(mode, payload)` performs queries, approvals, and charges.

Change and reason: Keep the capability name consistent with its contract. A public Operator proposal needs schemas, effects, errors, idempotency, ownership, and real-provider contracts. A connector can be released independently of core.

Verify and exception: Provider adapters can share a capability; authorization and actual charging must not be hidden behind one mode flag. Default `UNKNOWN`/`MIXED` declarations do not establish retry safety.

0.7 BP-06

0.7.1 Expose multi-fact policy and let routing consume its result

Symptom: More than three independent facts determine one result, or a policy edit requires tracing nested conditions.

Change and reason: List facts, outputs, and priority before evaluating a decision table. A branch consumes the decision. More than three is a review prompt, not an engine threshold: routing four enum values can still use a branch directly.

The loan starter rejects a blacklist hit, credit below 600, fraud at least 80, or weak income. Otherwise it approves credit at least 740 with fraud below 20 and strong income; remaining inputs require review. Chapter 34 moves this decision into a table while preserving seven output fields and three routes.

Verify: Compare fixed CSV answers, before/after results, metadata, SKIPPED states, and simulated action counts. Keeping a short, clear domain method is also a valid outcome when policy is stable.

Exception: Do not force recursive algorithms into rule rows or create a separate table for every simple guard.

0.8 BP-07

0.8.1 Explain overlap before choosing a hit policy

Contract	Policy	Required observation
Overlap is allowed; earlier rules have priority	`first`	Reordering can change the decision
Rules should be exclusive	`unique`	Executed overlapping inputs fail
Multiple matches must agree	`any`	Different outputs conflict
Every match contributes an item	`collect`	Empty, multiple, duplicate, and default items

Example: Credit 760, fraud 10, strong income, and a blacklist hit satisfy rejection and approval predicates. `first` preserves rejection priority. Replacing it with `unique` turns a valid rejection into a conflict. `unique` does not statically prove exclusion over an unlimited input domain.

Verify and exception: Exclusive rules can use unique after validating exclusions and gaps. In the current implementation, `collect + otherwise` always appends the default item. Add a normal explanation after collection when it is empty, rather than expressing that condition with otherwise.

0.9 BP-08

0.9.1 Pass business data explicitly and retain real dependencies

Symptom: Operators read hidden upstream values from shared context, independent checks are serialized, or a join assumes unselected outputs exist.

Change and reason: Bind business input explicitly and reserve context for request metadata. Remove dependencies without a business reason, then join the facts needed for a decision. Add control dependencies for required ordering without a field reference. Thread safety, CPU, and provider capacity remain separate constraints.

Verify: Inspect data sources and execution traces; test optional, missing, and SKIPPED outputs. Approval must precede a charge even if no charge field directly references approval.

0.10 BP-09

0.10.1 Give retries one primary owner and keep business keys stable

Symptom: Graph, Operator loop, and SDK each allow three total attempts, or every attempt creates a fresh UUID key.

Change and reason: Select a primary retry layer and coordinate the total budget. The same logical request keeps one key across retry and recovery; a new legitimate request gets a new identity. Execution identity is not automatically business request identity.

Verify: Count underlying attempts and committed effects separately; two layers with three total attempts can multiply to nine. In the Java builder, `.retry(1, backoff)` allows one retry after the initial attempt: two calls in total. The workshop demonstrates that an `EffectPort.idempotencyKey` declaration alone does not suppress calls. The provider or application implements deduplication.

0.9.9 boundary: Durable recovery budget is separate from call retry. It tracks consecutive failures without progress; taking a lease again does not constitute committed business progress.

0.11 BP-10

0.11.1 Preserve business facts through timeout, fallback, and compensation

Symptom: Credit timeout becomes score zero, payment timeout becomes not charged, or notification failure is described as a full rollback.

