Agentic Development with Koog

Building AI Agents on the JVM and Kotlin Multiplatform

Caelan Wexford

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press

© 2026 by N O B T R E X LLC. All rights reserved.

For permissions and other inquiries, write to:

P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language models have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

Contents

1 Introduction **1**

2 Koog Foundations and Project Bootstrapping **5**

2.1 Positioning Koog as a Production Agent Framework . . 5

2.2 Configuring Runtime and Build Prerequisites for Koog Projects 16

2.3 Choosing Stable, Beta, and Upgrade Boundaries 28

2.4 Choosing a Project Shape for JVM-Only, Mixed-Language, and Multiplatform Systems 39

3 The Core Agent Model and Provider Abstractions **53**

3.1 Instantiating and Running Agents 53

3.2 From Assistant Calls to Action-Capable Agents 65

3.3 Binding Providers and Models with Prompt Executors . 76

3.4 Designing for Model Switching, Rerouting, and Resilience 87

3.5 Operating Streaming Runs Under Coroutine Concurrency 99

4 Tools, External Actions, and Service Boundaries 111

4.1 Designing Custom Tools as Stable Agent Interfaces . . 111

4.2 Connecting Agents to APIs and Application Services . . 124

4.3 Parallel Tool Calls and Execution Coordination 137

4.4 Protocol-Based Interoperability with MCP and ACP . . 148

4.5 Java and Kotlin Interop at the Tool Layer 160

5 Type-Safe Graph Workflows for Predictable Agents 173

5.1 Why Graph Workflows Outperform Linear Prompt Chains 173

5.2 Modeling Typed Nodes, Edges, and Workflow State . . 185

5.3 Implementing Branching, Retries, and Compensating Paths 197

5.4 Orchestrating Tools, Validation, and Human Gates Inside Workflow Graphs 211

5.5 Choosing Between Explicit Workflows and Autonomous Planning 225

6 Planning, Memory, and Long-Running Context 239

6.1 Choosing Planner-Driven Execution with GOAP 239

6.2 Designing Short-Term Memory, Long-Term Memory, and Retrieval Boundaries 253

6.3 Building Embeddings and RAG Pipelines for Grounded Agent Behavior 265

6.4 Compressing History for Token-Efficient Continuity . . 276

6.5 Implementing Checkpointing and State Restoration . . 288

7 Observability, Reliability, and Production Operations 301

7.1 Tracing as a First-Class Development Practice 302

7.2 Instrumenting Koog with OpenTelemetry and External Observability Backends 312

7.3 Classifying Failure Modes Across Providers, Tools, and Workflows 324

7.4 Designing Retries, Recovery Boundaries, and Durable Resumption 336

7.5 Engineering Latency, Cost, and Throughput Under Production SLOs 347

8 Embedding Koog into JVM and Kotlin Multiplatform Systems 359

8.1 Integrating Koog with Spring Boot and Ktor 360

8.2 Wiring Memory, Tools, and Persistence Through Application Boundaries 372

8.3 Designing Shared Agent Logic for Kotlin Multiplatform 384

8.4 Choosing Transport Boundaries and Deployment Topologies 395

8.5 Assembling a Production Reference Architecture for Agentic Applications 407

Chapter 1

Introduction

Artificial intelligence agents are becoming a practical software concern for teams building production systems, not only an experimental topic for prototypes and demonstrations. As organizations move from isolated prompt-driven features toward applications that must reason, call services, manage state, recover from failure, and operate under clear reliability expectations, the engineering demands become substantially more rigorous. This book addresses that shift through the lens of Koog, a framework for building agentic systems on the JVM and across Kotlin Multiplatform targets.

Koog is significant because it brings agent development into a software environment already valued for strong typing, mature tooling, structured concurrency, and long-term maintainability. For Kotlin and Java teams, this creates an opportunity to treat agents as first-class application components rather than peripheral integrations. The central premise of this book is that agentic behavior should be designed with the same discipline applied to other production-critical systems: explicit contracts, inspectable execution paths, clear service boundaries,

durable state, and operational visibility.

The chapters that follow are organized to build that perspective methodically. The opening chapter establishes the foundations: what Koog is, how its runtime and build assumptions affect project setup, and how stability and versioning decisions influence architecture. It also helps readers choose an appropriate project shape, whether that means a JVM-only service, a shared multiplatform core, or a mixed Kotlin and Java deployment model.

From there, the book explains the core execution model behind Koog agents. It examines how agents are instantiated, how model providers are abstracted, and how prompt executors mediate between application logic and language model capabilities. It also considers model switching, rerouting, streaming, and concurrency, all of which are essential when systems must remain resilient under real operating conditions rather than idealized examples.

A central theme of the book is that useful agents must act, not merely generate text. For that reason, substantial attention is given to tools, external actions, and service boundaries. Readers will see how to expose capabilities as stable interfaces, connect agents to APIs and internal services, coordinate parallel actions, and incorporate protocol-based interoperability where standardization is beneficial. Because many teams operate in mixed-language environments, Java and Kotlin interoperability is treated as an architectural concern rather than an afterthought.

Koog's distinctive strength lies in its typed workflow model, and this book gives that topic sustained focus. Workflow graphs provide explicit control over branching, retries, compensating actions, and state transitions. They allow complex agent behavior to be expressed in forms that are testable, inspectable, and accountable. The book contrasts these explicit workflows with more autonomous planning strate-

gies so readers can make sound design decisions based on predictability, compliance, and operational risk.

The later chapters extend this foundation into memory, retrieval, checkpointing, observability, reliability, and deployment integration. These concerns determine whether an agent remains coherent over time, whether it can recover from interruptions, and whether teams can understand and improve its behavior in production. The final material synthesizes these elements into reference patterns for embedding Koog into backend and multiplatform systems.

This book is written for engineers, architects, and technical leads who need more than introductory examples. Its purpose is to present agentic development with Koog as a software engineering discipline: one that combines language model capabilities with typed design, controlled execution, and production-grade operational practice.

Chapter 2

Koog Foundations and Project Bootstrapping

Chapter 1 begins with a practical tension familiar to experienced teams: agent experiments are easy to start, but hard to evolve safely once they become part of a real service. The chapter shows how Koog changes that equation by giving Kotlin and JVM teams a structured foundation, then walks the reader through the version, build, and project-shape choices that quietly determine whether an agent system remains portable, debuggable, and maintainable six months later.

2.1. Positioning Koog as a Production Agent Framework

A direct model call is enough to produce a convincing demo. You send a prompt, receive text, maybe parse a little JSON, and wire the result into an HTTP handler. The design usually holds until the first real inte-

gration. Then the questions arrive all at once: where do you keep conversation state, how do you constrain external side effects, how do you retry tool failures without duplicating actions, how do you inspect a bad run after the fact, and which parts of the system belong to the model interaction layer rather than to the rest of the application. A production agent framework exists to answer those questions with structure instead of ad hoc glue.

Koog should be understood in that operational sense. It is not merely a convenience wrapper around provider APIs, and it is not just a prompt templating utility. It is a framework for building agent subsystems that can hold state, invoke tools, participate in controlled workflows, persist progress, and emit operational telemetry in forms that matter to software teams running JVM and Kotlin systems. That framing matters because it determines how you place Koog inside an architecture. You do not treat it as a string-processing helper sitting at the edge of a controller. You treat it as a governed execution layer with explicit contracts and lifecycle boundaries.

A minimal setup already hints at that larger role. Even a small agent definition starts from a concrete model binding and a framework-owned agent object rather than from a raw HTTP request to a model provider.

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.prompt.executor.clients.openai.OpenAILLMClient
import ai.koog.prompt.executor.clients.openai.models.OpenAIModels

val client = OpenAILLMClient(
 apiKey = System.getenv("OPENAI_API_KEY")
)

val model = OpenAIModels.Chat.GPT4o

val agent = AIAgent(
 promptExecutor = client,
 model = model
)
```

That code is intentionally small, but it already expresses two design decisions that separate a framework from a raw API client. First, the provider connection is a bound execution dependency, not an incidental helper hidden somewhere in business logic. Second, the agent is a first-class runtime object. You can build more behavior around it: tools, memory, workflow constraints, persistence, tracing, and integration with transport layers such as HTTP services or background workers. The point is not that every agent starts complex. The point is that the framework gives complexity a place to live when the system grows beyond a single prompt-response exchange.

From prompt call to governed execution

Teams often begin with the wrong abstraction boundary. They write code that assumes the model interaction itself is the unit of work: one prompt in, one answer out. That assumption collapses under ordinary production requirements. A useful agent run often includes multiple steps: gather context, call one or more external tools, reason over tool output, maybe compress or select history, produce a structured result, and record enough telemetry to explain why the run succeeded or failed. Once you have that shape, a prompt is only one ingredient in a larger execution loop.

Koog is valuable because it treats that loop as something worth structuring. The framework gives names to components that would otherwise be hidden inside callback chains and utility classes. Those names become the vocabulary for design and debugging. If an operator asks why an agent produced a wrong answer, you need more than “the prompt was bad.” You need to know which model binding ran, which tools were available, what state the agent carried, whether persistence resumed a prior checkpoint, what workflow path was taken, and what external calls were made.

Agent

An *agent* is the framework-managed execution unit that coordinates model interaction with system behavior. It is not just a chat persona and not just a prompt template. In operational terms, the agent is the thing you instantiate, configure, invoke, observe, and evolve. It owns the rules under which model calls happen and the surrounding mechanics required to make those calls safe and useful inside software.

That definition is intentionally stricter than the casual use of the word in many product demos. A chatbot that forwards every user message directly to a model may feel agent-like from the outside, but it lacks most of what matters for production operation. It has no explicit tool boundary, no durable state model, no reproducible execution path, and usually no meaningful telemetry beyond provider latency. Koog positions the agent as a software component with responsibilities and extension points, not as a conversational illusion.

Model binding and prompt executor

A *model binding* is the concrete pairing between your agent and the underlying model capability it uses. In the example above, the binding is expressed through an `OpenAILLMClient` and a specific model constant. This is more than a convenience parameter. It determines the provider transport, request semantics, model capabilities, and often operational constraints such as latency, token budgets, and supported response modes.

The *prompt executor* is the framework-facing component that actually performs model execution. You can think of it as the boundary where Koog stops being purely internal orchestration and starts depending on an external model service. This distinction matters because provider portability is never free. If your prompts, tool schemas, error handling, and response expectations are all tangled directly with one vendor SDK, you do not really have an abstract agent layer; you have provider-specific application code with a thin facade. Koog helps by

giving you a framework-level place to bind model execution, but abstraction discipline remains your job. You still need to keep provider quirks from leaking indiscriminately into the rest of the application.

Tools

A *tool* is an explicit contract for side effects or external data access. This is one of the clearest boundaries between toy assistants and production agent systems. Without tools, developers often let prompts imply side effects: "if the user confirms, create the ticket" or "look up the invoice and summarize it." That approach hides critical behavior inside natural language and makes the system difficult to audit, secure, and test.

A proper tool boundary does the opposite. It turns side effects into named operations with typed inputs, controlled execution, and visible outcomes. If an agent needs to query inventory, create a support case, send a message, or hit an internal API, the operation should exist as a tool contract. The model may decide *when* to request the tool, but your application decides *what* the tool is allowed to do, how failures are handled, what retries are safe, and which permissions apply. Later material will dig into tool design and external actions in detail; the key point here is that Koog gives tools a first-class place in the agent model instead of forcing you to smuggle operations through prompt text.

Workflow

A *workflow* is the structured control over how an agent run proceeds. In simple cases, the workflow may be close to a single model turn. In more capable systems, it becomes the mechanism that prevents agent behavior from degenerating into uncontrolled recursion or accidental branching. Workflows matter when you need repeatability, bounded execution, approval points, staged retrieval, or deterministic post-processing around model-driven decisions.

This distinction often gets blurred because many frameworks call any

multi-step interaction a workflow. The useful interpretation is narrower: a workflow is not merely "many prompts in a row." It is a defined execution structure with constraints and transitions that your software can inspect and reason about. Koog's support for workflows is part of why it belongs in production conversations. It gives you a path to express graph-like or staged agent behavior without reducing the entire system to unstructured conversational state.

Memory

Memory is the managed handling of prior context that the agent may use across steps or over longer interaction horizons. In a demo, memory often means appending all previous messages to the next prompt. That approach fails quickly. Context windows are finite, irrelevant history adds cost and noise, and different classes of state should not be treated identically. A customer identifier, a retrieved policy document, and five turns of transient conversational clarification are not the same kind of memory.

In a production framing, memory is selective, scoped, and policy-driven. Some context should stay ephemeral within a run. Some should be compressed or summarized. Some should be retrieved on demand from external storage. Some should never be exposed back to the model at all. Koog's role is to make memory a recognizable subsystem within agent execution rather than an ever-growing message list embedded in controller code.

Persistence and checkpointing

Persistence and *checkpointing* are the mechanisms that let an agent survive beyond a single in-memory execution. Persistence records enough state to support continuity, auditing, replay, or coordination with other system components. Checkpointing is the more specific practice of capturing progress at meaningful boundaries so that a run can resume or be inspected after interruption or failure.

These concerns emerge earlier than many teams expect. A long-running task may span multiple tool calls and user interactions. A server restart may interrupt a run after a non-idempotent external action. A support engineer may need to inspect what the agent knew before it issued a wrong recommendation. If your only record is the final assistant message, you have no operational footing. Koog's support for persistence-oriented capabilities reflects the reality that production agents are not just real-time text generators; they are stateful processes embedded in software systems with uptime, audit, and recovery requirements.

Tracing and observability

Tracing is the runtime record of what the agent did and when it did it. *Observability* is the broader ability to inspect, measure, and diagnose agent behavior in operation. This is one of the strongest signals that a framework is designed for production rather than for notebook experimentation. You need to know more than total request latency. You need the shape of a run: model calls, tool invocations, decision boundaries, retries, timing, payload sizes, and failure causes.

That data serves multiple audiences. Operators use it to debug incidents and performance regressions. Developers use it to refine prompts, tool schemas, and workflow design. Security and compliance teams may need to inspect external actions and data flows. Product teams may need aggregate signals about completion rates and failure patterns. Koog's observability-oriented features are important not because telemetry is fashionable, but because opaque agent systems are unmaintainable once they interact with real business processes.

Transport boundary

The *transport boundary* is where external requests enter your system and where agent results leave it: HTTP controllers, message consumers, scheduled jobs, CLI entry points, mobile-facing APIs, or other

integration surfaces. This boundary should remain conceptually separate from the agent runtime. A transport adapter should translate between external protocol concerns and internal agent invocations. It should not absorb the agent's orchestration logic.

This separation is easy to violate. A web controller starts assembling prompts, performing retrieval, deciding which external services to call, and patching together error handling. At that point the transport layer has effectively become the agent runtime, and the framework cannot provide coherent execution semantics. Koog works best when you let it own the agent concerns while your surrounding application owns delivery, authorization, persistence backends, domain services, and presentation formats.

What Koog is not responsible for

Drawing the framework boundary clearly prevents a different class of mistakes: asking Koog to own responsibilities that belong to the application around it. Koog does not replace your domain model, your authorization layer, your database schema, your secret management, your business invariants, or your operational governance. It can orchestrate tool use, but it does not decide your permission model. It can help structure memory and persistence, but it does not define retention policy for regulated data. It can emit traces, but it does not eliminate the need for log pipelines, metrics backends, alerting, or incident response.

That boundary discipline matters even more on the JVM, where mature application frameworks already provide strong answers for web transport, dependency injection, transaction management, scheduling, and service integration. Koog should sit alongside those systems, not dissolve them into agent-centric abstractions. If you force the framework to absorb every concern, you lose the advantages of the existing ecosystem and make the agent layer harder to reason about.

Why the JVM context matters

Koog's positioning is especially relevant in JVM environments because the surrounding ecosystem favors long-lived, integrated, operationally managed services. The JVM is where many teams already run transaction systems, internal APIs, scheduled workloads, and event-driven services. That means an agent framework cannot assume it is operating in isolation. It has to coexist with existing application infrastructure, deployment standards, security controls, and interoperability requirements.

Kotlin strengthens that fit. Its language features make it practical to express agent-oriented configuration and orchestration with a high signal-to-noise ratio. You get concise DSL-friendly code, strong typing, and concurrency tools that help structure asynchronous work without turning the codebase into callback soup. At the same time, the framework remains relevant to mixed Kotlin and Java environments because many organizations will not rebuild their service estate around a pure Kotlin stack. Production adoption depends on fitting into the code people already have, not into a greenfield ideal.

Kotlin Multiplatform as capability, not ideology

Koog's multiplatform story is useful, but you should interpret it carefully. Kotlin Multiplatform creates a path to share selected agent logic across JVM, JavaScript, WebAssembly-oriented targets, Android, and iOS. That is a meaningful capability when you have genuine cross-platform logic to share: prompt construction rules, workflow policy, lightweight decision code, or domain-specific agent behavior that should remain consistent across clients and services.

It is not an automatic argument for making every agent module multiplatform. Real systems have platform-specific edges: provider credentials, network stacks, persistence backends, local storage, background execution models, and integration with server-only frameworks such

as Spring Boot or Ktor-based infrastructure. The practical question is not whether multiplatform is elegant in principle. The practical question is which parts of the agent system gain enough reuse value to justify shared code without forcing lowest-common-denominator abstractions onto the rest of the design. Koog is valuable here because it does not confine you to a single deployment shape, but the architecture still needs explicit boundaries.

A production framework differs from an assistant wrapper

The most common conceptual error is to treat all LLM abstractions as equivalent. They are not. A thin assistant wrapper usually improves ergonomics around model calls: provider credentials, request formatting, maybe chat history, perhaps some structured output support. That can be useful, but it does not by itself solve production agent problems.

A production agent framework adds at least four things. It provides structured execution rather than a sequence of hidden helper calls. It exposes extension points for tools, workflows, memory, persistence, and observability rather than forcing every concern into prompt text. It treats state and lifecycle as first-class concerns instead of incidental side effects. And it acknowledges that the system will be operated, debugged, resumed, and evolved over time. Koog belongs in that second category. If you approach it as a nicer prompt wrapper, you will underuse its strengths and misplace important responsibilities in application code.

Anti-patterns that signal the wrong mental model

Several anti-patterns appear repeatedly when teams have not yet made this conceptual shift.

One is *implicit orchestration through prompts*. The prompt contains instructions such as “first look up the user profile, then check account

status, then if needed create a support case." The model is expected to simulate a workflow without explicit tool and control boundaries. This makes behavior hard to test and nearly impossible to audit.

Another is *uncontracted side effects*. The agent is allowed to trigger external actions through arbitrary callback code with no clear tool interface, no idempotency policy, and no permission boundary. Failures in such systems are hard to classify: did the model choose badly, did the callback misbehave, or did the external system partially apply the action.

A third is *false provider portability*. Teams claim they can swap model vendors because all calls pass through a helper function, but their prompts, parsing assumptions, and response handling are tightly coupled to one provider's behavior. A real framework boundary reduces coupling; it does not eliminate the need to manage it.

A fourth is *conversational convenience mistaken for production readiness*. A system that sounds polished during interactive testing may still have no replay story, no tracing, no checkpointing, and no bounded workflow behavior. Smooth dialogue is not the same as operational maturity.

A practical way to place Koog in your architecture

If you are designing a service around Koog, place it as an internal application subsystem with clear inbound and outbound boundaries. Inbound requests arrive through transport adapters: HTTP endpoints, events, jobs, or UI-facing APIs. The adapters translate those requests into agent invocations. Inside the Koog-managed area, you define the agent, bind the model, expose tools, choose memory and persistence strategies, and configure tracing. Outbound integration happens through tool contracts and persistence backends, not through arbitrary helper methods sprinkled across controllers.

That placement yields several benefits. You can test agent behavior separately from web transport. You can replay or inspect runs using persisted state and traces. You can evolve model bindings without rewriting delivery code. You can make tool permissions and failure handling explicit. Most importantly, you can discuss the system with a shared vocabulary. Instead of saying "the prompt code in the controller probably did something strange," you can say "the workflow retried after a tool timeout, resumed from a checkpoint, and produced the final answer with stale memory." That level of precision is what turns agent development from demo craft into software engineering.

Koog fits teams that need that precision. It gives you a framework boundary around the difficult parts of agent execution while leaving domain ownership, application integration, and operational policy where they belong: in the surrounding system you control.

2.2. Configuring Runtime and Build Prerequisites for Koog Projects

A Koog project usually fails long before the agent logic becomes interesting. The common failure mode is deceptively mild: a sample compiles, one provider call succeeds, and the team assumes the foundation is sound. Weeks later, the real requirements arrive—tool execution, a web integration, CI builds, dependency updates, or a multiplatform split—and the project starts fighting its own baseline. The friction rarely comes from one dramatic mistake. It comes from a stack of small, early choices: an outdated JDK image, an unpinned Kotlin plugin, a build that quietly allows transitive drift, or a project layout that makes Java interop awkward once the public API starts to harden.

You avoid that trap by treating runtime and build prerequisites as architectural constraints. For Koog, the baseline is concrete enough to

pin immediately: JDK 17 or newer, Kotlin 2.2.0 or newer, and either Gradle 8.0 or newer or Maven 3.8 or newer. Those versions are not arbitrary housekeeping. They define the floor under the framework, shape what language features and build plugins you can rely on, and determine how safely you can grow from a trivial agent into a system with tools, persistence, and observability.

A minimal Gradle baseline is small enough to establish early and strict enough to prevent later drift. Start with the stable module unless you have a specific reason to absorb beta churn.

```
plugins {
 kotlin("jvm") version "2.2.0"
}

repositories {
 mavenCentral()
}

dependencies {
 implementation("ai.koog:koog-agents:1.0.0")
}

kotlin {
 jvmToolchain(17)
}
```

If you prefer the Kotlin DSL in `build.gradle.kts`, this is a clean first anchor: one Koog dependency, one JVM target, one explicit toolchain. It does not preload convenience modules, starters, or experimental additions. That restraint matters. You want the first executable baseline to prove that the language, runtime, repository access, and dependency resolution are correct before you add higher-level framework pieces.

A minimal smoke test should compile against the actual Koog types you plan to use, not just download artifacts. Use environment variables for provider secrets from the start so that your local run, CI, and container runtime all converge on the same secret-handling pattern.

```
import ai.koog.agents.core.agent.AIAgent
```

```
import ai.koog.prompt.executor.clients.openai.OpenAILLMClient
import ai.koog.prompt.executor.clients.openai.models.OpenAIModels

fun main() {
 val apiKey = System.getenv("OPENAI_API_KEY")
 ?: error("OPENAI_API_KEY is not set")

 val client = OpenAILLMClient(apiKey = apiKey)
 val agent = AIAgent(
 promptExecutor = client,
 model = OpenAIModels.Chat.GPT4o
 )

 println(agent::class.qualifiedName)
}
```

A successful run proves less than many teams think, but it proves the right things early: the build resolves Koog, the Kotlin compiler can process your source shape, the runtime can load the framework classes, and your secret path is externalized rather than hardcoded.

`ai.koog.agents.core.agent.AIAgent`

Pin the compatibility floor before you add architecture

The first discipline is to treat version selection as a tested tuple rather than as independent moving parts. JDK, Kotlin, Gradle, Koog, and any surrounding framework integrations all interact. If one member of that set floats without review, you inherit failure modes that are hard to diagnose because the project still *mostly* works. That is the dangerous state: builds pass until one plugin upgrade changes bytecode targets, one CI runner picks a different Java image, or one developer silently compiles with a newer toolchain than production uses.

JDK 17 is the practical baseline because it is modern enough for current Kotlin and library ecosystems while still being widely supported across CI systems, container bases, and enterprise deployment environments. If your organization is already standardized on a newer long-term-support JDK, that can work well, but do not treat “newer” as

automatically safer. The build must declare the toolchain you expect, and CI must enforce it. A project that compiles on a developer laptop with JDK 21 but deploys from a JDK 17 builder is inviting subtle incompatibilities in generated bytecode, plugin behavior, and reflective libraries.

Kotlin 2.2.0 matters for more than syntax. It establishes the compiler and plugin generation the Koog ecosystem is documented against. In practice, Kotlin version mismatches show up as plugin resolution issues, Gradle model breakage, or source-level incompatibilities that surface only when you introduce multiplatform structure or framework integrations. If you let Kotlin drift independently of Koog and Gradle, you make the project baseline harder to reproduce.

Gradle 8.0 or newer is best understood as the minimum viable build platform for a serious Koog codebase. You want modern dependency resolution, toolchain support, version catalog features if you adopt them, and plugin behavior aligned with current Kotlin builds. Maven 3.8 or newer remains viable for JVM-centric teams, especially when Koog is embedded into an existing Java build estate, but the project shape and examples in most Kotlin-heavy environments are more naturally expressed with Gradle.

Start with the stable module, not the maximal module set

Koog exposes both stable and beta surfaces. Early in a project, the right question is not “what can I include” but “what is the smallest dependency set that gives me a runnable foundation.” The stable Gradle dependency is:

```
implementation("ai.koog:koog-agents:1.0.0")
```

The beta umbrella dependency is:

```
implementation("ai.koog:koog-agents-additions:1.0.0-beta")
```

That distinction should influence your bootstrap strategy immediately.

Use the stable module for the initial project skeleton unless a required capability exists only in beta and you are willing to isolate that risk. Pulling in beta additions at the start usually feels efficient because it postpones dependency edits, but it weakens the value of your first baseline. If the project later breaks, you do not know whether the fault lies in your environment, your architecture, or your choice to absorb less stable surfaces too early.

The same rule applies to Maven-based builds. If your organization is JVM-first and already invested in Maven, keep the Koog dependency surface just as narrow.

```
<dependencies>
 <dependency>
 <groupId>ai.koog</groupId>
 <artifactId>koog-agents-jvm</artifactId>
 <version>1.0.0</version>
 </dependency>
</dependencies>
```

If you truly need beta additions on Maven, the artifact is separate and explicitly labeled as such.

```
<dependency>
 <groupId>ai.koog</groupId>
 <artifactId>koog-agents-additions-jvm</artifactId>
 <version>1.0.0-beta</version>
</dependency>
```

That explicitness is healthy. It forces you to document where you have accepted churn risk rather than allowing beta surfaces to blend invisibly into your baseline.

Choose the build shape to match the surrounding system

A Koog project rarely lives alone. It usually enters one of three environments: a Kotlin-first service, a mixed Kotlin/Java JVM codebase, or a broader multiplatform project. The build setup should reflect that reality on day one because retrofitting the shape later is expensive.

For a Kotlin-first JVM service, use the simplest possible setup: Kotlin JVM plugin, explicit JVM toolchain, stable Koog module, and one provider dependency path at runtime through environment variables. Keep the initial source set narrow. Put the first agent code in a dedicated package rather than mixing it into transport handlers or framework configuration classes. That separation makes it easier to grow into tools and observability without moving files across layers later.

For mixed Kotlin/Java systems, decide early whether Kotlin is merely hosting Koog internally or whether it will define public APIs consumed by Java code. That decision affects naming, nullability exposure, coroutine boundaries, and the amount of Kotlin-specific DSL style you can safely embed into the surface area. Even if Java is not dominant, your build should reflect reality. A project that claims to be mixed-language but hides all interop decisions until after core agent APIs are written usually ends up with awkward wrappers or accidental Kotlin leakage into Java-heavy modules.

For Kotlin Multiplatform, the build becomes a design statement. The top-level `kotlin {}` block is no longer boilerplate; it is the primary place where you declare what code is shareable and what remains platform-specific. That power is useful, but it raises the cost of casual decisions. Multiplatform is not a decorative upgrade over a JVM-only build. It changes plugin interactions, source-set boundaries, target declarations, and what kinds of integration libraries you can consume directly in shared code.

Do not defer toolchain enforcement to CI alone

Many teams assume that adding a CI workflow later is enough to standardize the toolchain. It is not. The build file should declare the intended Java toolchain, and the repository should make version intent visible in source control. CI then becomes the enforcement mechanism, not the first place where the requirement exists.

A disciplined local and CI flow often starts with a plain compile and test pass.

```
./gradlew clean test
```

If you need a faster smoke cycle while bootstrapping, compile first and test next, but do not stop at dependency resolution. You want a real compile boundary, because that is where Kotlin plugin mismatches and bytecode target problems appear.

```
./gradlew clean compileKotlin
./gradlew test
```

For Maven, the equivalent baseline is just as straightforward.

```
mvn clean test
```

The valuable habit is not the command itself. It is the rule that every initial build decision must survive the same path locally and in CI. If developers use one Java distribution, CI uses another, and containers use a third, you no longer have a baseline; you have three environments that happen to compile similar code.

Repository and dependency policy are part of the runtime baseline

Build correctness is not only about versions. It is also about where dependencies come from and how they are allowed to change. For early Koog work, `mavenCentral()` is usually enough. Resist the urge to add extra repositories preemptively. Each additional repository increases ambiguity in resolution and makes supply-chain review harder. If your organization uses an internal proxy or artifact mirror, configure that deliberately and document it as part of the project baseline.

Transitive drift is another common sabotage point. A sample project with a single Koog dependency may appear stable, but as soon as you add testing libraries, web frameworks, and provider-adjacent integra-

tions, transitive versions can move under your feet. You do not need a heavy governance apparatus on day one, but you do need one of these disciplines early:

- Use a Gradle version catalog and keep Koog, Kotlin, and major framework versions declared centrally.
- Enable dependency locking if your team already uses it.
- Record a tested dependency tuple in repository documentation and review changes to it explicitly.

Any of those is better than letting upgrades happen as side effects of unrelated changes.

Secret handling belongs outside the build

Provider credentials are the first operational secret most Koog projects encounter. Handle them correctly from the start. Use environment variables for local development, CI, and runtime deployment. Do not commit secrets into source files, Gradle properties that are shared casually, or example configuration checked into the repository.

A shell export is enough for local bootstrap:

```
export OPENAI_API_KEY="your-api-key"
./gradlew run
```

For one-off execution, you can scope the secret to a single command.

```
OPENAI_API_KEY="your-api-key" ./gradlew run
```

This approach scales naturally into CI secret stores and container orchestration platforms. More importantly, it keeps the source tree honest. Example code that compiles only because a developer pasted a secret into a local config file teaches the wrong operational habit.

Use a smoke agent to validate the foundation, not the design

The first executable Koog program should validate the environment, not express your final architecture. Keep it intentionally boring. Bind one documented client, construct one `AIAgent`, and prove that the runtime can load and execute basic code. Avoid introducing tools, transport layers, persistence backends, or starter frameworks during that first pass. Each additional integration multiplies the number of ways the baseline can fail.

If you need multiple provider bindings later, Koog exposes `MultiLLMPromptExecutor`. That is a legitimate core identifier, but do not add it merely because you expect multi-provider support eventually. A smoke test should verify the smallest viable path. Architectural ambition can wait until the baseline is repeatable.

Mixed Kotlin and Java builds need an API policy early

In mixed-language systems, technical debt often begins as stylistic freedom. Kotlin authors write a concise API that feels natural inside Kotlin and assume Java callers can adapt later. That assumption is usually false once the API begins to spread. If Koog-backed services are going to be consumed from Java, decide early whether your Kotlin code is internal implementation or a published interop boundary.

If the boundary is public, avoid exposing coroutine-heavy or DSL-centric entry points as the only usable surface. Provide JVM-friendly facades where needed. Keep nullability explicit. Separate framework configuration from business-facing service methods. These are not generic interop niceties; they affect how safely Koog can inhabit an existing JVM organization. A framework that is operationally strong but socially impossible to call from the rest of the codebase will be sidelined into isolated modules and never integrated deeply enough to matter.

Multiplatform baselines require stricter restraint

Kotlin Multiplatform broadens what you can do with Koog, but it also

narrows your tolerance for sloppy setup. When you choose multiplatform, you are choosing source-set discipline, target declaration discipline, and stricter control over where platform-specific integrations live. Only one Android target may exist per Gradle subproject, and Java ecosystem plugins often fit poorly when pushed directly into a multiplatform root. That is why many robust codebases split responsibilities across subprojects: a shared core for truly portable logic, and platform-specific modules for provider integration, server frameworks, or persistence stacks.

That layout is not ceremony. It prevents the common architectural mistake of forcing server-only concerns into shared modules until the shared code becomes unusable anywhere else. If your real need is a JVM service with future optional reuse, start on the JVM and extract only proven shared logic later. Multiplatform rewards clear boundaries; it punishes speculative sharing.

Validate the build in the same shape you plan to ship

A local successful run is not enough if the deployment environment differs materially. If you expect to ship in a container, build and test in that container base early. If your CI agents use a pinned JDK image, make local toolchain resolution match it. If your organization requires offline or mirrored artifact resolution, validate that before the codebase accumulates dependencies.

You do not need a full delivery pipeline at the bootstrap stage, but you do need a shipping-shaped compile path. Many build issues that appear “later” were present from the beginning and merely hidden by friendly local environments. Koog’s later capabilities—observability, persistence, framework integrations—increase the number of moving parts. That makes baseline fidelity more important, not less.

A recommended bootstrap sequence

A repeatable Koog foundation usually follows a simple order.

- Choose the target shape: JVM-only, mixed JVM, or multiplatform.
- Pin JDK, Kotlin, and Gradle or Maven at the documented floor or your tested higher baseline.
- Add only the stable Koog core dependency first.
- Externalize provider credentials through environment variables.
- Compile a minimal smoke agent against real Koog types.
- Run tests in CI with the same toolchain declaration used locally.
- Introduce version catalogs, locking, or another dependency policy before adding broader integrations.
- Add beta modules only behind explicit boundaries when the stable baseline is already proven.

That sequence is intentionally conservative. It front-loads reproducibility and defers convenience. In practice, it saves time because the first broken build is easier to fix than the tenth.

Anti-patterns that quietly poison the baseline

A few early shortcuts create disproportionate pain later.

One is *preloading every likely dependency*. Teams add stable and beta Koog modules, a web framework starter, test helpers, and provider-specific extras before the first compile. When the build fails, the error surface is too wide. You lose the ability to reason about which layer is actually broken.

Another is *treating sample repository settings as production defaults*. Example builds often optimize for brevity, not for long-term governance. A project that will live in CI, containers, and shared development environments needs explicit toolchain and repository policy, not merely enough configuration to demonstrate a single local run.

A third is *postponing interop decisions*. If Java callers matter, design for them now. If multiplatform matters, separate shared and platform-specific code now. Retrofitting those boundaries after agent APIs have spread through the codebase is expensive and politically difficult.

A fourth is *floating version drift*. Leaving plugin and dependency updates to chance creates a false sense of progress. The build continues to work until an unrelated change pulls a different version into the graph, and then the failure appears far from the real cause.

Practical decision signals

When you need to choose between competing bootstrap options, use operational criteria rather than aesthetic ones.

Choose the newest supported toolchain only if your CI runners, base images, and organizational deployment standards can adopt it consistently. A slightly older but fully reproducible baseline is often safer than a newer one that only works on selected developer machines.

Choose Kotlin-first build ergonomics when the Koog layer is central and Kotlin will remain the main authoring language. Choose a more explicit mixed-language posture when Java callers or legacy modules are significant. Convenience for the agent author is not the only metric; long-term maintainability of the whole service matters more.

Choose the minimum dependency set that yields one executable agent. Every extra module you add before that point reduces diagnostic clarity. Extra convenience now often becomes upgrade surface later.

The build baseline should make later work easier: adding tools without dependency chaos, enabling persistence without reshaping the repository, introducing tracing without rebuilding the runtime foundation, and integrating with surrounding frameworks without discovering that the original project shape was too casual to carry real load. A Koog project earns its flexibility when the first compile, the first test run, and the first CI pass all agree on the same environment.

2.3. Choosing Stable, Beta, and Upgrade Boundaries

The first version number you pick for Koog becomes an architectural policy whether you acknowledge it or not. A team may think it is only adding a dependency, but that dependency determines who absorbs change, how quickly new capabilities can enter the system, and which parts of the codebase are allowed to rely on evolving framework behavior. The cost of getting this wrong usually does not appear at bootstrap time. It appears when the first production workflow depends on a feature that later shifts under a minor upgrade, or when a beta integration leaks into application APIs and turns an otherwise routine dependency refresh into a multi-module migration.

That is why release maturity belongs in design discussions, not only in dependency management tickets. Koog gives you a useful starting point because it separates stable modules from beta additions and documents different expectations for each. Stable modules follow semantic versioning. Public APIs are intended to remain compatible across minor and bugfix releases, and deprecations introduced in a minor release remain available until at least the next major release. Beta modules are explicitly marked with a `-beta` suffix and may change across minor releases. Those two facts should shape your boundaries immediately.

A small build file is enough to make the distinction visible. The stable umbrella module and the beta additions module are named separately for a reason.

```
dependencies {
  implementation("ai.koog:koog-agents:1.0.0")

  // Add only if you accept beta churn intentionally.
  // implementation(
  //   "ai.koog:koog-agents-additions:1.0.0-beta"
  // )
}
```

That comment is not stylistic advice. It is an architectural marker. A production system should be able to answer a simple question at any time: which parts of this codebase depend only on stable Koog guarantees, and which parts deliberately accept more volatility. If you cannot answer that question from the repository structure and dependency declarations, the boundary is already too weak.

Stability is multi-dimensional, not a single label

Engineers often compress release maturity into one binary judgment: stable or unstable. That is too coarse to be useful. In practice, you need to reason about at least four kinds of stability.

Language and runtime stability concerns whether the framework continues to operate with your chosen JDK, Kotlin plugin, build tool, and surrounding runtime environment. This is the floor established during project bootstrapping.

API stability concerns whether public types, method signatures, and documented entry points change incompatibly between releases. Koog's stable modules give the strongest guarantees here.

Behavioral stability concerns whether the same code continues to behave the same way under real execution: persistence format compatibility, trace semantics, workflow edge cases, tool invocation timing,

and similar operational details.

Operational maturity concerns how confident you are running the capability under production load, failure, observability, and recovery conditions. A feature can be API-stable and still be immature in your environment if you have not exercised it under realistic traffic and failure modes.

When you evaluate Koog versions, do not ask only whether the dependency is labeled stable. Ask which dimension of stability your application actually needs for each subsystem. A domain-facing agent API exposed to several services needs stronger guarantees than an internal prototype module. A persistence boundary needs stronger behavioral validation than a transient formatting helper. A beta integration behind an adapter may be acceptable if the adapter contains the risk. The same beta integration spread through public service interfaces is usually a mistake.

Stable core means more than "safe to compile"

The stable Koog core should be your default for any code that forms part of the application's long-lived contract: agent construction paths, domain-facing service boundaries, test fixtures, and integration surfaces that other teams will call. The reason is not only source compatibility. Stable core adoption reduces the blast radius of routine upgrades. If Koog adds APIs or deprecates older ones in minor releases without breaking existing stable public APIs, you can plan upgrades around normal engineering cadence instead of emergency rewrites.

This matters especially in code that will outlive the first experiment. If your team is building an internal library around agents, a reusable orchestration layer, or a domain service consumed by other applications, keep that layer on the stable umbrella module unless a missing capability forces a narrower exception. Stable code should depend on stable framework surfaces wherever possible. That principle sounds obvious,

but it is often violated by convenience: a developer discovers a helpful beta addition and imports it directly into the core service module. The benefit is immediate. The cost appears months later when a minor version refresh changes behavior or signatures in the beta area and the whole service surface must move with it.

Beta is a design tool, not a forbidden zone

Beta modules are not inherently reckless. They are useful when they unlock meaningful capabilities that justify the migration risk. The mistake is not using beta at all; the mistake is using beta without containment. Koog's documented compatibility between stable `X.Y.Z` modules and beta `X.Y.Z-beta` modules is helpful because it lets you combine a stable core with explicitly experimental edges and update them in sync. That compatibility should encourage modular design, not complacency.

Treat beta as a tool for bounded experimentation. If a beta feature delivers strong value—for example, a new workflow utility, a framework integration, or a higher-level addition that would otherwise require substantial bespoke code—isolate it behind one or more of these boundaries:

- A dedicated integration module with no domain ownership.
- An internal adapter interface that the rest of the application depends on instead of the beta API directly.
- A feature flag that allows you to disable the beta-backed path without removing the stable core behavior.
- Contract tests that validate only the application-facing behavior, allowing internal implementation changes during upgrades.

That isolation converts beta from a whole-system dependency into a local risk decision. You are no longer asking whether the framework

feature is perfect. You are asking whether the value of the feature is worth maintaining one boundary that may evolve faster than the rest of the codebase.

Boundary placement determines upgrade cost

The cleanest way to reason about release maturity is to map it to code ownership. Stable Koog dependencies belong in modules that define application contracts. Beta Koog dependencies belong in modules that can be rewritten with limited collateral damage. If you adopt that rule consistently, upgrades become localized engineering work. If you ignore it, every upgrade becomes a negotiation across unrelated teams and services.

Consider two repository shapes. In the first, the service module depends only on `koog-agents`; any beta-backed convenience lives in an internal adapter submodule. In the second, the main service module imports both stable and beta APIs directly, and test utilities also embed beta-specific behavior. The first shape allows a beta migration to be handled by one team within one module, provided the adapter contract remains steady. The second shape turns the same migration into a cross-cutting refactor touching service code, tests, and possibly downstream consumers.

That difference is architectural, not cosmetic. It affects how fast you can move, how confidently you can upgrade, and how much of your operational risk depends on a subset of framework features that may still evolve quickly.

Do not confuse source compatibility with operational equivalence

A stable upgrade that compiles cleanly is not automatically operationally equivalent. This distinction matters more in agent systems than in ordinary utility libraries because the framework participates

in execution behavior, not just data transformation. A minor release may preserve public APIs while still affecting trace detail, persistence interactions, retry boundaries, or workflow edge cases. None of those changes necessarily violate semantic versioning if the documented contract remains intact, but they may still matter in production.

You should therefore validate upgrades at the level of runtime behavior, not only at the level of compilation. Three areas deserve special attention.

First, validate *tool contracts*. If a tool is exposed to an agent with typed inputs and downstream effects, confirm that invocation patterns, serialization assumptions, and failure propagation still behave as expected after the upgrade.

Second, validate *workflow entry points*. If your system depends on particular staged execution paths, verify that the same prompts, tool decisions, or workflow constraints still lead to acceptable run shapes.

Third, validate *persistence and resumption*. If runs may be resumed or inspected later, test replay-like scenarios around checkpoints or persisted state. Changes that preserve APIs but alter state interpretation can be more damaging than a loud compilation failure, because they surface only under load or recovery conditions.

A practical module strategy

A useful default layout separates stable core logic from experiment-prone integrations. The exact directory names do not matter; the dependency directions do.