Change and reason: Distinguish success, rejection, degradation, and unknown outcomes. A timed-out write may require reconciliation by business key. A fallback needs recognizable degraded semantics. Compensation refers to the original receipt and must represent a business-allowed reversal.

Verify: Separate not executed, committed with response lost, exhausted retries, consumed fallback, failed compensation, and follow-up handling. Matching schemas do not establish equivalent business results.

Exception: Policy may reject when no trustworthy result is available, but must not fabricate a successful credit check. Compensation is not a cross-system database rollback.

0.12 BP-11

0.12.1 Make time, waiting, and recovery identity explicit

Symptom: Sleep waits for human approval, each node independently derives a deadline from now, or an in-process resume is called cold recovery.

Change and reason: Use declared suspension and deadlines for long waits, `ctx.timeSource()` for Operator time, and fixed business deadlines as facts. Test cold recovery using a new runtime and persisted state.

Verify: Cover before, exactly at, and after the deadline plus wrong correlation keys. In 0.9.9 an old worker loses local commit authority when its lease is lost; that does not establish that a remote effect did not happen. Verify checkpoints, leases, and external idempotency separately.

Exception: Bounded short polling can remain when attempts, interval, total deadline, and resource impact are explicit.

0.13 BP-12

0.13.1 Reuse multi-step workflows with one lifecycle owner

Symptom: Credit assessment is copied across graphs, subgraphs read arbitrary parent context, or Session and state machine both own the same authoritative state.

Change and reason: Extract a subgraph only around a stable business contract. Pass facts explicitly and return results without exposing internal node IDs. Choose the outer lifecycle owner by the dominant identity; communicate through public signals and terminal results.

Verify and exception: Internal node replacement must not break callers, and recovered signals must reach the correct owner. Similar flows with different policy ownership need not share an abstraction. Fewer classes or graphs do not prove clearer responsibility.

0.14 BP-13

0.14.1 Bound batch, loop, and Agent execution

Symptom: Unbounded input is immediately parallelized, a loop runs until success, or an Agent reports normal completion after exhausting turns.

Change and reason: State input size, capacity, business stop condition, and limits separately. Agents also need tool scope and budget. Use parameters supported by each model; do not invent a universal DSL throttling field.

Verify: Test empty, boundary, over-limit, partial failure, and never-terminating conditions. Bounded asynchronous recovery in 0.9.9 is opt-in; a shared budget is process-local. Permits cover actual worker lifetimes: returning from a method or requesting cancellation does not establish resource release.

0.15 BP-14

0.15.1 Keep expected answers independent and test forbidden actions

Symptom: Only SUCCESS is asserted, the implementation generates expected values, or a fully mocked Operator is claimed to validate its internal policy.

Change and reason: Test computation in domain methods, dependency and routing in graphs, provider behavior in port contracts, and business promises in Scenarios with an approved Oracle. For a rejected loan, observe the rejected result, approve SKIPPED, and zero simulated approval actions separately.

Verify: The workshop moves approval ahead of blacklist rejection and exposes the changed outcome. Equivalence preserves old behavior; it does not establish that old policy is correct. Sensitivity and bounded property claims remain limited by samples and execution controls.

0.16 Capability proposal and refactoring completion card

Material	Reviewable deliverable
Problem and existing capabilities	Current code, real consumers, catalog inspection, composition gap
Choice	Expression, execution responsibility, ownership, rejected alternatives
Contract	Types, null, errors, effects, determinism, bounded cost, version
Compatibility	Existing consumers, migration, maintenance owner
Evidence	Fixed answers, boundaries, counterexamples, routes, effects, test scope

Before submitting, explain whether the change moves data shape, policy, or execution responsibility; what observable benefit the new structure provides; which outputs and external actions must remain; and which failure injection exposes a mistake. Fewer lines or nodes alone do not establish completion.

Complete code belongs to the Chapter 34 asset directory. Commands, twelve standard examples, counterexamples, and validation limits are in the workshop.