```
app-service
 -> depends on agent-core

agent-core
 -> depends on ai.koog:koog-agents:1.0.0

agent-beta-integrations
 -> depends on agent-core
```

```
 -> depends on ai.koog:koog-agents-additions:1.0.0-beta

agent-testkit
 -> depends on agent-core
```

In that shape, `agent-core` defines the stable agent-facing abstractions your application relies on. `agent-beta-integrations` may add optional capabilities, but the application does not take a compile-time dependency on beta types unless it explicitly opts into them. `agent-testkit` should also prefer stable abstractions. If you bind tests directly to beta implementation details, you increase churn and make version upgrades appear larger than they are.

This strategy also helps with incident response. When a runtime issue emerges after a framework update, you can narrow the suspect area quickly. If beta-backed modules are segregated, you know where to look first.

When to accept beta in production

There are legitimate reasons to run beta Koog components in production. The right threshold is not "only in prototypes." The right threshold is whether the value justifies the engineering posture required to operate it safely. Accept beta in production when all of the following are true:

- The capability solves a real problem that would otherwise force substantial custom implementation or operational compromise.
- The dependency is isolated behind an internal boundary.
- You have rollback or disablement options.
- You have targeted tests around the behavior that matters.
- The team is willing to read release notes and perform adaptation work during minor upgrades.

Do not accept beta in production when it becomes part of a public or widely reused application contract, when rollback is impossible, or when the team expects upgrades to be effortless. A beta feature can be operationally acceptable if its scope is narrow and its value is high. It becomes reckless when it is allowed to define the center of the application.

Upgrade cadence is a policy choice

Every codebase implicitly chooses an upgrade cadence: aggressive, periodic, or conservative. Koog's stable-versus-beta split should influence that cadence. Stable-core-only projects can usually adopt a periodic upgrade model: refresh intentionally, validate with regression tests, and rely on compatibility guarantees to keep the work bounded. Projects using beta additions need a more deliberate rhythm. They should still avoid stagnation, but they must reserve time for adaptation because beta surfaces may change across minor releases.

A simple rule helps: upgrade stable dependencies on a schedule; upgrade beta-backed modules behind an explicit work item. That prevents a transitive refresh from accidentally migrating a riskier part of the system without review. In Gradle-based repositories, the discipline can be expressed through centralized version declarations rather than scattered literals. Even if you are not yet using a version catalog, keep Koog version definitions centralized enough that changes are visible and reviewable.

Read release notes with architectural intent

Release notes and migration guides are not only for library maintainers or build engineers. If you use Koog to drive stateful, tool-capable systems, each release note should be read with three architectural questions in mind.

Did any stable public API deprecations touch your long-lived bound-

aries?

Did any behavioral changes affect persistence, observability, or execution semantics that your operators rely on?

Did any beta additions you use change enough that your containment boundaries need adjustment?

This style of reading is different from scanning for new features. You are looking for signals that require either code changes or validation work. Even when an upgrade appears low risk, note whether it touches operationally sensitive areas. A harmless-looking observability change can still invalidate dashboards or trace-based diagnostics. A persistence-related adjustment can require replay testing even if no compiler errors appear.

Testing for upgrades means more than unit coverage

Upgrade safety depends on test shape, not just test count. A project with many unit tests can still be fragile if none of them exercise the important framework boundaries. For Koog, the highest-value upgrade tests are usually narrow integration tests around agent construction, tool invocation, workflow entry points, and resume-or-replay scenarios if persistence is involved.

Keep these tests focused on application contracts rather than framework internals. For example, assert that a given agent path can invoke the expected tool and produce a valid domain result under a controlled input scenario. Do not overfit tests to every internal trace detail unless that detail is part of an operator-facing contract. The goal is to detect meaningful regressions without turning the test suite into a frozen snapshot of Koog internals.

If you use beta additions, add one more category: adapter conformance tests. These tests should exercise the interface your stable code depends on, not the raw beta API. When the beta surface changes, you

repair the adapter and keep the rest of the application untouched.

Anti-patterns that destroy upgrade boundaries

A few patterns almost guarantee upgrade pain.

One is *API leakage*. Beta framework types appear directly in service interfaces, DTOs, or domain-facing abstractions. Once that happens, a framework update becomes an application migration.

Another is *implicit transitive adoption*. A stable module pulls in a beta-backed helper through an internal convenience dependency, and developers start using it casually because it is already on the classpath. This is one of the strongest arguments for module isolation and explicit dependency review.

A third is *operational blindness during upgrades*. Teams run compile and unit tests, declare victory, and discover later that resumed runs, trace ingestion, or tool retry behavior changed in production. Agent systems punish shallow upgrade validation because the most important failures are often temporal or stateful.

A fourth is *confusing compatibility with sameness*. Stable semantic versioning means you can expect disciplined API evolution. It does not mean every performance characteristic, trace payload, or operational nuance remains identical forever.

Decision signals for drawing the boundary

When you decide whether a Koog feature belongs on the stable side of your architecture or behind a beta boundary, use concrete questions.

How expensive would it be if this dependency changed next quarter?

How many modules or teams would be forced to update with it?

Does this code define application behavior, or is it only an internal implementation detail?

Can you disable or replace it without rewriting domain logic?

Would an operational regression here be visible immediately, or only during recovery and incident handling?

If the answers point toward broad impact, difficult rollback, or deep coupling to business logic, prefer stable surfaces. If the answers point toward local implementation detail, clear containment, and high strategic value, a beta-backed edge may be reasonable.

A disciplined upgrade playbook

A Koog upgrade should follow a short, repeatable routine.

- Review the target release notes and identify stable versus beta surfaces you actually use.
- Update version declarations centrally rather than ad hoc.
- Compile and run the existing test suite.
- Run targeted integration tests around tools, workflows, and any persistence or resume paths you operate.
- Inspect observability outputs that your operators depend on.
- If beta-backed modules changed, repair adapters before touching domain code.
- Promote the upgrade only after runtime behavior matches your operational expectations.

The playbook is intentionally narrow. The important part is not ceremony; it is ensuring that each upgrade is handled as both a dependency event and a runtime-behavior event.

Koog's version model gives you room to be deliberate. Stable core adoption lets you build durable contracts. Beta additions let you experiment

where experimentation is worth the cost. The quality of the system depends on whether those two modes remain visibly distinct in the codebase, in the test strategy, and in the way you plan upgrades over time.

2.4. Choosing a Project Shape for JVM-Only, Mixed-Language, and Multiplatform Systems

The first structural decision in a Koog codebase is not which model you call or which tool you expose. It is where the agent logic lives and who is allowed to depend on it. That choice becomes expensive to reverse once the code starts accumulating transport adapters, persistence logic, tests, and integrations with the rest of your platform. Teams often postpone the decision because all three common shapes can compile the same introductory example. The real differences appear later: when a Java service must call the agent layer cleanly, when an Android client wants to share some reasoning logic, or when a multiplatform build starts colliding with server-only plugins and libraries.

A small bootstrap example makes the contrast concrete. A JVM-only Koog service can start with a minimal Gradle build and a single `main` source set.

```
plugins {
 kotlin("jvm") version "2.2.0"
}

repositories {
 mavenCentral()
}

dependencies {
 implementation("ai.koog:koog-agents:1.0.0")
}

kotlin {
 jvmToolchain(17)
```

```
}
```

That build is intentionally narrow. It is easy to compile, easy to package, and easy to integrate into existing server infrastructure. The question is whether that simplicity matches the system you are actually building. If the agent code will only ever run server-side and the rest of the organization is already JVM-centric, this shape is usually the right default. If the code must straddle Java services, mobile targets, or multiple runtime surfaces, you need stronger boundaries from the start.

Project shape is an ownership decision

A project shape is best understood as a statement about ownership and execution locality. You are choosing where Koog-centric code executes, which modules are allowed to know about it directly, and which concerns must stay outside the shared core. That is why project shape affects more than packaging. It influences API design, build complexity, operational control, upgrade strategy, and how much platform-specific detail leaks into domain logic.

Three shapes dominate in practice:

- A *JVM-only service*, where Koog runs entirely inside a server-side Kotlin or Java/Kotlin application.
- A *mixed-language JVM system*, where Kotlin hosts agent code but Java remains substantial in the surrounding system.
- A *Kotlin Multiplatform core*, where selected agent logic is shared across targets and platform-specific edges remain isolated.

None of these is universally superior. The right choice depends on what must be shared, what must stay close to infrastructure, and how much build and interop complexity your team is willing to absorb.

Start by asking where the code must execute

The cleanest first filter is execution location. If provider access, persistence, tracing, and tool side effects all occur on the server, a JVM-only deployment is usually the lowest-friction answer. You avoid multiplatform plugin interactions, keep your runtime environment uniform, and preserve straightforward access to JVM libraries, container tooling, and mature web frameworks.

If some logic truly needs to execute in more than one place, separate that need into categories. Shared prompt wording is not the same as shared workflow policy. Shared workflow policy is not the same as shared provider integration. The farther you move toward external systems—HTTP clients, database drivers, framework starters, credential resolution, mobile background execution, device storage—the less likely the code is to be worth sharing across targets. Multiplatform is strongest when the shared portion is narrow, stable, and intentionally free of platform-heavy dependencies.

JVM-only is the default for a reason

For most production Koog systems, JVM-only is the most economical starting point. The reasons are practical rather than ideological. Koog is comfortable on the JVM. Java and Kotlin are both first-class there. Server frameworks, monitoring stacks, schedulers, and persistence technologies are mature. CI and container support are predictable. The build is easier to reason about, and your team can focus on agent behavior rather than cross-target build mechanics.

A JVM-only shape also gives you maximum operational control. Providers usually live behind server-side credentials. Tools often touch internal systems that should never be directly exposed to client applications. Persistence and checkpointing are commonly server concerns. Observability pipelines are server concerns almost by definition. If your architecture already assumes those capabilities will

remain centralized, sharing the agent core across client runtimes may create more complexity than value.

A minimal JVM module remains straightforward even as the codebase grows. You can add Spring Boot or Ktor integration in dedicated modules, isolate beta surfaces behind adapters, and keep deployment packaging conventional. That stability matters when the system is still finding its behavioral boundaries.

Signals that JVM-only is the right choice

Choose JVM-only when most of the following are true:

- Agent execution happens in backend services.
- Provider credentials and tool permissions must remain server-side.
- You need direct access to JVM ecosystem libraries and frameworks.
- The team values lower build complexity over speculative reuse.
- Java interoperability matters, but cross-platform code sharing does not yet justify the added constraints of multiplatform.

This choice is especially strong for service-oriented organizations where the agent is a backend capability consumed over HTTP, messaging, or RPC rather than embedded into rich clients.

Mixed Kotlin/Java JVM is not a transitional accident

Many real Koog deployments land in a mixed-language JVM estate. Treat that as a first-class architecture, not as an awkward halfway state on the way to "pure Kotlin." Kotlin is often the best place to author the agent layer because it expresses DSL-style configuration and structured concurrency cleanly. Java may still dominate elsewhere: service

interfaces, domain services, older modules, integration libraries, or entire product lines.

That mixed reality forces an important design discipline. You must decide whether the Kotlin Koog layer is internal implementation or a public surface called from Java. If it is internal, you can keep the code idiomatic and narrow the interop boundary to a small adapter. If Java callers consume the API directly, you need to design with Java ergonomics in mind from the start: explicit nullability, stable method-based facades, and restraint around exposing Kotlin-specific patterns as the only entry point.

This is where teams often make a costly mistake. They write elegant Kotlin APIs that are pleasant inside Kotlin tests, then discover later that Java callers face awkward construction paths, implicit default parameters, or coroutine-centric interfaces with no natural Java representation. Retrofitting a JVM-friendly API after multiple services have already integrated with the agent layer is difficult. Decide early who the callers are.

How to shape a mixed-language boundary

In a mixed Kotlin/Java system, a useful pattern is to let Kotlin own the Koog-rich internals while Java sees a narrower application-facing interface. The boundary should translate from Java-friendly concepts into the richer Kotlin implementation details. That keeps the expressive advantage of Kotlin without forcing the rest of the estate to adopt Kotlin idioms immediately.

A repository layout might look like this:

```
agent-api
 -> Java-friendly service interfaces and DTOs

agent-core
 -> Kotlin implementation using Koog
```

```
agent-http
 -> transport adapters, controllers, handlers
```

In that arrangement, `agent-api` stays conservative. It should avoid leaking framework-specific or Kotlin-specific assumptions unless every caller can tolerate them. `agent-core` can use Kotlin expressively because it is not the boundary consumed by the rest of the platform. This separation also helps with testing and version management. Stable contracts can remain steady even if the Koog internals evolve.

Signals that mixed-language JVM is the right choice

Choose a mixed Kotlin/Java shape when these conditions dominate:

- The platform is already heavily JVM-based.
- Kotlin is welcome for new agent-centric modules, but Java remains a significant consumer or host.
- You want Koog close to existing web, data, or service layers without forcing a broad language migration.
- Interoperability quality matters as much as DSL elegance.
- Cross-platform reuse is secondary to backend integration.

This shape is often the most pragmatic answer in established organizations. It gives you Koog where it adds the most value while respecting the system that already exists.

Multiplatform is powerful when the edges are honest

Kotlin Multiplatform is attractive because it promises shared logic across JVM, Android, iOS, JavaScript, or WebAssembly-oriented targets. That promise is real, but only for code that is genuinely portable. If you choose this shape, the most important discipline is to keep

platform-specific edges explicit rather than pretending everything belongs in shared code.

The center of a multiplatform project is the `kotlin {}` block. That is where you declare targets such as `jvm()`, `androidTarget()`, `js()`, `iosArm64()`, `wasmJs()`, or `wasmWasi()`. A minimal skeleton may look like this:

```
plugins {
 kotlin("multiplatform") version "2.2.0"
}

kotlin {
 jvm()
 iosArm64()
 js()
 wasmJs()

 sourceSets {
 val commonMain by getting
 val jvmMain by getting
 }
}
```

This syntax is simple enough to read, but the implications are not simple. Shared code in `commonMain` must stay within the subset of dependencies and APIs that can exist across the chosen targets. Server-specific provider integrations, persistence technologies, and framework starters rarely belong there. If you push those concerns into shared code, you either fail the build or degrade the design into weak abstractions that serve no platform well.

Only share what earns the right to be shared

A multiplatform Koog design works when you share the narrow core that truly benefits from cross-target consistency. Good candidates include domain rules for prompt shaping, some workflow policy, schema definitions for structured requests and responses, and lightweight decision logic that is meaningful on every target. Poor candidates include provider client setup, server-side tool

implementations, database-backed persistence, or direct coupling to web frameworks.

This distinction is easiest to see in terms of dependency gravity. Code tends to become less portable as it moves outward toward infrastructure. The fact that a file compiles in `commonMain` is not enough reason to leave it there. Ask whether the code *should* be shared. If a shared abstraction exists only to hide obvious platform differences, the abstraction may already be a liability.

Multiplatform comes with build-shape constraints

Multiplatform is not merely "JVM plus more outputs." It changes how your build behaves. The `kotlin {}` block becomes the center of configuration. Only one Android target is allowed per Gradle subproject. Some Gradle Java ecosystem plugins, including the traditional Java, Java Library, and Application plugins, have compatibility caveats or deprecated interactions in multiplatform contexts. Those constraints are not trivia. They influence how you split modules and where you place server-only responsibilities.

A common mistake is to force Java-oriented plugin behavior directly into a multiplatform root project because it feels convenient early on. That convenience fades quickly once the build grows. A more robust strategy is to separate a shared multiplatform core from JVM-specific subprojects that own server frameworks, Java plugin interactions, and runtime packaging. If a module needs the full freedom of the Java ecosystem, make it a JVM module rather than stretching the multiplatform root to accommodate it.

Signals that multiplatform is the right choice

Choose a multiplatform shape only when the reuse case is concrete. Strong signals include:

- You have clearly identified agent-related logic that must behave consistently across multiple targets.

- That shared logic is mostly domain or workflow policy rather than infrastructure integration.

- The team can tolerate higher build complexity and stricter module boundaries.

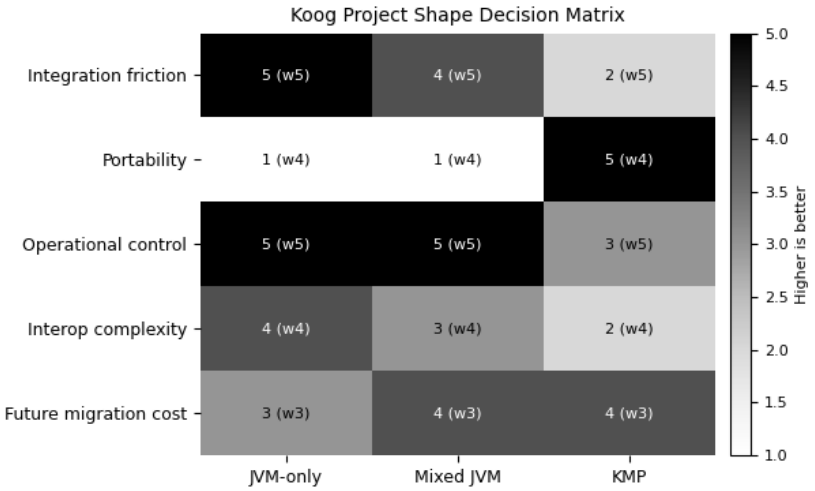
- Platform-specific edges can remain isolated without constant pressure to leak into shared code.

- The long-term value of reuse outweighs the cost of more complex builds, testing, and plugin management.

Do not choose multiplatform because it feels strategically modern. Choose it because you can point to specific code that would otherwise be duplicated, diverge over time, and create meaningful maintenance cost.

Decision matrix for project shape

When the trade-offs still feel close, score the options against the forces that matter most operationally.



The weighting is not universal, but the pattern is instructive. JVM-only usually wins on integration friction and operational control. Multiplatform wins on portability but pays for it in build and interop complexity. Mixed JVM sits in the middle: slightly more complex than JVM-only, far simpler than multiplatform for most server-heavy organizations, and often the best match for incremental adoption.

Do not let provider and transport details dictate the core

Regardless of shape, a healthy Koog architecture keeps provider setup, transport adapters, and persistence technologies from dictating the structure of the shared or domain-facing core. This is especially important when comparing JVM-only and multiplatform designs. The temptation is to shape modules around whatever is easiest to wire initially: a Spring service, a Ktor route, a provider-specific client, or a mobile networking layer. That choice tends to produce short-term convenience and long-term rigidity.

Instead, let the architectural center be the agent-relevant logic that should remain coherent as transports and platform edges vary. In a

JVM-only service, that often means one or more internal modules that own agent behavior and tool contracts, with HTTP or messaging modules around them. In multiplatform, it means a genuinely portable core with platform-specific provider, persistence, and transport adapters outside it. The rule is the same in both cases: keep the edges honest.

Migration cost depends on what you freeze early

Teams sometimes frame project shape as if it were irreversible. It is not, but the cost of change depends on what has already become public or deeply reused. Moving from JVM-only to mixed JVM later is usually manageable if your Koog layer already sits behind a clean service interface. Moving from a sprawling JVM-only module directly into multiplatform is much harder if server-specific dependencies, web framework annotations, and provider details have seeped through the codebase.

That suggests a practical tactic: choose the simpler shape now, but keep the core boundaries clean enough that extraction remains possible later. You do not need a full multiplatform build to preserve optionality. You need disciplined module boundaries and refusal to let infrastructure-heavy code masquerade as domain logic.

Common anti-patterns

Several mistakes recur across all three shapes.

One is *sharing too much too early*. A multiplatform project starts by pushing provider integration, persistence interfaces, and server-specific error handling into `commonMain`. The resulting code is technically clever and architecturally weak.

Another is *treating mixed-language design as a compiler problem*. Java compatibility is not solved just because the Kotlin code compiles to JVM bytecode. You still need public APIs that humans can use comfortably.

A third is *embedding transport concerns into the agent core.* Controllers, routes, and message handlers begin to own prompt construction, orchestration, or provider setup. That makes every future shape change harder because the Koog logic is no longer a coherent subsystem.

A fourth is *selecting multiplatform for prestige.* If you cannot identify exactly what must be shared and why that reuse matters economically, multiplatform is probably the wrong first move.

A practical decision sequence

A useful decision flow starts with five questions:

- Where will the agent execute in production?
- Which callers need direct access to the agent-facing API?
- What code must truly be shared across platforms?
- Which concerns are irreducibly platform-specific?
- How much build and interop complexity is the team prepared to own for the lifetime of the system?

If the answers emphasize server-side execution, server-side secrets, and close integration with existing JVM infrastructure, choose JVM-only. If the same environment also includes substantial Java callers or Java-owned modules, choose mixed JVM and formalize the interop boundary. If the answers identify durable, high-value shared logic across multiple targets and you can keep infrastructure out of the shared core, choose multiplatform.

What to optimize for

Early in a Koog codebase, optimize for maintainable control rather than theoretical flexibility. The best project shape is the one that makes

the agent subsystem easy to evolve, test, and operate without forcing unnecessary complexity into every surrounding module. JVM-only is often the right answer because it keeps the architecture aligned with where the operational weight already lives. Mixed Kotlin/Java is often the right answer when your organization is evolving gradually and needs clean interop more than language purity. Multiplatform is the right answer when you can defend every shared module with a concrete reuse case and keep the platform-specific edges from leaking inward.

A strong starting shape does not eliminate future change. It makes that change local, intentional, and affordable because the boundaries were drawn before convenience hardened into architecture.

Chapter 3

The Core Agent Model and Provider Abstractions

A great many agent failures begin with a category error: teams think they are wiring up "a model call," when they are really defining an execution contract that governs prompts, state transitions, streaming behavior, and provider dependence. This chapter gives the reader a precise mental model of that contract so later chapters on tools, workflows, memory, and operations can build on stable ground rather than folklore.

3.1. Instantiating and Running Agents

A Koog agent looks small in code, which is exactly why teams often misread what they are creating. You can instantiate an `AIAgent`, pass

a model, call `run()`, and get a result back in a few lines. That surface simplicity hides four distinct contracts that need to stay separate if you want predictable behavior: the prompt executor that knows how to talk to model providers, the model binding that selects a concrete capability, the agent configuration that gives the run its durable behavioral frame, and the invocation input that changes from one run to the next. If you blur those layers together, debugging quickly turns into prompt archaeology.

A minimal runnable agent is the right place to make those boundaries explicit because every later feature—tool calls, graph workflows, checkpointing, streaming, rerouting—depends on the same execution core. The point is not merely to get text back from a model. The point is to understand what has been fixed at construction time, what remains variable at run time, and where Koog turns a single model call into an agent run with its own lifecycle.

A compact example makes that separation concrete early.

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.prompt.executor.llms.MultiLLMPromptExecutor
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
 val executor = MultiLLMPromptExecutor(
 llmClients = mapOf(
 /* provider -> client bindings */
 )
 )

 val agent = AIAgent<String, String>(
 promptExecutor = executor,
 llmModel = /* concrete LLModel */
 )

 val result = agent.run(
 "Extract the main operational risk from this text: " +
 "'The backup job completed, but restore validation " +
 "was skipped for three weeks.'"
 )

 println(result)
```

```
    executor.close()
}
```

This example is intentionally sparse. It does not add tools, sessions, memory, workflow nodes, or provider routing policy. It shows the smallest useful shape of an agent run:

- construct a `PromptExecutor`,
- construct an `AIAgent`,
- invoke the suspend `run()` operation with per-call input,
- receive typed output,
- close the executor when you are done with it.

That sequence already tells you something important: a Koog agent is not the provider client. The client lives underneath the prompt executor. The agent is not the model either. The model is passed as a binding that the executor uses when it executes prompts. The run input is not agent configuration. It is the stimulus for a particular invocation. Once you hold those distinctions firmly, later design choices stop looking arbitrary.

Construction-time state versus run-time state At instantiation time, you define the stable part of execution. That usually includes the prompt executor and the chosen model, and may include a durable system prompt or other agent configuration depending on how you structure the agent. These are construction-time decisions because they shape behavior across many runs. They answer questions such as:

- Which provider clients are available?

- Which concrete model should this agent use by default?
- What baseline instructions frame the agent's role?
- Which registries or features are attached to the runtime?

At run time, you supply the changing part: the input payload for this invocation. That payload might be a string, a structured request object, or another application-specific type depending on the agent's input type parameter. The key operational rule is simple: if a value changes per request, do not bury it in agent construction. If a value expresses long-lived behavior or infrastructure, do not smuggle it through the invocation input.

A common early mistake is to rebuild the agent for every request because the request text changes. That inflates latency, complicates resource management, and makes it harder to reason about what is actually stable. Build an agent when you want a reusable execution participant. Invoke `run()` when you want to process a new unit of work.

Why `run()` is suspend Koog makes agent execution suspendable because the run is not a trivial local function. It may involve network I/O to a provider, internal iterative control, structured output handling, and later, tool execution or streaming. Even the minimal case is still an asynchronous interaction with an external model backend. Putting `run()` inside a coroutine scope forces you to acknowledge that reality in your program structure.

```
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
 // agent construction omitted
 val output = agent.run("Summarize the incident in one line.")
 println(output)
}
```

For a backend service, you would usually call `run()` from an existing coroutine-aware request handler rather than wrapping every invoca-

tion in `runBlocking`. The example uses `runBlocking` because it makes the lifecycle visible in a small standalone program. The important part is not the entrypoint style. The important part is that agent execution belongs in a coroutine-managed lifetime.

The minimal runtime vocabulary Before following a run end to end, define the terms precisely.

Prompt executor. This is the execution boundary between Koog and one or more model providers. It accepts a prompt, a model, and optionally tool descriptors, then returns an assistant message. The core `PromptExecutor` contract is where model invocation actually happens.

Model binding. This is the concrete `LLModel` selected for the run. The executor uses it to dispatch to the right provider client and capability path.

Agent. This is the runtime participant that coordinates prompt construction, execution, and response handling under a strategy-driven lifecycle.

Run input. This is the per-invocation data passed to `run(agentInput)`. It is not transport configuration and not provider state.

Output. This is the typed result returned by the agent. In the minimal case it may be plain text. In richer designs it may be a structured domain object.

These terms matter because they let you ask the right debugging questions. If the wrong model was used, investigate the model binding and executor dispatch. If the tone is wrong across all invocations, inspect the stable prompt frame or agent configuration. If one request behaves strangely while others do not, inspect the run input and the prompt assembled from it.

Following one run from input to output A single

`agent.run(...)` call reads like a one-step operation, but internally it is a lifecycle. In the minimal case, that lifecycle has five phases.

Phase 1: input arrival. Your application passes an input value of type `Input` to the agent. This is the invocation boundary. Validation that belongs to the application domain should usually happen before the agent sees the payload. Do not rely on prompt instructions to sanitize malformed requests.

Phase 2: prompt assembly. The agent transforms durable configuration plus the invocation input into a `Prompt`. This is where the system prompt, user content, and any internal framing become a provider-ready request. Even if you never construct a `Prompt` directly in your own code, the runtime still has to do that assembly.

Phase 3: model invocation. The agent passes the assembled prompt and the selected `LLModel` to the prompt executor. At this point provider-specific transport, authentication, and request formatting are the executor's concern rather than the agent's.

Phase 4: response handling. The executor returns a `Message.Assistant`. The agent interprets that response according to its runtime contract. In the minimal case, this may simply become the final output. In more complex cases, the response may trigger another internal step.

Phase 5: iterative continuation. The run may continue rather than terminate after the first model response. This is where Koog begins to differ from plain request-response wrappers. The runtime can treat one model output as an intermediate state in a controlled loop rather than as the immediate final answer.

That last phase deserves careful attention because it changes how you think about agent execution.

A run is not necessarily a single completion Experienced backend developers often start with a chat-completion mental model: send messages, receive text, return it to the caller. Koog's agent runtime can support that pattern, but it does not reduce the run to that pattern. The framework treats execution as a strategy-driven process. Even before introducing external tools or graph nodes, you should think in terms of steps, not just responses.

This does not mean every run becomes an elaborate loop. Many runs will terminate after one prompt execution. The point is that the framework's contract leaves room for controlled continuation. A response can be interpreted as something other than final user-facing prose. It can be an intermediate artifact inside a larger execution policy. That distinction becomes crucial later when the runtime starts handling tool calls, retries, validations, or branching. If you understand it now at the minimal level, those later features feel like extensions of one consistent model rather than unrelated special cases.

Prompt assembly is a first-class operation Teams often treat prompt creation as string concatenation because the first prototype really was just a template plus user text. That habit becomes expensive once behavior becomes important. Prompt assembly deserves first-class status because it is the point where durable behavior and request-specific data are joined.

For a minimal agent, prompt assembly usually combines at least two logical components:

- a stable behavioral frame, often expressed through system-level instructions,
- the invocation payload, which supplies the concrete task or content.

Keep those two roles separate in your design even if the final provider

request serializes them into a single prompt object. The stable frame should express durable behavior: tone, output constraints, or task identity. The invocation payload should express the work item for this run. If you start embedding business rules, edge-case handling, and request-specific state indiscriminately into one giant system prompt, you make the agent harder to test and nearly impossible to evolve safely.

A useful practical rule is to ask whether a change should affect one request or many. If the answer is one request, push it into invocation data or into a typed parameter that feeds prompt assembly. If the answer is many requests, it may belong in the stable frame or agent configuration.

The executor owns model-facing behavior The prompt executor is the layer that keeps provider logic from leaking into agent code. Its verified core operation accepts a `Prompt`, an `LLModel`, and an optional list of tool descriptors, then returns a `Message.Assistant`. You do not need to manually orchestrate HTTP calls or provider-specific payloads at the agent level. That is the executor's job.

This matters even in the minimal case because it draws a durable boundary:

- the agent decides *what* should be executed,
- the executor decides *how* to execute it against the bound provider.

That separation is what later enables model switching and rerouting without rewriting the agent's control logic. It also gives you a clean place to manage executor lifetime. Prompt executors are closeable. If you allocate one for a bounded program or test, close it explicitly. In a long-running service, manage it like any other shared client-side infrastructure object.

A typed agent is more than ergonomics Koog's `AIAgent<Input, Output>` type parameters are not cosmetic. They force you to state what the runtime accepts and returns. That makes a difference the moment you move beyond demos.

If your input is a raw string, you are telling the type system that prompt assembly starts from unstructured text. That is acceptable for simple tasks and examples. If your input is a domain object such as `TicketDraft` or `RiskReviewRequest`, you gain a place to encode validation, defaults, and explicit fields before prompt creation starts. The same logic applies to output. Returning `String` is fine for free-form text generation. Returning a domain type creates a stronger contract between the agent and the rest of the system.

The minimal runnable agent often starts as `AIAgent<String, String>` because that keeps attention on runtime flow. Treat that as a starting point, not a design ideal.

One-off runs and reuse A call to `run()` expresses one bounded execution. That does not mean the agent itself is single-use in every design. Koog distinguishes between one-off execution and broader session-oriented lifecycles elsewhere in the API surface, and those lifecycles matter when you need sequential reuse. For now, keep the operational distinction simple: the invocation is one unit of work; the agent instance may be reused if its configuration is intended to remain stable across many such units.

This distinction avoids two opposite mistakes. The first is rebuilding the agent per request when nothing about the durable configuration has changed. The second is treating a reused agent as if it automatically carries arbitrary conversational state across invocations. Whether state persists across runs depends on the specific lifecycle you are using, not on wishful thinking. A bare run should be treated as a bounded execution unless you have explicitly adopted a stateful session

model.

Thread safety and run lifecycle When the runtime documents a stateful single-use variant, it also documents locking and start-state constraints, including an `IllegalStateException` if the agent was already started. That tells you something general even if you are not using that exact variant: agent runs have lifecycle rules, and those rules are enforced. Do not assume that an agent instance is an infinitely reentrant black box.

In practice, you should decide early whether the agent instance is:

- a reusable configured component handling independent runs,
- a stateful execution object with a narrower lifecycle,
- or a per-request object created intentionally for isolation.

Make that decision explicit in your service architecture. Hidden lifecycle assumptions are one of the fastest ways to create race conditions, duplicate work, or invalid reuse.

A realistic minimal example A slightly fuller example shows how to keep the stable pieces stable while varying only the request payload.

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.prompt.executor.llms.MultiLLMPromptExecutor
import kotlinx.coroutines.runBlocking

data class IncidentInput(
 val summary: String
)

fun main() = runBlocking {
 val executor = MultiLLMPromptExecutor(
 llmClients = mapOf(
 /* provider -> client bindings */
 )
 )

 val incidentAgent = AIAgent<IncidentInput, String>(
```

```
 promptExecutor = executor,
 llmModel = /* concrete LLModel */
 )

 val result = incidentAgent.run(
 IncidentInput(
 summary = "Primary database failed over cleanly, " +
 "but write latency doubled for 18 minutes."
 )
 )

 println(result)
 executor.close()
}
```

What changes per invocation is `IncidentInput`. What remains stable is the executor and model binding. That is the shape you want for most production integrations: isolate durable runtime policy from request-specific content.

What counts as output The executor returns an assistant message. The agent returns an `Output`. Those are related but not identical concepts. The assistant message is a model-facing result inside the runtime. The output is the value the runtime exposes to your application after any internal handling is complete.

In the minimal case, the mapping may be almost trivial: assistant text becomes a returned string. Even then, keeping the distinction in mind helps. It reminds you that the framework reserves the right to do work between provider response and application-visible result. Later, that work may include structured extraction, internal continuation, or controlled action handling. If you treat raw assistant text and final agent output as interchangeable concepts from the beginning, you will make poor assumptions when the runtime grows more capable.

Compatibility-sensitive areas Agent construction is one of the places where framework ergonomics change most often. Constructor defaults, builder helpers, and model-binding DSLs are exactly the sort

of conveniences that evolve across releases. The stable concepts are more durable than the construction sugar:

- an `AIAgent` needs a prompt executor,
- execution binds to a concrete model,
- `run()` is suspend,
- the executor is a closeable execution resource,
- and the agent runtime may perform iterative continuation rather than a single completion.

If you keep your architecture anchored on those contracts, minor API reshaping is usually manageable. If you design around incidental conveniences from one release, upgrades become much more expensive than they should be.

Early anti-patterns Several mistakes appear almost immediately once teams move from playground examples to services.

Treating the system prompt as a dumping ground. If every rule, exception, validation branch, and business policy ends up in one enormous prompt, the runtime becomes opaque. Put durable behavioral framing in prompts; put executable policy in code.

Rebuilding agents per request. If the executor, model, and behavioral frame are stable, construct them once and reuse them deliberately. Rebuilding them for every request often adds cost without adding safety.

Conflating configuration with payload. A customer-specific request field is not the same thing as a provider binding or agent-wide instruction set. Keep request data and runtime setup in different places.

Assuming a run is just chat. A run may terminate in one step, but the

contract is wider than a bare completion call. Design with the possibility of iterative runtime control in mind.

Ignoring resource lifetime. Executors are not disposable string helpers. They encapsulate model-facing infrastructure and should be closed or managed as long-lived resources.

If you hold a clean mental model at this point, the rest of the runtime becomes much easier to reason about. You instantiate an agent by fixing the durable execution boundary: executor, model choice, and stable behavior. You invoke a run by supplying a per-call input. Koog assembles a prompt, executes it through the model-facing contract, interprets the assistant response, and may continue internally before producing the typed output your application actually consumes. That is the core unit of agent execution: not a raw completion request, but a bounded runtime with explicit construction, invocation, and lifecycle rules.

3.2. From Assistant Calls to Action-Capable Agents

A plain assistant call solves a narrow problem well: you provide input, the model produces language, and your application decides what to do with that language. That pattern is often enough for summarization, drafting, classification, and other bounded transformations. The trouble starts when surrounding code quietly expects more than language generation. A prototype that looked excellent in a prompt playground now has to choose whether another step is needed, hold onto intermediate intent, validate before acting, or recover when the first answer is incomplete. At that point, the difference between “generated text” and “runtime-controlled execution” stops being philosophical. It becomes an architectural fault line.

Koog is useful precisely because it does not treat an agent as a renamed

chat wrapper. It treats an agent as a structured, type-safe execution participant. The language model remains central, but it is no longer the only thing that defines behavior. The runtime can interpret responses, apply control policy, and continue execution step by step. That shift changes how you design inputs, outputs, failure handling, observability, and eventually action boundaries.

A small contrast makes the boundary concrete.

```
import ai.koog.agents.core.agent.AIAgent
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
    val answer = agent.run(
        "Read this incident note and produce a one-line " +
        "severity assessment."
    )

    println(answer)
}
```

Taken at face value, this looks like a text-generation call wrapped by an agent API. For a bounded task, that is exactly what it may be. The agent receives input, assembles a prompt, invokes the model, and returns a result. If the only contract is "produce text for the caller," you are still in assistant-style territory even though the runtime object is an `AIAgent`.

Now consider a different expectation for the same domain problem. The system no longer wants only a sentence. It wants the runtime to determine whether the note is complete enough, whether clarification is needed, whether a policy check must happen before the answer is considered valid, and eventually whether an external capability should be invoked under controlled rules. That is no longer a single assistant transaction. It is the beginning of agentic execution.

Assistant-style calls are bounded text transactions It helps to define the simpler case precisely before enlarging it. An assistant-style call has four stable characteristics.

- The primary output is language or another directly model-produced representation.
- The control loop lives outside the model interaction.
- Application code decides whether to retry, re-prompt, validate, or proceed.
- The model has little or no authority beyond generating a candidate response.

That pattern has real strengths. It minimizes surface area. It gives you a clean failure domain: if the model call fails, the transaction fails. Testing is straightforward because a single request maps to a single model output. Latency is easier to understand because there is usually one model invocation and little internal orchestration. Many production systems should stay in this mode longer than teams initially assume.

For example, if you need a concise summary of a support ticket, an assistant-style interaction is often ideal. The application sends the ticket text, receives a summary, stores it, and moves on. The surrounding service still owns business logic. The model is a specialized transformation engine, not an execution participant.

The pressure that breaks the assistant model The assistant pattern starts to fray when the application expects the model interaction to contribute to control flow rather than only to content generation. Several pressures push systems across that boundary.

Multi-step tasks. The first model response may not be the final deliverable. It may expose ambiguity, identify missing information, or create an intermediate artifact that needs another reasoning step.

Conditional branching. The system may need different next steps depending on what the model inferred. A harmless explanation request

and a suspected fraud report should not follow the same execution path.

Validation before action. Generated text is not necessarily executable intent. You may need policy checks, schema checks, or explicit runtime guards before you treat model output as actionable.

Separation of proposal from execution. The model can suggest what to do next, but something else must decide whether that suggestion is allowed, safe, complete, and timely.

Intermediate intent. Real work often requires state that is not yet final output: extracted entities, tentative plans, unresolved assumptions, or confidence-bearing hypotheses.

Once these pressures appear, it is not enough to say "we call the model again if needed." That may still work in an ad hoc prototype, but the control logic becomes scattered. Some steps live in prompt wording, some live in service code, and some are implicit in whichever developer last touched the retry logic. You end up with hidden workflows rather than explicit runtime behavior.

An agent is an execution participant Koog's stronger claim is that an agent is more than a wrapper around an LLM. That claim matters because it places the runtime, not just the prompt, at the center of system design. An action-capable agent still reasons through a language model, but it does so inside a controlled execution contract.

That contract introduces three important properties.

Stateful progression. The run can move through intermediate steps rather than collapsing everything into one response. Even when the run is short, the runtime treats execution as a sequence with meaning, not merely as a text return.

Decision-bearing outputs. A model response can be interpreted as a

request for another step, not just as user-visible prose. The response may encode intent, uncertainty, or a proposed action surface.

Policy-mediated continuation. The runtime, not the prompt alone, governs what kinds of continuation are possible. The agent can continue, halt, reject, ask for clarification, or later invoke tools under explicit control.

This is the key architectural shift: the model stops being the entire system and becomes one component inside a larger execution loop.

What does not change It is easy to overstate the shift and accidentally suggest that agents are autonomous software beings with opaque powers. That framing is unhelpful. Several fundamentals remain unchanged.

The model still produces the semantic content that drives reasoning. If the model is weak at a task, wrapping it in agent terminology will not fix that. Prompts still matter because they shape the model's behavior and output quality. Provider capabilities still matter because some models are better suited to structured responses, long context, or follow-on reasoning than others. The application still owns the operational boundary: authentication, authorization, data access, side-effect policy, and system-level recovery do not disappear into the agent runtime.

What changes is not the basic dependence on the model. What changes is where you place execution semantics. In an assistant integration, the application owns almost all control flow. In an action-capable agent, the runtime participates in that control flow in a structured way.

A practical distinction: text generation versus executable intent You can test whether a system has crossed into agentic behavior by asking a simple question: if the model produces text that looks like an action request, what happens next?

In an assistant-style design, the answer is usually "the application

parses it and decides." The model emits language, perhaps with a bit of structure, and all executable meaning is imposed afterward by external code.

In an action-capable design, the answer becomes "the runtime interprets that result inside an execution policy." The distinction is subtle but important. The runtime recognizes that some outputs are not merely content. They are step-control signals or candidates for controlled continuation.

That does not mean you should let generated text directly trigger side effects. Quite the opposite. The disciplined pattern is to separate *proposal* from *execution*. The model may propose an action, but the runtime decides how that proposal is represented, validated, authorized, and either continued or rejected.

Why prompt-only control is insufficient A common anti-pattern is to encode executable policy entirely as prose instructions. You tell the model things like:

> Never perform risky actions. Always ask before using external systems. Only continue when confidence is high.

Those instructions may improve behavior, but they do not create enforceable control boundaries. They are advisory text interpreted by a probabilistic model. If the surrounding system needs guarantees, policy has to exist somewhere stronger than prompt wording.

This is where agent runtimes earn their place. They let you move from "the model was told not to do that" to "the runtime does not permit that execution path without explicit handling." Chapter 3 takes over the detailed mechanics of tool exposure and invocation, but the conceptual boundary belongs here: prompts influence behavior; runtime policy constrains execution.

Iterative execution is the first real step toward agency The minimal run introduced earlier already hinted that a single invocation can contain more than one internal step. That idea becomes central here. An agent becomes action-capable not because it has access to tools by default, but because it can iterate under control.

Suppose a request asks for a deployment recommendation based on a short incident note. A plain assistant can produce a recommendation. An agentic runtime can instead do something closer to this:

- interpret the request and identify missing context,
- decide whether the note is sufficient for a recommendation,
- produce either a provisional answer or a clarification request,
- hold the result as part of a controlled execution state,
- continue only if the policy for the task allows the next step.

None of those steps requires a tool yet. The shift is already architectural. The model is no longer answering a question once. It is participating in a bounded control process.

That matters because many teams jump straight from assistant calls to tool invocation and conclude that tools are what make a system agentic. Tools are important, but they are not the first boundary. The first boundary is iterative, policy-aware execution.

Controlled action surfaces Once you accept that a runtime can continue after a model output, you also need to decide what kinds of continuation are legal. This is where controlled action surfaces enter.

A controlled action surface is any explicitly modeled operation that the runtime may consider invoking or advancing toward. The surface might eventually include external service calls, database lookups,

ticket creation, or notification dispatch, but the deeper idea is broader: not every possible next step should be expressible as free-form language.

Why does this matter? Because natural language is too permissive to serve as a production action interface. If the model can express anything and the application opportunistically interprets that text into behavior, you have built an accidental execution language with weak validation and poor auditability.

Controlled action surfaces reduce that ambiguity. They define *what* can happen, under *which* representation, and with *which* validation boundary. In Koog, the full mechanism appears later through tool registration and execution semantics. For now, keep the conceptual rule sharp: generated language should propose within a bounded action vocabulary, not invent the runtime's behavior ad hoc.

Why the distinction affects system design The difference between assistant-style and action-capable systems propagates into several engineering decisions.

Testing. A single-shot assistant is often tested with prompt fixtures and output assertions. An agent needs step-aware tests. You care not only about the final answer but also about whether the runtime continued, halted, or proposed an action path under the right conditions.

Observability. If one request can contain several internal decisions, a single request-response log entry is no longer enough. You need traces or step records that explain how the runtime reached its outcome.

Failure handling. A bad summary is a content-quality problem. A bad continuation decision is a control problem. They require different diagnostics and mitigations.

Interfaces. If downstream code expects a final answer, assistant-style output may be enough. If downstream code needs to know whether the

system is asking for clarification, waiting for validation, or ready to act, you need a richer contract than plain text.

Security and risk. The moment a model's output influences control flow, generated text becomes part of your execution boundary. You need stricter separation between what the model suggests and what the runtime can actually do.

This is why relabeling a chat wrapper as an agent creates false confidence. The name suggests a runtime discipline that the design does not actually provide.

Where Koog fits Koog gives you a vocabulary and runtime structure for this transition. The agent is typed. Execution is suspendable and strategy-driven. The runtime can coordinate model calls and, later, tool execution within a defined loop. The tool workflow itself is explicit rather than magical: create a tool, add it to a registry, pass the registry to the agent, and let the runtime execute tool calls under policy. The functional context even exposes multi-tool execution with a `parallelTools` flag, which makes the action surface an explicit runtime concern instead of a hidden prompt convention.

Those details matter because they show that Koog treats action as something with structure. Even the existence of modes such as sequential and parallel tool execution tells you that the framework assumes execution policy must be modeled explicitly. It is not merely parsing words and hoping the application does something sensible.

The practical consequence is that you should design your system around explicit boundaries from the start. Use the agent when you need model-mediated decisions inside a runtime contract. Stay with assistant-style calls when text generation is the whole job.

Decision signals for staying simple Not every problem benefits from agentic execution. You should remain in assistant mode when

several signals are present.

- The request is a bounded transformation from input text to output text.
- Application code already owns a stable deterministic workflow.
- There is no need for the model to choose among next-step options.
- Side effects are either absent or entirely external to the model call.
- Validation and policy checks are simpler outside the model loop.

Examples include summarization, tone rewriting, metadata extraction with independent post-validation, and static drafting tasks. In these cases, adding an agent runtime may increase complexity without increasing control.

Decision signals for moving to an action-capable agent You should start thinking in agent terms when the runtime itself needs to participate in decisions.

- The system must decide whether another reasoning step is required.
- Intermediate results affect what happens next.
- Clarification, validation, or conditional branching is part of the task.
- The model may eventually request controlled capabilities.
- The application needs a structured account of how the final result was reached.

Notice that none of these signals requires unconstrained autonomy. They only require that the model interaction be embedded in an explicit execution loop.

A concrete mental model A useful way to think about the progression is this:

- **Assistant-style call:** "Generate an answer and return it."
- **Action-capable agent:** "Participate in a controlled run that may reason, continue, and eventually request bounded actions."

The first is a language service. The second is a runtime participant. Both use a model. Both can be valuable. The mistake is to collapse them into one category and then expect production behavior to stay understandable.

Common false starts Several anti-patterns recur when teams try to make the transition too casually.

Renaming without redesign. Wrapping a chat call in an `agent` object does not create agentic execution if all control flow still lives outside and the runtime has no policy-bearing role.

Letting generated text trigger side effects. If the model says "create a high-priority ticket" and your application immediately does so by parsing that sentence, you have fused natural language with execution in the weakest possible way.

Encoding policy only in prompts. Instructions such as "never take action without approval" are useful behavioral guidance but weak enforcement. Runtime boundaries need stronger representation.

Assuming iteration is equivalent to conversation. A multi-step run is not just a chat transcript with more turns. It is an execution process whose intermediate states can carry operational meaning.

Hiding uncertainty. Assistant output often looks complete even when it is speculative. An agentic runtime should make room for clarification, refusal, or deferred continuation rather than forcing every state into final prose.

These failure modes are dangerous because they look productive at first. The system appears flexible until the first high-stakes case exposes that no real control boundary exists.

What belongs elsewhere Detailed tool definition, registry design, service adapters, and external capability wiring need their own treatment because they introduce concrete API and safety mechanics. The same is true for explicit graph orchestration, persistence, and recovery. Those topics expand the execution model; they do not define its first boundary. The essential conceptual move is already complete once you stop treating the model interaction as a one-shot text transaction and start treating it as a step inside a runtime with controlled continuation.

That shift gives you a cleaner way to assign responsibility. The model proposes. The runtime interprets. Policy constrains. External systems execute only through bounded surfaces. Once those roles are separated, the system stops depending on optimistic prompt wording and starts behaving like software you can reason about under load, failure, and change.

3.3. Binding Providers and Models with Prompt Executors

A model integration usually looks portable right up until the day you try to move it. The prototype worked against one hosted endpoint, so the team assumed the rest was plumbing. Then a compliance review requires a different provider, a local model is introduced for development, or a latency spike forces a reroute. Suddenly code that seemed

"model agnostic" turns out to depend on one vendor's authentication flow, one response shape, one capability assumption, and one naming scheme for models. The real problem is not the provider switch itself. The problem is that provider binding leaked into places where it never belonged.

Koog addresses that leak with a clear execution boundary: the `PromptExecutor`. Agent code should decide what to ask, how to interpret results, and when a run should continue. The executor should decide how a prompt reaches a concrete provider client and which provider-specific transport machinery is used. That separation is what keeps provider choice from infecting workflow logic.

A minimal example shows the pattern without drowning it in setup details.

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.prompt.executor.llms.MultiLLMPromptExecutor
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
 val executor = MultiLLMPromptExecutor(
  llmClients = mapOf(
   /* provider -> client bindings */
  )
 )

 val agent = AIAgent<String, String>(
  promptExecutor = executor,
  llmModel = /* concrete LLModel */
 )

 val result = agent.run(
  "Write a concise incident summary for on-call handoff."
 )

 println(result)
 executor.close()
}
```

This code is intentionally ordinary. That is the point. The agent does not know whether the chosen model is served by OpenAI, Anthropic,

Google, DeepSeek, OpenRouter, Bedrock, Ollama, or another backend exposed through a provider client. The model-facing routing lives in the executor. The agent holds a `PromptExecutor` and a model binding, then runs.

That design buys you more than convenience. It establishes a disciplined split between three concerns that teams frequently collapse into one word, "client".

Provider, model, executor You need a sharper vocabulary here because architectural mistakes usually begin as naming mistakes.

Provider. A provider is the external or local system that actually serves model inference. It brings authentication requirements, transport conventions, endpoint behavior, rate limits, and capability constraints. OpenAI, Anthropic, Google, OpenRouter, Bedrock, and Ollama are different providers even when two of them expose models with similar semantic behavior.

Model. A model is the concrete capability target used for a run. It has a name, a provider association, and behavior-relevant properties such as context length, structured output quality, streaming support, tool-call quality, cost, and latency profile.

Executor. A prompt executor is Koog's provider abstraction layer. It accepts a prompt and a selected model, chooses the right client path, issues the request, and returns a normalized assistant message into the runtime.

If you keep those roles distinct, the architecture stays modular. If you conflate them, you end up hardcoding vendor-specific assumptions in the very place where you most want portability: agent logic.

The execution contract boundary The `PromptExecutor` is not just a convenience wrapper around HTTP calls. It is the runtime contract for model execution. Its verified core operation has the shape:

```
execute(
 prompt: Prompt,
 model: LLModel,
 tools: List<ToolDescriptor> = emptyList()
): Message.Assistant
```

That signature is revealing. The executor accepts a prompt, not raw strings scattered through the codebase. It accepts a model, not a fixed provider endpoint baked into the agent. It accepts tool descriptors optionally, which means the abstraction is designed to remain the execution boundary even when the runtime becomes more capable than plain text completion.

Two design consequences follow immediately.

First, provider choice belongs to the executor and underlying client layer, not to your business logic. If application code is branching on vendor names before every run, the abstraction has already failed.

Second, model selection is still explicit. Koog does not erase model identity behind a fake universal interface. You still bind a concrete `LLModel` because model capabilities matter. The executor decouples provider plumbing, not model differences.

Single-provider, multi-provider, and routed execution Koog documents three executor families, and each one corresponds to a different architectural need.

SingleLLMPromptExecutor. Use this when you have exactly one provider client and portability is not yet a requirement. The abstraction still matters because it keeps agent code independent from raw client calls, but the topology is simple: one provider path, one client, one execution route.

MultiLLMPromptExecutor. Use this when you want multiple providers available behind one execution boundary. The executor

receives a map of providers to clients and dispatches according to the provider associated with the requested model. This is the common shape when you want one application to be able to run some tasks on a hosted vendor and other tasks on a local or alternate backend.

RoutingLLMPromptExecutor. Use this when you need multiple client instances for at least one provider and want routing across them. This moves beyond simple provider selection into a richer topology for throughput shaping, rate-limit avoidance, or specialized client pools. Koog marks routing APIs as experimental, so you should treat this executor family as version-sensitive and operationally higher risk than the single or multi-provider variants.

The progression here is important. You do not start with routing just because it sounds flexible. You start with the smallest executor topology that isolates provider plumbing without overfitting the current environment.

A builder can choose topology for you Koog's prompt executor builder encodes a practical heuristic that is worth understanding even if you eventually construct executors directly. When each provider appears once, the builder can produce a `MultiLLMPromptExecutor`. When any provider has multiple registered clients, the builder can promote the topology to a `RoutingLLMPromptExecutor`. That means executor shape is not only a class hierarchy concern; it is a consequence of how many clients you register and for which providers.

This is a useful design signal. It tells you that provider abstraction is not just about syntax. It is about describing the available transport and routing surface in a way the runtime can reason about. If your executor topology changes because you added a second client for the same provider, that is not an incidental implementation detail. It changes how requests may be dispatched and how failures may be diagnosed.

Why model binding remains explicit A common misconception

about provider abstractions is that they should hide model identity completely. That sounds attractive until the first portability bug. Different models vary in context handling, structured output quality, instruction following, and support for capabilities such as streaming or tool-oriented interactions. If you erase those differences too aggressively, you create an interface that looks universal but silently depends on the strongest vendor-specific behavior in your test environment.

Koog does not make that mistake. You pass a concrete `LLModel` for execution, and the executor uses the model's provider association to dispatch correctly. This preserves two essential truths at once:

- provider transport can be abstracted behind the executor,
- model capability differences still need to remain visible.

That is the right level of abstraction. You want to avoid scattering provider authentication and endpoint logic through your code, but you do not want to pretend that all models behave identically.

What can be normalized Prompt executors are valuable because they normalize a substantial amount of runtime behavior.

Transport invocation. Agent code does not need to know how each provider expects requests to be serialized or authenticated.

Execution entry point. A single execution contract can drive one or many provider clients.

Message return type. The executor returns a normalized assistant message into Koog's runtime rather than forcing each agent path to parse vendor-specific payloads.

Capability attachment surface. Optional tool descriptors and related execution features enter through the same abstraction boundary rather than through provider-specific side channels.

Resource lifecycle. Executors are closeable, which lets you treat them as managed infrastructure objects whether they wrap one client or many.

Normalization at this layer is what lets agent code focus on runtime semantics. It also improves testability: you can reason about prompt execution as one contract instead of as a pile of vendor adapters called from arbitrary places.

What should remain visible A sound abstraction hides transport details without erasing operationally important differences. Several concerns should remain visible to the calling design even when the executor is doing its job.

Context limits. Different models tolerate different prompt sizes and interaction histories. An executor can dispatch correctly, but it cannot make an undersized context window behave like a large one.

Structured-output fidelity. Some models are better than others at consistent schema-constrained responses. That matters if later runtime stages depend on structured interpretation.

Streaming behavior. Streaming exists on the executor surface, but models and providers differ in chunk quality, latency to first token, and feature parity.

Tool-call behavior. Some backends support richer tool-oriented interaction patterns than others. The abstraction should expose a unified API surface where possible, but you still need to choose models with appropriate capability fit.

Latency and cost. The executor can hide the transport path. It cannot make a slow, expensive model behave like a cheap, low-latency one.

This is where teams sometimes over-abstract. They build a single "ModelService" that promises every model does everything equally well.

The result is usually a portability story that works only on slides. Good abstractions isolate plumbing; they do not deny reality.

A durable binding pattern A practical binding pattern is to construct the executor as long-lived infrastructure and keep the model choice explicit at the agent boundary. That preserves two forms of flexibility.

First, you can swap the provider client topology without rewriting agent logic. Moving from a single provider to multiple providers, or from one hosted environment to a mixed hosted-local environment, becomes an executor concern.

Second, you can change the selected model for an agent or a family of agents without forcing the rest of the runtime to learn provider-specific details. The model remains a first-class configuration decision, but the transport remains encapsulated.

This pattern also gives you a place to centralize credentials, connection reuse, observability hooks, and fallback configuration rather than rebuilding those concerns in every agent factory.

Why vendor names do not belong in agent logic One of the clearest signs of abstraction failure is business logic that knows too much about providers. If you find code like "if provider is X, change the prompt" or "if vendor is Y, parse the response differently," you have already allowed provider semantics to leak upward.

Sometimes a limited amount of model-specific specialization is justified. For example, you may tune prompt wording for a model family or choose one model for summarization and another for extraction. That is not the same thing as scattering provider conditionals throughout application code. Keep specialization at deliberate configuration points. Do not distribute it through workflow branches, output parsing routines, or domain services.

A strong discipline is to bind provider-specific concerns in exactly two places:

1. client and executor assembly,
2. explicit model selection policy.

Everywhere else, work in terms of prompts, models, agent inputs, and outputs.

Portability is a design constraint, not a retrofit Many teams only discover the need for abstraction when they are already trying to switch providers. By then, the hidden coupling has spread into prompt templates, environment wiring, retry code, and observability dashboards. Prompt executors are valuable because they force you to choose a clean boundary before that drift begins.

That does not mean you should prematurely optimize for every possible provider. Portability has a cost. Multiple clients mean more configuration, more test cases, and more operational variance. But even if you only have one provider today, using the executor boundary from the start gives you a controlled place to evolve. The difference between "abstracted from day one" and "retrofitted under incident pressure" is usually measured in outages, not elegance.

Provider families and deployment modes The provider abstraction matters just as much across deployment styles as across vendors. Hosted APIs, proxy aggregators such as OpenRouter, cloud-platform adapters such as Bedrock, and local model servers such as Ollama all impose different operational assumptions. Authentication may range from API keys to cloud credentials. Network behavior may range from internet round trips to local process hops. Availability and observability may differ sharply. Yet from the agent's perspective, these are still model execution backends behind a `PromptExecutor`.

This is precisely why the abstraction needs to exist at the runtime boundary rather than as a pile of helper functions. A local model backend is not merely "another endpoint." It may have different cold-start characteristics, different output stability, and different operational failure modes. The executor/client layer is where those concerns belong.

Model switching without business-logic churn If model choice is explicit and provider transport is encapsulated, you gain a clean upgrade path for ordinary changes:

- move a development workflow from a local model to a hosted one,
- split tasks across providers by capability fit,
- test a new model family on one agent without touching others,
- add a second client path for a provider that is hitting rate limits.

Those changes are still operationally significant, but they stop being invasive code rewrites. The agent continues to depend on the same execution contract. This matters later when resilience features such as fallbacks and rerouting enter the design. If you have already kept provider-specific assumptions localized, those features remain tractable.

Resource management and lifecycle Because `PromptExecutor` is closeable, you should treat it like any other infrastructure component with a lifecycle. Construct it once per intended scope, reuse it deliberately, and close it when the scope ends. For a command-line program or integration test, explicit closure in the same function is usually enough. For a service, tie executor lifetime to application startup and shutdown rather than to individual requests.

That lifecycle discipline becomes more important as executor topology becomes richer. A routed executor may wrap multiple clients with different network and credential characteristics. Closing those resources

predictably is not optional hygiene; it is part of running the system cleanly.

Compatibility-sensitive areas Provider integrations evolve quickly. Model names change, capabilities shift, builders gain convenience methods, and experimental routing APIs may move before they stabilize. The stable architectural lesson is not any one constructor shape. The stable lesson is where provider binding belongs.

Use current, documented client classes and builders when writing real code. Keep the abstraction boundary fixed even if the exact ergonomics change across releases. If a builder adds new helpers for `openAI(...)`, `ollama(...)`, or fallback configuration, those are valuable conveniences. They should not change the deeper design: assemble provider clients in the executor layer, select concrete models explicitly, and keep agent logic free of provider plumbing.

Common anti-patterns Several mistakes recur when teams approach portability without a clear abstraction strategy.

Baking vendor names into prompts or business logic. If the prompt says "when using Provider X" or the workflow branches on specific vendor identifiers, you are making portability harder than it needs to be.

Hiding capability differences behind a false universal interface. You can normalize invocation, but you should not pretend that every model has equal context tolerance, structured-output quality, or tool-call behavior.

Treating model choice as a transport detail. A model is not just an endpoint string. It is a capability decision with runtime consequences.

Recreating executors per run. That throws away the benefits of centralized transport configuration and resource reuse.

Designing for one provider and calling it abstract. If all prompts, tests, and response expectations quietly assume one backend's behavior, the presence of an executor type alone does not make the system portable.

The clean design is straightforward even if the ecosystem around it is not. Let the agent own execution semantics. Let the model remain an explicit capability choice. Let the prompt executor own the binding from runtime intent to provider-specific transport. When those boundaries hold, you can change providers, mix local and hosted backends, and introduce richer routing policies without rewriting the logic that actually defines your agent's behavior.

3.4. Designing for Model Switching, Rerouting, and Resilience

A provider abstraction is useful only when it survives stress. The easy case is a stable environment with one acceptable model, comfortable latency, and no surprises from quotas or outages. Real systems do not stay in that state for long. A model that looked cost-effective during evaluation may become too expensive at production volume. A provider that was fast in one region may degrade during a traffic spike. A model that performs well for summarization may fail badly on extraction or schema-constrained output. If your architecture answers every one of those situations with "just switch models," then you do not yet have a switching strategy. You have a hope.

Koog gives you the execution boundary needed for switching, but the framework cannot choose your policy for you. That policy has to answer three operational questions clearly:

- When should a run stay on its primary model?

- When should it fail over to a fallback?
- When should it be rerouted proactively to a different path?

Those decisions belong at the provider boundary. They are not general workflow recovery, and they are not full application reliability design. The focus here is narrower and more practical: preserve agent continuity when the problem is model choice, provider health, or execution-path suitability.

A compact example makes the boundary concrete. The code deliberately stays close to verified API shapes and avoids speculative routing helpers. The important pattern is that you construct the executor topology separately from agent logic and keep the model choice explicit.

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.prompt.executor.llms.MultiLLMPromptExecutor
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
 val executor = MultiLLMPromptExecutor(
 llmClients = mapOf(
 /* provider -> client bindings */
 )
 /* optional fallback settings may be configured here */
 )

 val primaryAgent = AIAgent<String, String>(
 promptExecutor = executor,
 llmModel = /* primary LLModel */
 )

 val result = primaryAgent.run(
 "Summarize this incident and identify the likely " +
 "operational risk."
 )

 println(result)
 executor.close()
}
```

This example already encodes a resilience principle: the agent does not hardwire one provider client into its business logic. The executor

owns the provider map, and the model binding remains explicit. That is the minimum structure required for switching decisions to remain localized.

Three switching modes

You should separate three different operational patterns because they solve different problems and carry different risks.

Proactive substitution. You intentionally choose a different model before execution begins. Typical reasons include cost reduction, specialization for a task class, regional placement, or a lower latency target. Nothing is failing yet. You are making a policy decision up front.

Reactive failover. The primary execution path is unavailable or degraded enough that you need another model or provider to keep the run alive. This is the classic fallback case: outage, rate limiting, transport failure, or another provider-bound execution failure.

Conditional rerouting. You route a request to a different path based on request characteristics rather than on a failure of the primary model. This is a policy-controlled dispatch problem: perhaps long-context extraction goes one way, low-latency summarization another, and schema-heavy tasks to a model with better structured-output behavior.

These modes sound similar when discussed casually, but they are operationally distinct. Proactive substitution is about planned optimization. Reactive failover is about continuity under failure. Conditional rerouting is about matching request classes to execution paths. If you mix them together under one vague concept of "fallback," you will build confusing policies and poor observability.

Why switching is harder than swapping a model name

Teams often assume that if prompt executors abstract providers,

changing models is mostly a configuration edit. That is only partially true. The executor isolates transport plumbing, but several compatibility questions remain:

- Does the substitute model interpret the same prompt structure well?
- Does it preserve the output contract your downstream logic expects?
- Does it handle the same context size and message ordering safely?
- Does it support the same execution features, such as streaming or tool-oriented behavior?
- Does it fail differently under load, producing timeouts, refusals, truncation, or malformed structure?

The practical lesson is that switching is not a transport-only concern. It is a semantic substitution problem. The executor lets you move the request to another provider. Your policy must ensure that the substitute remains compatible enough with the task and the runtime expectations.

Design a smallest viable portable contract

The cleanest way to make model switching survivable is to define a smallest viable portable contract for the runs you expect to reroute. That contract should be narrower than the full set of features offered by your strongest provider.

For many systems, the contract includes the following:

- a prompt shape that does not depend on one vendor's quirks,

- an output structure that can be validated independently,
- bounded assumptions about context size,
- a clear distinction between required and optional model features,
- an explicit terminal behavior when the substitute cannot satisfy the requested capability.

This is the same discipline used in distributed systems when you define a portable wire contract instead of coding to one server's current implementation accident. Here the wire contract is prompt-plus-output behavior rather than JSON alone.

If you design every run around the richest model you currently have, you may still be able to switch later, but only by accepting silent behavior drift. That is a poor resilience strategy because it moves failure from transport into semantics, where it is harder to detect.

Fallback is narrower than many teams assume

Koog's documented executor surface makes one important boundary explicit: fallback support is documented for `execute()` and `executeMultipleChoices()`, but not for streaming execution. This matters operationally because many teams assume streaming can fail over as transparently as ordinary request-response execution. You should not make that assumption.

If a streaming path is critical, you need a different plan. Typical options are:

- pre-run rerouting before the stream starts,
- abort-and-restart semantics when the primary fails,
- or a deliberate downgrade to non-streaming fallback behavior.

That design constraint belongs in your policy from the beginning. Otherwise you will discover under pressure that your "resilient" streaming system only handles resilience for the non-streaming path.

Routing introduces power and instability

Koog documents `RoutingLLMPromptExecutor` for cases where you register multiple clients for at least one provider and want routing behavior such as throughput balancing, rate-limit avoidance, or failover across client instances. That is a useful capability, but it is also marked experimental and requires opting in with `@OptIn(ExperimentalRoutingApi::class)`. Treat that signal seriously.

An experimental routing API can be the right choice when your operational need is immediate and you accept version-sensitive maintenance. It is the wrong choice if you are trying to establish a calm, conservative baseline for a production platform with a low tolerance for upgrade churn. The main architectural lesson is not "always route." It is "contain routing policy at the executor boundary so that its complexity does not spread into agent logic."

Choosing primary, fallback, and rerouting criteria

Good switching policy starts with explicit criteria rather than folklore. A useful decision matrix usually considers at least four dimensions:

- **Latency:** How quickly does the path respond under your real workload and region?
- **Cost:** Is the model affordable for the expected request volume and token profile?
- **Capability fit:** Does the model handle the task class, structure, and context needs reliably?

- **Failure tolerance:** How much degradation can you accept before switching paths?

The primary model is the one that best fits the common case. The fallback model is the one that preserves enough correctness under failure to keep the service usable. The rerouting policy is the logic that decides whether a request belongs on the primary path at all.

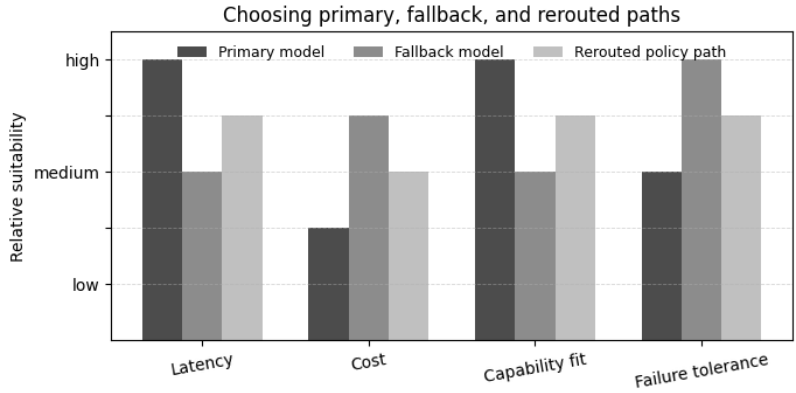


Figure 3.1: *A practical scoring view for substitution policy. The primary path usually maximizes capability fit, the fallback path emphasizes continuity under failure, and rerouted paths balance fit and operating cost for specific request classes.*

Interpreting the trade-offs

The chart is qualitative on purpose. You are not trying to predict one universal winner. You are making the criteria visible so that policy can be discussed and tested.

A strong primary path often scores highest on capability fit and acceptable latency, even if it is not the cheapest option. A fallback path often sacrifices some quality or richness in exchange for robustness and

availability under stress. A rerouted path typically exists because one request class has a different optimum than the general case. For example, a long-context analysis route may cost more but avoid truncation risk; a low-latency customer-facing response path may favor a faster model with slightly lower reasoning depth.

This framing prevents a common failure mode: choosing the fallback solely by price. Cheap models are attractive, but a fallback that cannot preserve the output contract or task semantics is not resilient. It is just a silent downgrade.

Context-preserving substitution

The hardest switching problem is not moving a fresh request to a different model. It is preserving continuity for a run that already has prompt state, intermediate conclusions, or conversation history. If the substitute model cannot interpret that state in the same way, your reroute effectively becomes a semantic restart.

You reduce this risk by keeping intermediate state portable:

- prefer explicit structured context over implicit prompt lore,
- keep generated intermediate artifacts concise and typed where possible,
- avoid provider-specific formatting conventions in the prompt,
- validate that substitute models handle the same state shape,
- keep the restart boundary explicit when semantic continuity cannot be preserved.

This is particularly important for multi-step runs. If the first model produced intermediate reasoning that the second model cannot reliably continue from, do not pretend the run remained seamless. Mark

it as a controlled restart or degraded continuation in your runtime semantics and telemetry.

Switching by request class

Conditional rerouting is often safer than reactive failover because you can make the decision before entering a fragile path. A simple policy might direct short, latency-sensitive drafting tasks to one model, long-context extraction to another, and cost-sensitive background batch work to a cheaper path. The key is to base the policy on observable request attributes rather than on hidden intuition.

Good request-class features include:

- approximate prompt size,
- need for structured or schema-constrained output,
- tolerance for latency,
- user-facing versus background execution,
- importance of deterministic formatting,
- expected follow-on execution features such as streaming.

Notice that this remains a provider-boundary decision, not application business logic. The domain service should describe the nature of the request. The executor policy should decide which model path best serves it.

Failover without hidden quality collapse

Reactive failover is where teams often overestimate their resilience. The system reconnects, requests succeed again, and everyone assumes the problem is solved. But from the user's perspective, the real question is whether the fallback path preserved sufficient behavior. A fall-

back that returns structurally different output, ignores key instructions, or mishandles context may keep uptime graphs green while quietly breaking the product.

You should therefore define explicit failover expectations:

- which task classes may fall back automatically,
- what level of quality degradation is acceptable,
- whether the output contract remains unchanged,
- how the system records that a fallback path was used,
- and when the correct answer is to fail closed rather than silently continue.

Not every task should fail over. High-stakes actions, legal analysis, financial interpretation, or any task whose correctness envelope is narrow may require stricter substitution rules than casual summarization.

Observability is part of resilience

Switching policy without observability becomes superstition. You need to know not only that a request succeeded, but also which path served it, why that path was chosen, whether fallback occurred, and what quality trade-off was accepted. Koog's tracing support is useful here because it can expose agent lifecycle, node execution, model calls, and error tagging across the run.

At minimum, instrument enough metadata to answer these questions during an incident:

- Which model was attempted first?
- Did the request reroute proactively or fail over reactively?

- Did the substitute path preserve the expected output contract?
- Was the request streaming or non-streaming?
- Did the runtime downgrade behavior to keep the run alive?

Without that visibility, fallback chains tend to accumulate silently until nobody can explain why a request was slow, cheap, malformed, or unexpectedly terse.

When a homogeneous setup is better

Provider diversity is not automatically superior. A single-provider setup is often simpler to test, easier to debug, and more predictable under load. You should favor homogeneity when the provider is operationally reliable enough, the model family covers your task envelope, and the cost or compliance profile is acceptable. Complexity is not resilience by itself.

Introduce multiple providers or richer routing only when you have a specific pressure that justifies the added variance: rate-limit pressure, regional constraints, workload specialization, or a real need for continuity across outages. If you cannot name the pressure clearly, you probably do not need the extra topology yet.

Anti-patterns that look like resilience

Several common designs create the appearance of flexibility while making failures harder to understand.

Silent fallback chains. If the runtime can descend through multiple models without recording that fact, you will ship quality regressions that look like random behavior drift.

Choosing the fallback only by price. A cheap path that cannot preserve semantics or output structure is an outage with better marketing.

Assuming streaming and non-streaming fail over the same way. Koog's documented fallback boundary does not support that assumption.

Mixing incompatible output expectations across providers. If one path returns a schema-like structure and another tends toward prose, do not pretend they are interchangeable without validation.

Treating routing as a free upgrade. Experimental routing can improve throughput and reduce provider hot spots, but it also complicates reproducibility, debugging, and version management.

Equating transport recovery with product recovery. A request that reached a provider is not necessarily a request that preserved user-visible correctness.

A practical policy shape

A disciplined switching policy usually reads more like an SRE runbook than like a model benchmark sheet:

1. Define the primary path per task class.
2. Define the substitute path only for classes where compatibility is validated.
3. Keep prompts and outputs within a smallest viable portable contract.
4. Distinguish proactive rerouting from reactive failover.
5. Record every substitution decision in traces and metrics.
6. Fail closed when the substitute cannot preserve the contract.

That approach keeps the provider boundary honest. The executor can switch paths, the agent can continue its run, and the application can

trust that a substitution either met policy or was rejected explicitly. Once those rules are in place, model independence stops being a slogan. It becomes an operational property you can test, observe, and rely on under real failure conditions.

3.5. Operating Streaming Runs Under Coroutine Concurrency

A non-streaming run hides most of its internal timing. The user waits, the model works, and the runtime eventually returns a terminal result. That makes many design errors invisible. You can block too long in the middle of a run, collect too much intermediate state, or ignore cancellation entirely, and the system may still appear correct in a simple test. Streaming removes that comfort. The moment you emit partial output, timing becomes part of the contract. Users see pauses, bursts, interruptions, and incomplete fragments. Upstream and downstream services see backpressure, cancellation, and ordering effects. The run is no longer just about *what* is produced. It is also about *when* each piece becomes observable.

Koog treats streaming as a typed runtime surface rather than as raw token callbacks. That choice matters because streaming is not merely a transport optimization. It changes execution semantics. Partial output is now visible before the run is complete, and visible state creates obligations: you need a clear end condition, disciplined cancellation, and a concurrency model that does not quietly reorder or corrupt the stream under load.

A compact collector illustrates the basic mechanics.

```
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.flow.collect

fun main() = runBlocking {
    val stream = executor.executeStreaming(
```

```
        prompt = prompt,
        model = model
    )

    stream.collect { frame ->
        when (frame) {
            is StreamFrame.Append -> print(frame.text)
            is StreamFrame.ToolCall -> {
                println()
                println("tool request: ${frame.name}")
            }
            is StreamFrame.End -> {
                println()
                println("finish: ${frame.finishReason}")
            }
        }
    }
}
```

The exact place where you obtain the stream depends on the executor you use, but the important runtime shape is stable: Koog streaming yields a `Flow<StreamFrame>`. You collect typed frames rather than unstructured chunks. That gives you a stronger contract than "append bytes until the socket closes."

Streaming changes the meaning of completion In a non-streaming run, the returned value and the final state usually appear at the same moment. In a streaming run, those two ideas separate. Partial content may already be visible to the user or a downstream consumer while the run is still active. That creates three distinct states you need to keep straight.

Observed content. This is what has already been emitted. Once it is sent to a terminal, browser, websocket, or log sink, you should treat it as externally visible.

Current runtime state. This is the agent's internal progress at the moment of emission. The runtime may still decide more text, a tool request, or a terminal frame is needed.

Terminal outcome. This is the authoritative end of the stream, represented by a typed terminal frame rather than by guesswork. Until you receive that terminal signal, you do not have a complete run.

If you collapse these states into one mental model, you will make two common mistakes. First, you may render partial output as if it were final truth. Second, you may ignore the need to record how the stream ended, which makes observability and recovery far harder than they should be.

Typed frames are a semantic boundary Koog documents three frame types: `StreamFrame.Append(text)`, `StreamFrame.ToolCall(id, name, content)`, and `StreamFrame.End(finishReason, metaInfo)`. These types are not cosmetic. They separate three very different events in the life of a streaming run.

Append means incremental language output. You can treat it as user-visible text in many interfaces, but you still should remember that it is provisional until the stream ends.

ToolCall means the stream has emitted an action-oriented event rather than plain language. Even without expanding into the broader tool execution patterns, the existence of this frame type tells you that streaming runs are not reducible to character accumulation. The stream can carry execution-relevant signals.

End marks the terminal boundary. This is where finish reason and metadata become part of the observable run state. If your application only prints appended text and ignores the terminal frame, it loses the only authoritative signal that the stream completed normally.

Once you treat these frames as semantically distinct, your collector logic becomes cleaner. Text rendering, tool-oriented handling, and

terminal accounting do not need to be improvised from one mutable string buffer.

Flow semantics matter more than most teams expect Kotlin `Flow` is sequential by default. That one fact explains a large amount of streaming behavior. When you collect a flow, each emitted element is handled in order unless you explicitly introduce concurrency through operators such as `buffer`, `flatMapMerge`, or other parallelizing transformations. A plain collector is therefore an ordered, sequential consumer of stream frames.

That default is safer than many callback-based streaming APIs because it preserves a straightforward mental model:

1. the producer emits a frame,
2. the collector processes that frame,
3. the next frame proceeds afterward unless buffering or another concurrency operator changes the behavior.

This means you should assume ordering is stable only as long as you keep the pipeline sequential. The moment you introduce buffering, parallel downstream work, or shared mutable state across collectors, you need to reason about visibility and race conditions explicitly.

A simple collector often does exactly what you want:

```
val collected = StringBuilder()

stream.collect { frame ->
    when (frame) {
        is StreamFrame.Append -> collected.append(frame.text)
        is StreamFrame.ToolCall -> { /* handle separately */ }
        is StreamFrame.End -> println(collected.toString())
    }
}
```

The logic is predictable because collection is sequential. That predictability disappears if you start pushing frames into multiple concurrent sinks without a clear design for ordering.

Partial output is not committed truth Users tend to read streamed text as progress toward the final answer. Systems designers sometimes make the more dangerous mistake of reading it as already committed state. That is not the same thing. Streamed text is a visible hypothesis until the terminal boundary confirms how the run finished.

This distinction is especially important when output drives application behavior. For a chat UI, speculative partial rendering is usually acceptable as long as the user understands the answer is still forming. For a workflow engine, turning partial stream fragments into durable state may be incorrect. If a run is canceled, times out, or transitions into a tool-oriented step, the partially emitted text may not represent the run's final intent.

A useful rule is to separate *display state* from *committed state*. Display state can update on every append frame. Committed state should update only when the runtime reaches the terminal condition appropriate for your application.

Cancellation is part of the stream contract Streaming makes cancellation visible and operationally important. A user closes a browser tab, a client disconnects, a deadline expires, or an upstream service withdraws interest in the answer. If your streaming collector continues happily appending to a dead sink, you are paying for work that no one can consume.

Kotlin's cancellation model is cooperative and built into structured concurrency. A streaming collection should therefore run inside a man-

aged coroutine scope tied to the request, session, or agent lifecycle. When the parent scope is canceled, collection should stop, and the runtime should release the work associated with the stream.

A disciplined pattern looks like this:

```
import kotlinx.coroutines.withTimeout
import kotlinx.coroutines.CancellationException

suspend fun streamToClient() {
    try {
        withTimeout(10_000) {
            stream.collect { frame ->
                emitFrameToClient(frame)
            }
        }
    } catch (e: CancellationException) {
        cleanupStreamState()
        throw e
    }
}
```

The goal is not just to stop the collector. The goal is to make cancellation part of the normal operational path. If cancellation is treated as an exceptional afterthought, streams will leak work, continue writing into closed channels, or leave traces that suggest the run hung mysteriously.

Blocking collectors break responsiveness One of the easiest ways to damage a streaming system is to perform blocking work directly inside the collector. The default sequential flow semantics mean each frame waits for the collector to finish processing the previous one. If the collector performs slow disk I/O, synchronous network calls, heavy parsing, or blocking tool coordination inline, you have created backpressure whether you intended to or not.

That may be acceptable when deliberate. It is harmful when accidental. For example, appending text to an in-memory buffer is cheap. Writing every fragment synchronously to a remote datastore is not. If you place that write directly in the collector, latency to the user now includes the

storage path, and cancellation becomes sluggish because the collector is repeatedly trapped in blocking operations.

You have several options:

- keep the collector lightweight and move heavy work elsewhere,
- switch blocking sinks to interruption-aware wrappers,
- isolate expensive operations onto an appropriate coroutine context,
- or batch updates so that every frame is not treated as a full persistence event.

The right choice depends on the product requirement. The important point is that streaming exposes any mismatch between your desired responsiveness and your collector's actual work.

Backpressure is now user-visible In a non-streaming call, queueing and slowdown appear mostly as increased end-to-end latency. In a streaming run, the user often sees the slowdown directly: bursts of text, long pauses, or a stream that appears to freeze before finally completing. Backpressure can arise at several layers:

- the provider emits slowly,
- the network path delays frame delivery,
- the collector processes frames too slowly,
- the downstream sink cannot accept updates quickly enough.

Because `Flow` is sequential by default, slow collection naturally slows visible progress. This can actually be a useful property because it propagates pressure honestly. The danger appears when you try to "solve"

the slowdown by adding uncontrolled buffering everywhere. Buffers can smooth bursts, but they also decouple producer and consumer timing, which can increase memory usage and make cancellation less immediate.

Use buffering deliberately. If you add `buffer()`, do it because you understand which stage needs elasticity and how much reordering or delay your application can tolerate. Do not add it merely because the stream feels slow in a superficial test.

Streaming and concurrency are not the same thing Teams often say "we are using coroutines, so streaming is concurrent." That is imprecise. A coroutine-based streaming collector may still be strictly sequential. Concurrency only appears where you introduce it: separate coroutines, buffered stages, merged flows, or asynchronous sinks. This is a feature, not a limitation. It lets you begin from deterministic ordered behavior and add concurrency only where the operational case justifies it.

That distinction matters when you reason about correctness. A sequential collector is easy to test and reason about. A collector that emits to multiple sinks concurrently may improve throughput, but it also creates questions about atomicity, partial failure, and ordering. If one sink fails and another succeeds, what state should the run record? If one consumer lags, should the others continue? Those are design decisions, not coroutines automatically making things faster.

A safe streaming flow for request-bound delivery A good baseline for a request-bound stream has four steps:

1. create the stream inside the request's coroutine scope,
2. collect frames sequentially,

3. keep per-frame work lightweight and non-blocking where possible,

4. record the terminal state explicitly when the end frame arrives.

In code, that often means one collector path for delivery and one small piece of terminal accounting:

```
suspend fun handleStream() {
    val buffer = StringBuilder()

    stream.collect { frame ->
        when (frame) {
            is StreamFrame.Append -> {
                buffer.append(frame.text)
                sendText(frame.text)
            }
            is StreamFrame.ToolCall -> {
                recordToolCall(frame.id, frame.name)
            }
            is StreamFrame.End -> {
                recordTerminalState(
                    finishReason = frame.finishReason,
                    finalText = buffer.toString()
                )
            }
        }
    }
}
```

This remains intentionally conservative. It preserves order, avoids concurrency surprises, and gives you an explicit terminal record.

Tool-oriented frames change what the stream can mean Even without expanding into full parallel tool orchestration, `StreamFrame.ToolCall` changes the semantics of a streaming run. It means the stream can carry more than presentation data. The runtime may emit a typed request for a controlled action while the run is still unfolding. That in turn means your collector cannot always assume that every frame should be appended to the user-visible transcript.

This is one reason typed frames are so important. A raw chunk stream would force you to recover semantics from text conventions. Koog instead places tool-oriented events on an explicit channel within the same streaming surface. That keeps the runtime honest: presentation and execution signals are both first-class, but they are not confused with one another.

The detailed mechanics of tool execution belong elsewhere. The operational takeaway here is simpler: a streaming collector should be written against frame types, not against the assumption that everything is printable text.

Timeouts and abort boundaries Timeout handling becomes subtler once partial output exists. A timeout no longer means only that "no answer was returned." It may mean that some content was already exposed before the run exceeded its budget. Your runtime therefore needs a clear abort policy.

For user-facing interfaces, you may choose to keep the partial text visible and mark the run as interrupted. For API consumers, you may prefer to emit a terminal error state and instruct the caller not to treat the partial output as committed. For internal pipelines, you may discard partial text entirely unless an end frame confirms completion. The right choice depends on product semantics, but the important thing is to choose intentionally.

Do not let timeouts create ambiguous half-finished state. A terminal record should say whether the run ended normally, was canceled, or timed out after partial emission.

Observability is richer for streams A streamed run has a larger observability footprint than a non-streamed one. That is not overhead for its own sake. It is how you reconstruct what the system actually did.

At minimum, capture:

- stream start time,
- time to first append frame,
- number of append frames,
- whether tool-call frames were emitted,
- terminal finish reason,
- whether cancellation or timeout occurred,
- and whether downstream delivery lagged behind frame production.

These measurements help you distinguish model latency from sink latency, provider stalls from collector stalls, and clean completion from silent truncation. Without them, two very different failures can look identical to the operator: "the answer was incomplete."

Common anti-patterns Several streaming bugs recur because they work fine in trivial demos.

Appending to shared mutable state from multiple coroutines. Sequential collection is safe. Concurrent mutation of a shared transcript without discipline is not.

Treating every append frame as durable state. Visible progress is not the same thing as committed completion.

Ignoring the end frame. If you do not record terminal state, you cannot tell clean completion from interruption.

Blocking in the collector. This creates hidden backpressure, delayed cancellation, and poor user experience.

Swallowing cancellation exceptions. A canceled request should stop work and propagate cancellation semantics through the scope hierarchy.

Assuming streamed and non-streamed runs are operationally identical. They are not. Streaming exposes timing, partiality, and sink behavior as part of the run contract.

A disciplined mental model The safest way to think about a streamed run is as an ordered sequence of typed events collected within a managed coroutine scope. Frames become observable incrementally, but the run is not complete until the terminal boundary is reached. Collection is sequential unless you intentionally introduce concurrency. Cancellation is cooperative and should be wired into the request or agent lifetime. Backpressure is not an implementation detail to wish away; it is part of what the user experiences.

Once you adopt that model, the implementation choices become much clearer. Keep collectors lightweight. Separate visible progress from committed state. Treat the end frame as authoritative. Introduce concurrency only where it solves a real problem and where you can still explain ordering and failure semantics afterward. That is how a streaming agent remains responsive without becoming nondeterministic.

Chapter 4

Tools, External Actions, and Service Boundaries

The most dangerous moment in agentic development is not when the model says something wrong, but when it is allowed to do something wrong. This chapter teaches readers how to design stable tool contracts, connect them to real services, coordinate concurrent actions, adopt protocol-based interoperability where it pays off, and preserve sane JVM team boundaries when Kotlin and Java coexist around the tool layer.

4.1. Designing Custom Tools as Stable Agent Interfaces

The agent stops being merely suggestive when it can cross a service boundary. A generated paragraph can be ignored, edited, or discarded. A generated tool call can send an email, place an order, modify a record,

or trigger a workflow that outlives the current conversation. That shift changes your design problem. You are no longer shaping prompts alone; you are defining a capability surface that must remain governable under ambiguity, retries, partial failure, and operational pressure.

A tool is therefore not just "a function the model can call." In Koog, a tool is an explicit, typed contract that the runtime can expose, validate, execute, and observe. The design quality of that contract determines whether the agent behaves like a bounded system component or like an unreviewed scripting surface.

A narrow, domain-shaped tool makes the difference immediately. Compare a generic action hook with a tool that states its intent, arguments, and result shape directly:

```
import ai.koog.agents.core.tools.Tool
import kotlinx.serialization.Serializable
import kotlin.reflect.typeOf

@Serializable
data class CancelOrderArgs(
 val orderId: String,
 val reason: String
)

@Serializable
data class CancelOrderResult(
 val orderId: String,
 val status: String,
 val cancellationAccepted: Boolean
)

class CancelOrderTool : Tool<CancelOrderArgs, CancelOrderResult>(
 argsType = typeOf<CancelOrderArgs>(),
 resultType = typeOf<CancelOrderResult>(),
 name = "cancel_order",
 description = "Cancel an order that is still eligible for " +
 "cancellation."
) {
 override suspend fun execute(
 args: CancelOrderArgs
 ): CancelOrderResult {
 require(args.orderId.isNotBlank()) {
 "orderId must not be blank"
```

```
  }
  require(args.reason.isNotBlank()) {
  "reason must not be blank"
  }

  return CancelOrderResult(
  orderId = args.orderId,
  status = "CANCELLED",
  cancellationAccepted = true
  )
  }
}
```

The code is simple, but the important part is not the syntax. The contract says what operation is being offered, what inputs are legal, and what the caller can expect back. That makes the runtime boundary explicit. The model no longer has a vague ability to "do something with orders." It has one named capability with visible semantics.

What deserves tool status

Teams often place executable behavior in three locations: the prompt, orchestration glue, or a first-class tool. Those are not interchangeable.

Prompt logic is appropriate when you are shaping interpretation, style, extraction rules, or decision guidance that remains purely cognitive. You can instruct the model to summarize a ticket, infer sentiment, draft a reply, or choose among already available options. None of that requires a tool if no external action occurs and no separately governed capability is needed.

Orchestration glue is appropriate when the runtime must sequence internal steps that the model should not directly control. That includes bookkeeping, caching, tracing enrichment, or deterministic branches that exist to keep the system maintainable rather than to advertise a business capability.

A first-class tool is appropriate when the capability has domain meaning and must stand on its own as a controlled action. Use a tool when

at least one of the following is true:

- the action reads or mutates external state,
- the action needs argument validation beyond prompt wording,
- the action can fail independently of model generation,
- the action requires authorization, policy checks, or audit,
- the action should be observable as a discrete unit,
- the action must remain stable across prompt revisions,
- the action carries latency, cost, or retry implications.

That decision rule prevents a common failure mode: using prompts to compensate for missing interface design. If the system prompt says "only cancel refundable orders after checking policy," but the only exposed capability is `executeAction(action, payload)`, you have not created a reliable control surface. You have created a hope-based one.

A tool is a capability boundary

You get better results when you treat the tool contract as the allowed question the agent may ask the rest of the system. That question must be narrow enough to validate, meaningful enough to audit, and stable enough to evolve without rewriting all surrounding logic.

A well-designed tool boundary answers four things clearly:

1. What the agent is allowed to ask for.
2. What inputs are valid.
3. What outputs are guaranteed.
4. What side effects are permitted.

That framing is more useful than thinking in terms of wrappers around existing functions. Existing functions reflect implementation convenience. Tools should reflect governed capabilities. A repository method like `updateStatus(id, value)` may be useful internally, but a tool such as `approve_invoice` or `cancel_order` is usually the safer public boundary because it names the business intent instead of leaking a storage-level mutation primitive.

This is where schema shape starts to matter. If the tool accepts free-form JSON, a bag of strings, or a generic map, the model can form requests that are syntactically valid but semantically weak. If the tool accepts enumerated values, constrained fields, and purpose-specific arguments, the runtime gains leverage. Validation becomes crisp, failure modes become legible, and policy enforcement has somewhere concrete to attach.

From generic hooks to domain verbs

The fastest way to see the design pressure is to start with a bad interface.

```
import ai.koog.agents.core.tools.Tool
import kotlinx.serialization.Serializable
import kotlin.reflect.typeOf

@Serializable
data class ExecuteActionArgs(
 val action: String,
 val payload: Map<String, String>
)

@Serializable
data class ExecuteActionResult(
 val success: Boolean,
 val message: String
)

class ExecuteActionTool : Tool<
 ExecuteActionArgs,
 ExecuteActionResult
>(
 argsType = typeOf<ExecuteActionArgs>(),
```

```
 resultType = typeOf<ExecuteActionResult>(),
 name = "execute_action",
 description = "Execute a requested system action."
) {
 override suspend fun execute(
 args: ExecuteActionArgs
 ): ExecuteActionResult {
 return ExecuteActionResult(
 success = true,
 message = "Action executed"
 )
 }
}
```

This tool is structurally typed but semantically unbounded. The field `action` is a hidden second dispatch layer. The payload is untyped. Validation depends on string conventions that the model must infer. Observability becomes poor because traces show one tool name covering many unrelated behaviors. Authorization becomes awkward because the policy surface is collapsed into one escape hatch. Testing becomes broad and fragile because every new action extends a polymorphic command language without changing the interface.

A better design splits capabilities by business intent:

- `get_order_status(orderId)`
- `cancel_order(orderId, reason)`
- `refund_payment(paymentId, reasonCode)`
- `resend_invoice(invoiceId)`

These narrower tools look repetitive at first. In practice, they are cheaper to govern. Each one can carry tailored validation, clear timeout budgets, distinct authorization rules, and domain-specific result envelopes. The repetition is not duplication; it is explicitness.

Schema design is operational design

Tool schemas are often treated like serialization details. They are not. They encode which mistakes the runtime can reject before execution and which ambiguities are pushed into production behavior.

Narrow input domains are one of your strongest control mechanisms. If a field has only three legal values, make it an enum-like closed set in your data model. If an identifier has a stable format, validate it. If a field is optional only for one branch of behavior, that is often a sign the tool is trying to do too much.

You should prefer arguments that reflect business meaning over transport convenience. An argument named `customerId` is better than `pathSegment`. `reasonCode` is better than `metadata`. `effectiveDate` is better than `value2`. Models reason over names. Operators debug names. Auditors review names. Semantic naming reduces both model confusion and human ambiguity.

Outputs need the same discipline. Returning a single unstructured text blob is attractive because it is easy, but it weakens downstream control. A structured result gives the runtime and orchestration layer stable fields to inspect. It also lets you separate machine-relevant state from human-readable explanation.

```
import kotlinx.serialization.Serializable

@Serializable
data class RefundPaymentResult(
 val paymentId: String,
 val outcome: String,
 val refundReference: String? = null,
 val retryable: Boolean,
 val message: String
)
```

That envelope is much more useful than “Refund attempted.” The model can still read the message, but the runtime can also reason over `outcome`, and your orchestration can decide whether to continue, retry, or escalate without parsing prose.

Validation belongs at the boundary

You should reject invalid requests before side effects begin. That sounds obvious, but many tool implementations perform validation late, after remote calls have started or after internal state has already been queried in ways that carry cost and timing variance.

Boundary validation has three layers:

1. **Structural validation**: types, required fields, range checks, allowed values, and basic format checks.
2. **Semantic validation**: whether the request makes sense in domain terms, such as whether a cancellation reason is allowed for the requested operation.
3. **Contextual validation**: whether the action is permitted now, for this caller, against this resource, under current policy.

Only the first layer is purely about schema. The second and third layers usually require application logic and current state. Still, the tool boundary should own the responsibility to make those layers explicit. Do not hide them behind vague failures.

If a tool can fail because the order does not exist, because the order exists but is not cancellable, and because the caller lacks permission, the result model or error taxonomy should distinguish those cases. Otherwise, the model, the orchestration layer, and operators all receive the same opaque failure and react badly. One retries when it should stop. Another asks the user for information that was not the problem. A third files the incident under infrastructure noise instead of policy mismatch.

Bound side effects and idempotent intent

Every tool with side effects needs a clear statement of what it may

change and how repeated invocation behaves. This is where the interface becomes a control surface rather than a convenience wrapper.

Some tools are read-only. Their contract should state that they do not mutate state. Some tools initiate a mutation exactly once. Some are safe to repeat with the same business key. Some are not safe to repeat at all and therefore require stronger runtime coordination.

You should make idempotency expectations part of the tool design, not an implementation footnote. If a tool may be retried because of timeout, cancellation, or upstream uncertainty, define whether the same logical request can be submitted again without duplicating the business effect. Sometimes that means the downstream service owns an idempotency key. Sometimes it means the operation is naturally idempotent, such as setting a resource to a target state. Sometimes it means retries must be prohibited after the side effect boundary.

Even when the runtime does not expose a dedicated idempotency primitive at the tool definition level, you can still design the contract around stable business intent. “Cancel order `O-1042`” is easier to make safe under retry than “run cancellation workflow now.” The first names a target state on a known resource. The second names an implementation process whose repeatability is unclear.

Do not leak implementation details

A tool should not advertise how the system happens to work internally. That includes storage details, HTTP semantics, queue mechanics, and repository vocabulary unless those are genuinely part of the domain.

Avoid these naming patterns:

- `query_database`
- `invoke_endpoint`
- `run_sql`

- `patch_record`
- `post_crm_event`

They are implementation-oriented, overly broad, or both. The model does not need a database tool. It needs a domain capability such as `lookup_customer_account` or `record_shipping_exception`. The narrower name is not cosmetic. It determines the granularity of validation, permission checks, audit logs, and failure handling.

Leaking transport details also makes contract evolution harder. If the tool is named after an endpoint or table, infrastructure refactors become externally visible. A domain-named tool can survive a change from REST to gRPC, from one provider to another, or from synchronous execution to queued processing without changing the agent-visible capability.

Keep descriptors durable and metadata ephemeral

Koog's tool model distinguishes the tool definition from call-time execution context. That separation is useful. The descriptor should contain stable identity and meaning: name, description, argument shape, result shape, and any other static metadata that defines the capability. Request-scoped details such as correlation IDs, tenant context, deadlines, or tracing attributes belong at invocation time, not in the static contract.

That distinction keeps the interface durable. If you bake transient routing or caller-specific assumptions into the tool definition, the tool stops being a reusable capability and becomes an accidental binding to one deployment context. Static contract, dynamic execution context, and domain service implementation should remain separate concerns.

Register only intentional capabilities

A registry is not just a wiring artifact. It is the published menu of what

the model may invoke. Koog exposes tools through a `ToolRegistry`, which makes the publication step explicit.

```
import ai.koog.agents.core.tools.ToolRegistry

val orderTools = ToolRegistry {
 add(CancelOrderTool())
}
```

That explicit registry matters operationally. You can review it, version it, test it, and compare it across environments. A large share of avoidable agent risk comes from accidental exposure: helper methods turned into callable actions, debugging capabilities left enabled, or broad administrative operations registered because they were convenient during development. Treat the registry as an allowlist, not a bag of available classes.

Anti-patterns that look productive early

Several tool designs feel efficient during prototyping and become expensive later.

The universal tool. One tool accepts an action name and arbitrary payload. This compresses development effort at the cost of every control property that matters later.

The transport wrapper. A tool mirrors an HTTP client or repository API directly. The agent ends up steering infrastructure instead of requesting business operations.

The kitchen-sink tool. One tool validates input, performs lookup, checks authorization, mutates state, publishes an event, and returns a human summary. It is difficult to reason about, difficult to test, and nearly impossible to retry safely.

The prose-only result. The tool returns a sentence instead of a result model. Downstream logic must parse language or give up on deterministic handling.

The hidden side effect. A tool sounds like a query but mutates state as a side effect, such as logging a contact attempt, updating a cache that triggers downstream processing, or changing a status during lookup. The name and contract must reveal the effect zone.

The prompt-enforced contract. The interface is broad, and the team relies on instructions such as "only use this for read-only access." If the runtime cannot enforce the rule, it is not a boundary.

A practical invocation flow

Once the tool contract is shaped correctly, the execution path becomes easier to reason about.

The model proposes a tool call based on the capability names and descriptions it has been given. The runtime validates the supplied arguments against the declared schema and any immediate structural constraints. The tool executes within a bounded policy context: timeouts, authorization, side-effect rules, and service dependencies apply there, not in the prompt. The result returns in structured form. The orchestration layer then decides whether to continue, ask for clarification, retry, surface an error, or stop.

That flow is conceptually simple because the responsibilities are separated. The model selects among capabilities. The runtime validates. The tool performs one governed action. The orchestration decides what the result means for the broader task.

When those responsibilities blur, failure handling degrades quickly.

Failure modes worth designing for

Malformed arguments are the easiest case and should fail fast. The runtime or tool boundary should reject them deterministically and return a clear category of failure.

Ambiguous tool purpose is subtler. If the model cannot reliably distin-

guish between two overlapping capabilities, you will see tool flapping, unnecessary retries, and strange plans. That is usually an interface problem, not a prompt problem. Rename the tools or split the semantics more cleanly.

Hidden side effects are operationally dangerous because they destroy retry safety. If a tool appears read-only but triggers mutation, incident analysis becomes confused and rollback assumptions fail.

Weak validation turns model uncertainty into production variance. A free-form reason field may be fine for notes, but if downstream policy depends on a small set of allowed reasons, validate and encode them explicitly.

Permission mismatches should be expressed cleanly. A denied action is not the same as an invalid request, and it is not the same as a dependency outage. Treating all three as generic failure teaches the orchestration layer the wrong recovery behavior.

Timeouts deserve special care. A timeout does not tell you whether the side effect happened. That uncertainty should shape the contract, the downstream service design, and the retry policy. If the action is not safely repeatable, the tool must return enough structured state for the caller to decide whether to reconcile before retrying.

Version-sensitive parts of the boundary

The conceptual design principles are stable, but framework details can change. Tool-definition APIs, serialization conventions, schema generation helpers, and JVM-specific interop features should be treated as version-sensitive implementation details. Keep your contracts conservative enough that an internal library upgrade does not force a redesign of the agent-visible surface.

That is another reason to prefer domain-oriented arguments and results over framework-shaped ones. A stable DTO with explicit fields

survives serializer changes better than loosely typed maps or reflective magic. A capability named for business intent survives runtime evolution better than one named after a convenience method.

The strongest tool contracts look almost boring. They do one thing, use explicit schemas, reject bad input early, expose bounded effects, and return results that both the model and the runtime can interpret without guesswork. Under production load, that boring surface is exactly what keeps the agent governable.

4.2. Connecting Agents to APIs and Application Services

A clean tool contract is only half the job. The harder problem starts when you bind that contract to real systems that own money, inventory, identity, policy, or customer data. At that point, the temptation is strong: the tool already has typed arguments and a clear name, so why not call the database, hit an HTTP endpoint, or embed the business logic directly in the tool class? That shortcut makes early demos look efficient and production systems brittle. The agent boundary then absorbs transport concerns, retry policy, authorization checks, and persistence details that belong elsewhere.

You get a more durable design when the tool remains a boundary adapter and delegates domain work to an application service. In practice, that means the tool translates a validated agent request into a business use case, the service owns rules and coordination, and lower layers handle HTTP, persistence, and vendor-specific behavior.

A concrete example makes the layering visible. Suppose you expose a refund capability to the agent. The tool should not decide refund eligibility from raw tables, open an HTTP client itself, and construct provider-specific requests inline. Keep the tool thin and push policy

into a service that already models the business action.

```
import ai.koog.agents.core.tools.Tool
import kotlinx.serialization.Serializable
import kotlin.reflect.typeOf

@Serializable
data class RefundPaymentArgs(
 val paymentId: String,
 val reasonCode: String
)

@Serializable
data class RefundPaymentResult(
 val paymentId: String,
 val outcome: String,
 val refundReference: String? = null,
 val message: String
)

interface RefundPaymentService {
 suspend fun refund(
 paymentId: String,
 reasonCode: String
 ): RefundPaymentResult
}

class RefundPaymentTool(
 private val refundService: RefundPaymentService
): Tool<RefundPaymentArgs, RefundPaymentResult>(
 argsType = typeOf<RefundPaymentArgs>(),
 resultType = typeOf<RefundPaymentResult>(),
 name = "refund_payment",
 description = "Refund a payment when policy allows it."
) {
 override suspend fun execute(
 args: RefundPaymentArgs
 ): RefundPaymentResult {
 require(args.paymentId.isNotBlank()) {
 "paymentId must not be blank"
 }
 require(args.reasonCode.isNotBlank()) {
 "reasonCode must not be blank"
 }

 return refundService.refund(
 paymentId = args.paymentId,
 reasonCode = args.reasonCode
 )
```

```
  }
}
```

That shape gives you three useful properties immediately. First, the tool remains easy to audit because its responsibility is narrow. Second, the service can be reused by non-agent callers, which prevents the agent path from becoming a parallel business architecture. Third, infrastructure changes stay below the tool layer. You can swap an HTTP integration, introduce caching, or change persistence strategy without changing the agent-visible capability.

Keep the layers distinct

A practical layering model for agent execution is simple:

- **Tool contract**: the agent-facing capability with typed arguments, explicit semantics, and bounded effects.
- **Application service**: the business façade that enforces policy, coordinates domain operations, and owns use-case semantics.
- **Adapters**: HTTP clients, repositories, queues, SDK calls, and provider-specific translation code.
- **Infrastructure**: transport configuration, connection pools, serializers, retries, credentials, and telemetry plumbing.

You can compress these layers in a toy program, but you should still preserve the responsibilities. If a tool creates its own HTTP client, computes business eligibility, writes directly to a repository, and retries failed network calls, the boundary becomes opaque. Tests either become integration-heavy or unrealistic. Authorization is duplicated. Side effects spread into code that is hard to review as a capability surface.

The service layer is the hinge point. It gives you a place to hold business rules that are too important to live in a prompt and too domain-specific to live in transport adapters. It also keeps the agent from becoming the system's control center. The agent asks for a capability. The application service decides how that capability is fulfilled under domain policy.

What belongs in the tool

Keep the tool focused on four jobs:

- expose a stable, domain-oriented contract to the model,
- validate basic arguments early,
- delegate to an application-facing abstraction,
- normalize the result into the contract the agent expects.

That is enough. Once you go beyond those responsibilities, you usually start leaking infrastructure and policy upward.

Basic validation at the tool boundary is still valuable even when deeper validation exists in the service layer. Rejecting blank identifiers, malformed enums, or obviously invalid ranges saves cost and removes noise from downstream systems. More importantly, it makes failure categorization cleaner. Structural rejection belongs at the edge. Domain rejection belongs in the business service.

Normalization also belongs in the tool layer. A provider SDK may return dozens of fields, some unstable and some irrelevant to agent reasoning. The tool should reduce that response to the fields the contract promises. If the agent only needs `outcome`, `reference`, and a message, do not surface the rest just because the backend produced them.

What belongs in the application service

The application service should own business meaning. It decides whether a refund is allowed, whether a shipment can be rescheduled, whether a customer record may be updated, and whether an operation is idempotent or must be rejected when repeated. It can coordinate multiple adapters because that coordination is part of the use case rather than part of agent mechanics.

A service method is also where you want transaction ownership to be explicit. If a tool-triggered action updates internal state and then calls an external API, the service layer is where you reason about ordering, compensation, or outbox-style event emission. The tool should not be forced to understand those concerns.

That separation matters especially when the same business action exists outside the agent channel. A call-center console, a batch repair process, and an agent tool may all invoke the same refund service. If the business rules live in the tool, you now have multiple inconsistent paths for the same action. If they live in the service, the tool becomes another client of a governed use case.

Read-only query tools

Read-only tools are the easiest place to establish discipline because they have lower blast radius. They still need careful shaping. A query tool should not expose raw repositories or generic search primitives unless free-form query is itself the domain capability.

A good query tool depends on a read model or application service that returns deterministic, presentation-ready data. Consider a tool for checking order status:

```
import ai.koog.agents.core.tools.Tool
import kotlinx.serialization.Serializable
import kotlin.reflect.typeOf

@Serializable
data class GetOrderStatusArgs(
 val orderId: String
```

```
)

@Serializable
data class GetOrderStatusResult(
 val orderId: String,
 val status: String,
 val lastUpdatedAt: String,
 val customerVisibleMessage: String
)

interface OrderQueryService {
 suspend fun getOrderStatus(
 orderId: String
 ): GetOrderStatusResult
}

class GetOrderStatusTool(
 private val orderQueryService: OrderQueryService
): Tool<GetOrderStatusArgs, GetOrderStatusResult>(
 argsType = typeOf<GetOrderStatusArgs>(),
 resultType = typeOf<GetOrderStatusResult>(),
 name = "get_order_status",
 description = "Get the current status of an order."
) {
 override suspend fun execute(
 args: GetOrderStatusArgs
 ): GetOrderStatusResult {
 require(args.orderId.isNotBlank()) {
 "orderId must not be blank"
 }
 return orderQueryService.getOrderStatus(args.orderId)
 }
}
```

The query service might call a repository, cache, or external order platform. The tool should not care. If the service later switches from direct database reads to a denormalized status view, the contract stays intact.

Read-only tools are also a good place to establish deterministic result shaping. Keep timestamps in a consistent format, return domain states rather than transport-specific codes, and avoid backend-specific null conventions. If one upstream system uses `PEND` and another uses `WAITING_FULFILLMENT`, normalize both into the tool's published vocabulary before the result reaches the agent.

Mutation tools demand stricter boundaries

Mutation tools are where architectural shortcuts become risky. If a tool can change state, charge money, update entitlements, or trigger an irreversible workflow, it must pass through the same business façade that any other privileged caller would use.

Do not let the tool bypass policy by calling a repository directly because "the service only adds a small wrapper." That wrapper is often where authorization, invariants, idempotency keys, and audit hooks live. If the agent path skips those checks, you have created a fast lane around your own governance.

The service layer should also own hidden coordination. A refund may require checking payment state, recording a domain event, invoking an external provider, and persisting the provider reference. Those are business steps, not tool-layer details. The tool's job is to represent the capability clearly and return the outcome cleanly.

For mutations, prefer result envelopes that make execution state obvious. A boolean `success` is usually too weak. Return whether the action was accepted, completed, already applied, rejected by policy, or left uncertain due to timeout. That supports safer orchestration and better user messaging.

Composition tools versus orchestration

Some capabilities need data from multiple systems. For example, "summarize customer renewal risk" may require billing data, support history, and account tier information. You have two valid places to compose that work: inside a domain service behind one tool, or in orchestration across several tools. Choose based on business ownership, not just implementation convenience.

If the composition is a single business use case with stable meaning, keep it behind one tool and one service façade. The agent then asks one

question and receives one coherent answer. The service can coordinate the necessary backends and hide provider-specific inconsistencies.

If the work is genuinely exploratory or if the intermediate results matter to later reasoning, separate tools may be better. That decision becomes more relevant when you introduce parallel execution and graph-level control. Keep the distinction clean: a composition tool is justified when the composed action is itself a bounded domain capability, not merely because you wanted to hide complexity.

A useful test is whether the business would name the composed operation as a use case independent of the agent. If yes, a service façade and a single tool often fit well. If no, you may be collapsing orchestration into the tool for the wrong reason.

Direct external API call or internal façade

A recurring design choice is whether the tool should call an external API directly or go through an internal application façade. Both are possible; the façade is usually safer.

Direct API calls can be acceptable when the capability is thin, the organization truly treats the external API as the source of business semantics, and there is no meaningful internal policy beyond credential management and response normalization. This is more common for low-risk read-only lookups.

An internal façade is the better default when any of the following are true:

- the operation carries business rules,
- the same action is used by multiple internal clients,
- you need stable normalization across changing vendors,
- you require audit records independent of the provider,

- retries and timeouts need domain-aware handling,
- the external API is not trustworthy as your long-term contract.

The façade gives you an internal point of control. You can map provider codes into domain outcomes, hide credential rotation, isolate breaking changes, and test the use case without booting the entire agent stack. It also reduces blast radius when the external API degrades. Instead of many tools each handling provider quirks separately, one service boundary absorbs the instability.

Timeouts and retries belong at the right layer

A common mistake is to implement retries in the tool because the tool is where failure becomes visible. Visibility is not ownership. Retry policy belongs where you know whether repetition is safe and useful.

If a downstream HTTP GET is idempotent and transient failure is common, the adapter or service layer can often retry safely. If a mutation may already have committed remotely despite a timeout, blind retries in the tool are dangerous. The service layer is better placed to decide whether to reconcile state, query status, or require an idempotency mechanism.

Keep the tool aware of time budgets, but do not make it the home of transport heuristics. It is reasonable for the tool or surrounding runtime to impose an overall deadline for the capability. It is less reasonable for the tool to own exponential backoff logic tied to a particular vendor's error modes. That policy is infrastructure- and domain-aware, so keep it lower.

When a timeout leaves business outcome uncertain, reflect that explicitly in the result or error model. An agent should not receive "failed" if the payment provider may already have processed the refund. That is how double actions happen.

Serialization choices affect boundary stability

Koog serializes tool arguments and results, which means the types you choose at the tool boundary are not just in-memory convenience objects. They are part of a serialized contract that may interact with persistence, tracing, or cross-component boundaries.

Prefer simple, explicit DTOs with stable field names and serializer-friendly types. Avoid exposing internal entities directly. Entities often contain lazy relationships, storage annotations, derived fields, or nullability rules that make sense inside persistence code and become liabilities at a service boundary.

Serializer choice also matters. The default serializer is Kotlinx-based, while a Jackson-based serializer is available on the JVM. If your broader application already standardizes on Jackson, that may shape how you design DTOs and compatibility tests. Even then, keep the tool contract independent of any single serialization library's quirks. Do not rely on reflection-heavy magic, ambiguous polymorphic structures, or backend-specific naming tricks if the type may cross persistence or interop boundaries later.

A small, explicit result type is easier to evolve than a flexible map. Flexible maps look future-proof and usually become schema debt.

Per-call metadata is not part of the contract

Tool static metadata identifies the capability. Per-call metadata identifies the invocation context. Keep those apart.

Do not encode tenant identity, correlation IDs, request origin, or authorization facts into the tool definition itself. Those values belong to runtime call context. If you treat request-specific context as static tool metadata, you turn a reusable capability into a per-request artifact and make registry management harder.

The same separation helps when testing. You can verify that a tool contract stays stable across releases while independently testing that the runtime supplies the right call metadata for tracing, tenancy, or policy enforcement.

A concrete operational flow

A production-grade invocation path is straightforward when responsibilities are placed correctly.

The request reaches the agent with some user goal. The model selects a tool whose name and description match the needed capability. Koog validates the arguments against the declared tool types and rejects structurally invalid requests before any external work begins. The tool receives typed arguments and calls an application service. The service performs domain validation, authorization, idempotency checks, and coordination across repositories or remote APIs. Adapters handle HTTP, SDK, database, or queue interactions. The service then maps the outcome into a domain result. The tool returns a normalized, structured response to the runtime. Orchestration decides whether to continue reasoning, ask a clarifying question, surface a recoverable error, or terminate the action.

That flow gives you several good failure boundaries. Structural errors stop at the tool edge. Domain errors stop at the service boundary. Transport failures are absorbed and categorized below that line. The agent receives a result shape that preserves meaning instead of raw backend leakage.

Partial success versus hard failure

Some composed actions do not fit a binary success model. A customer-summary tool may fetch account data successfully but fail to retrieve support history. A mutation may update internal state but fail to notify a secondary system. You need an explicit rule for when to return partial

success and when to fail the entire capability.

Use partial success when the tool contract can define a useful degraded result without misleading the caller. That usually works for read-heavy aggregation. Include explicit completeness markers so orchestration can reason about whether the result is safe to use.

Use hard failure when the action's meaning depends on all coordinated steps completing, or when partial completion creates unacceptable ambiguity. Mutations fall into this category more often. If only half the side effect happened, the response should communicate that as a failure state that demands operational handling, not as a successful result with a warning buried in text.

Keep the decision in the service layer because it depends on business semantics, not just transport outcomes.

Testing the boundary you intend to keep

You should test tools and services separately. Tool tests should verify contract behavior: argument rejection, delegation, and result normalization. Service tests should verify business rules, downstream coordination, and failure mapping. Adapter tests should verify provider-specific translations.

That test split is not ceremony. It protects the architecture from drifting into agent-centric code. If every meaningful test must instantiate the agent runtime to verify a refund rule, your business logic has migrated into the wrong layer. If a tool cannot be tested without a live HTTP service because it owns transport setup itself, the boundary has already leaked.

Anti-patterns that collapse the architecture

Several integration shortcuts deserve active avoidance.

Direct repository access from tools. This bypasses business pol-

icy, couples the tool to persistence shape, and makes transaction semantics accidental.

Generic HTTP tools. A tool that accepts a URL, method, and body is not an integration strategy. It is unconstrained remote code execution with JSON syntax.

Policy bypass through convenience paths. If the application has a vetted service but the tool calls the lower-level adapter because it is easier, the agent path becomes the least governed path in the system.

Infrastructure-rich tools. When the tool owns client construction, retry loops, fallback rules, and circuit-breaking behavior, it becomes difficult to review as a business capability and difficult to reuse elsewhere.

Non-idempotent mutation behind ambiguous names. A tool called `update_customer` that performs multiple remote writes and sends a notification is too broad for safe recovery. Split intent and name effects honestly.

Service boundaries shaped by backend topology. If the tool mirrors the fact that data lives in three databases and one SaaS platform, you are exporting implementation accidents instead of domain capabilities.

The best integrations keep the tool contract narrow, the business façade authoritative, the adapters replaceable, and the infrastructure invisible at the boundary. When the agent invokes a capability, it should cross one intentional service boundary into governed application logic rather than falling through a trapdoor into whatever backend happened to be easiest to reach.

4.3. Parallel Tool Calls and Execution Coordination

Latency pressure pushes teams toward parallelism early. A single tool call to a payment gateway or internal service may take hundreds of milliseconds, and a useful answer may require three or four such calls. The naive conclusion is obvious: if one call is slow, launch all of them together. That conclusion is sometimes right for pure lookups and frequently wrong for actions that share state, compete for rate limits, or depend on subtle ordering. Once tool calls can mutate external systems, parallelism stops being a mere performance trick and becomes a coordination problem.

You should therefore ask a narrower question than "Can these calls run concurrently?" Ask whether these calls are independent enough that overlapping them preserves correctness, produces a coherent result, and improves user-visible latency enough to justify the operational cost. That framing keeps parallel execution attached to tool semantics rather than to thread availability.

A minimal coroutine example makes the distinction concrete. If you need customer profile data and recent invoices from independent services, fan-out can reduce total wait time because each request is read-only and neither one changes the assumptions of the other.

```
import kotlinx.coroutines.async
import kotlinx.coroutines.coroutineScope

suspend fun loadCustomerSnapshot(
 customerService: CustomerService,
 billingService: BillingService,
 customerId: String
): CustomerSnapshot = coroutineScope {
 val profile = async {
 customerService.fetchProfile(customerId)
 }
 val invoices = async {
 billingService.fetchRecentInvoices(customerId)
```

```
    }

    CustomerSnapshot(
    profile = profile.await(),
    recentInvoices = invoices.await()
    )
}
```

This code is intentionally plain. It uses `coroutineScope` so both child coroutines belong to the same structured unit of work. It uses `async` because you need values back, not just completion signals. If one branch fails, the scope cancels the sibling rather than letting unrelated work keep running in the background. That matters for tool orchestration because stray external requests consume rate limits, hold connections, and may continue mutating state after the calling operation has already failed.

Concurrency is not the same as correctness

Kotlin coroutines make concurrent work cheap to express because suspension avoids blocking a thread while waiting for I/O. That does not mean concurrent work is free. On the JVM, the suspended computations still resume on OS-managed threads, and the external systems still experience real overlap in traffic, contention, and failure.

The real design problem is semantic, not mechanical. Parallelism is safe only when overlapping calls does not change the business meaning of the operation. A pair of independent reads usually meets that condition. Two writes to the same customer record usually do not. A read followed by a write often requires ordering because the write may depend on the observed state. Two writes to different systems may still be unsafe if the user expects one coherent business action and you have no compensation strategy when only one write succeeds.

That is why parallel tool execution belongs alongside capability design and service boundaries, not inside a generic concurrency discussion.

The tool layer determines what can be asked of the system. Coordination determines whether multiple such requests can overlap without corrupting intent.

Use explicit tests for independence

Before you run tools in parallel, test four kinds of independence.

- *Input independence.* Each call should be computable without waiting for the result of the others. If one tool's arguments depend on another tool's output, you do not have a parallel workload. You have a sequence.
- *State independence.* The calls should not contend on the same mutable business object unless the downstream system explicitly guarantees safe concurrent handling. Two calls that update the same order, payment, or account usually need serialization.
- *Effect independence.* Even if the resources differ, the side effects should not conflict semantically. Refunding one payment and cancelling the corresponding shipment may be logically coupled even if they touch separate services.
- *Merge independence.* You need a clear rule for combining the results. If the orchestration layer cannot state how two outcomes produce one coherent next step, fan-out is premature.

These tests are stricter than "the calls use different URLs." Service boundaries hide shared state all the time. Two endpoints may write the same ledger. Two repository methods may lock the same row. Two tools may each look harmless in isolation and still produce contradictory outcomes when issued together.

Execution categories

Tool interactions fall into a few recurring categories, and each category suggests a different execution strategy.

- *Pure lookups.* Read-only calls with disjoint dependencies are the best candidates for fan-out. A weather lookup and a calendar query, or a customer profile fetch and a billing summary fetch, can usually run together if the results merge cleanly.

- *Lookup plus decision plus mutation.* This pattern usually needs ordering. You read current state, decide whether action is allowed, then mutate. Running the read and the mutation concurrently creates a race against your own validation logic.

- *Coupled mutations.* These should usually run sequentially or inside a workflow that can compensate. If one action reserves inventory and another charges a card, parallel launch may shorten latency while expanding failure handling dramatically.

- *Long-running operations with polling.* You may start one long-running action and concurrently fetch unrelated supporting data or poll a separate status endpoint. Here the parallel work is about progress coordination rather than simultaneous mutation.

- *Speculative alternatives.* Sometimes you run multiple retrieval paths and keep the fastest or highest-quality result. This can be useful for read-heavy tasks, but it raises observability and external-pressure costs quickly.

These categories matter more than implementation detail because they force you to reason about business meaning. If you cannot place a candidate workload into one of them, you probably have not clarified the semantics enough to parallelize it safely.

Execution style	Latency benefit	Correctness risk	External pressure
Sequential	Low	Low	Low
Bounded parallel	Medium to high	Medium	Medium
Unbounded parallel	High in narrow cases	High	High

Figure 4.1: *Relative trade-offs among sequential, bounded parallel, and unbounded parallel tool execution. Higher is better only for latency benefit; higher is worse for the risk and pressure dimensions.*

Coroutines give you overlap; structure gives you control

At the code level, `launch` and `async` solve different problems. Use `launch` when you care about lifecycle and completion but not about returning a value. Use `async` when each branch computes a result and you intend to `await()` it. Tool orchestration usually favors `async` because each branch returns a structured outcome that must be merged, inspected, or escalated.

Structured concurrency is the more important property. A group of tool calls should live inside one scope so cancellation and failure have one owner. Without that structure, one branch can fail, the parent can move on, and sibling requests may keep running against external systems with nobody left to consume their results safely.

A small anti-pattern makes the point:

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.async

suspend fun unsafeFanOut(
 serviceA: ServiceA,
 serviceB: ServiceB
) {
 val detached = CoroutineScope(Dispatchers.IO)

 detached.async { serviceA.call() }
 detached.async { serviceB.call() }
}
```

This code launches work that is not tied to the caller's lifecycle. If the surrounding operation fails or is cancelled, those external calls may continue anyway. Detached concurrency is especially dangerous for tool execution because the work crosses service boundaries and may produce effects the runtime can no longer coordinate.

Sequential, bounded parallel, and unbounded parallel

Most real systems choose among three broad execution styles.

Sequential execution minimizes concurrency risk. It simplifies tracing, preserves ordering, and reduces pressure on downstream services. The trade-off is user-visible latency, especially when the work is read-heavy and independent.

Bounded parallel execution overlaps only the calls that are known to be independent and constrains fan-out to a small, deliberate set. This usually gives the best balance of latency reduction and operational control.

Unbounded parallel execution treats every possible branch as a latency optimization. It can reduce median latency in narrow cases and degrade everything else: correctness, tail latency, external rate-limit behavior, and debugging.

The practical default is bounded parallelism. You should earn unbounded fan-out with strong evidence, not assume it as a baseline.

User-perceived latency versus system-wide pressure

Parallelism can make a response feel much faster because the user sees the slowest branch rather than the sum of all branches. That is the main benefit, and you should keep it visible in your reasoning. A fan-out of three independent 400 ms lookups can finish in roughly the time of one lookup rather than 1.2 seconds, ignoring overhead. That improvement is real and often worth pursuing.

The same fan-out also multiplies outgoing requests, connection usage, and exposure to slow tails. If each dependency has a modest chance of being slow, the chance that at least one of them is slow rises with every branch. This is why systems that optimize only average latency often end up with worse tail behavior. You save the easy cases and make the hard cases noisier.

External pressure matters just as much. Suppose each branch hits the same vendor API with shared tenant limits. Parallel fan-out may speed up one request while increasing throttling across the whole fleet. A local win becomes a fleet-wide regression. That is one reason to keep concurrency decisions close to service ownership rather than burying them inside ad hoc orchestration logic.

Cancellation is part of correctness

Cancellation is not just a resource-saving feature. It is part of semantic correctness when concurrent tool calls represent one higher-level intention. If one branch fails and the overall result can no longer be used, sibling branches should stop promptly. Otherwise you continue doing work for an operation that no longer has a valid consumer.

Structured concurrency helps here. Inside a `coroutineScope`, a failing child cancels its siblings. That behavior is often what you want for independent read aggregation where the combined answer is meaningful only if all required parts arrive.

There are exceptions. Sometimes a partial answer is acceptable, and you want one branch to fail without cancelling the others. In those cases you should make that choice explicit in the orchestration and result model rather than accept it accidentally. Partial-success design is a business decision, not a side effect of the coroutine API.

Cancellation also becomes harder once calls cross service boundaries. An HTTP request already sent to a remote system may not stop merely

because your local coroutine is cancelled. The runtime can stop waiting, release local resources, and prevent further steps, but the downstream action may already be in flight. That is another reason mutation-heavy concurrency needs stronger contracts, clearer idempotency, and explicit reconciliation paths.

Deadlines and budget allocation

A useful concurrent execution plan has both an overall deadline and per-branch budgets. Without budgets, one slow branch can dominate the user experience and hold up all the faster work. With poorly chosen budgets, you can cancel calls too aggressively and throw away useful partial results.

The clean mental model is hierarchical. Start with the user-visible latency budget for the whole operation. Allocate a portion of that budget to each concurrent branch according to its expected latency and criticality. A branch whose result is optional or easily recomputed should not consume the entire request budget. A branch that gates a business-critical decision may deserve a larger share or an earlier start.

Do not confuse per-branch deadlines with permission to retry blindly inside each branch. Concurrent retries compound quickly. Three branches with aggressive retry loops can generate a storm against already degraded dependencies. Keep retries bounded and domain-aware, and prefer coordination at the service layer when repeated attempts might multiply side effects.

Merge semantics are the real contract

Fan-out is easy to code. Merge logic is where correctness either survives or collapses. If multiple tool calls produce values, you need an explicit rule for how those values determine the next step.

For simple read aggregation, the merge rule may be straightforward: construct a composite response from all successful branches. For rank-

ing or selection, the merge rule may choose the best result by a score. For speculative execution, the merge rule may pick the first acceptable result and cancel the rest. For decision support, the merge rule may require all mandatory fields and mark the outcome incomplete otherwise.

Koog's graph workflows support concurrent node execution through `parallel()` and provide merge strategies such as `selectBy`, `selectByMax`, `selectByIndex`, and `fold`. Those mechanisms are useful because they force you to state how multiple branches become one continuation. That stated merge policy is more important than the fact of parallelism itself. If you cannot define a merge rule cleanly, the work probably does not belong in parallel branches yet.

You should also distinguish technical merge from semantic merge. Technical merge combines data structures. Semantic merge decides whether the combined result means "ready to proceed," "degraded but usable," or "stop and escalate." Keep both explicit.

Parallelism at the tool layer versus workflow layer

Some coordination logic is small enough to live near tool invocation. A few independent reads followed by a simple merge often fit there. When the execution plan requires branching, retries with policy awareness, compensation, staged batches, or deterministic sequencing across many steps, you are already moving toward workflow logic. At that point, a typed graph is usually a better home than ad hoc nested coroutines.

This is not about code style. It is about owning state transitions. A workflow makes dependencies, merge points, retry paths, and cancellation boundaries first-class. Raw concurrency primitives alone do not. Use coroutines to express overlap; use a workflow when the business process itself has structure that must remain visible and auditable.

Concrete coordination patterns

Three patterns cover most safe use cases.

- *Fan-out/fan-in reads.* Launch independent lookups, await them, and merge into one response. Keep the branch count small and the merge rule explicit.
- *Staged concurrency.* Run one stage sequentially to establish state or eligibility, then fan out independent reads or side-effect-free enrichments. This often gives most of the latency benefit without violating business ordering.
- *Speculative lookup with winner selection.* Query multiple sources for the same informational result and choose the fastest acceptable answer. Use this only when the extra pressure is justified and the duplicate requests are harmless.

Mutation-heavy patterns need more caution. Two side effects that appear independent can still create user confusion when they complete in different orders or fail asymmetrically. If you cannot explain the recovery story in one or two sentences, do not parallelize the mutations casually.

Failure isolation and partial success

Not every concurrent branch should have the power to fail the whole operation. The decision depends on what the user asked for and what the tool contract promises.

If the user asked for a unified answer that requires all components, fail fast when one mandatory branch fails. If the user asked for a best-effort summary, you may accept partial results and return explicit completeness markers. The key is that the contract must make this visible. A partial answer presented as complete is worse than a hard failure because it invites incorrect downstream action.

Isolate failures by category as well. A timeout on an optional enrichment service should not be treated the same way as a validation failure on a required branch. The merge logic should know which branches are critical, which are optional, and which produce advisory data only.

Anti-patterns that make parallelism dangerous

A few failure modes recur often.

- *Parallelizing contending mutations.* Two tools update the same business object through different service boundaries. The code runs faster in testing and produces occasional contradictions in production.
- *Ignoring cancellation.* One branch fails, but siblings keep hitting remote systems because the work was launched outside a structured scope or because downstream cancellation was never considered.
- *Underspecified merge rules.* The team fans out multiple retrievals and then informally decides what to do in prompt text or scattered glue code. This is how contradictory or partial data becomes a "successful" answer.
- *Retry storms.* Every branch retries independently against the same degraded dependency. Fleet-wide latency and error rates climb together.
- *Using parallelism to mask poor tool design.* If one tool tries to perform multiple unrelated operations, teams sometimes split the work dynamically and run it in parallel. That does not fix the capability surface. It multiplies ambiguity.
- *Assuming parallel means faster.* When the branches share a bottlenecked dependency, fan-out often shifts latency from sequential waiting to external throttling and queueing.

Decision criteria you can apply quickly

Use sequential execution when correctness depends on ordered state changes, when calls share mutable resources, or when merge semantics are ambiguous.

Use bounded parallelism when the branches are independent, read-heavy or safely isolated, and the latency savings are large enough to matter to users or upstream budgets.

Avoid unbounded parallelism unless you have strong evidence that the dependencies can absorb it, the merge rule is precise, and the operational team can observe and troubleshoot the resulting behavior.

A simple litmus test helps: if you would hesitate to explain to an incident responder why two given actions were launched together, do not launch them together. Concurrency that cannot be defended operationally is usually concurrency that has not been designed semantically.

Well-coordinated parallel tool execution feels almost uneventful in production. It reduces latency where independence is real, preserves ordering where business meaning depends on it, cancels wasted work promptly, and makes the merge rule visible enough that operators and code reviewers can reason about it without guessing.

4.4. Protocol-Based Interoperability with MCP and ACP

A local tool is the most direct way to give an agent a capability. It runs in the same process, shares the same deployment unit, and stays under your immediate control. That directness is also a constraint. The moment capabilities must cross team boundaries, runtime boundaries, or product boundaries, in-process tools become awkward. You can still

wrap remote APIs behind local tools, but every integration becomes bespoke, every capability advertisement is ad hoc, and every remote trust decision is hidden inside custom code.

Standardized agent protocols change that boundary. They let capabilities be described, discovered, invoked, and governed through a shared contract rather than through one-off wrappers. That shift is architectural, not cosmetic. When you choose a protocol such as MCP or ACP, you are choosing where trust lives, how schemas are negotiated, how latency and failure are surfaced, and who owns the long-term compatibility story.

A concrete Koog example helps anchor the idea. Koog can import remote MCP-provided tools into a local registry through an MCP registry provider. The exact transport set is version-sensitive because MCP support is still marked beta in the Koog documentation, but the shape of the integration is stable: connect to an MCP server over a supported transport, obtain a registry of remote tools, and expose those capabilities to your agent as if they were tools in the local environment.

```
import ai.koog.agents.mcp.client.McpToolRegistryProvider

suspend fun loadRemoteRegistry() =
    McpToolRegistryProvider.fromSseUrl(
        url = "http://localhost:8080/sse"
    )
```

Depending on the transport and module version you pin, Koog also exposes other provider construction paths such as process-based and transport-based factories. The important design point is not the factory name. It is that the agent is no longer consuming a local implementation detail. It is consuming a remotely advertised capability surface governed by a protocol contract.

What protocols solve that local tools do not

Local tools are excellent when you control the runtime, own the service

boundary, and want low latency with strong in-process observability. They become less attractive when a capability belongs to another team, another product, or another platform. In those cases, a protocol gives you three things that local wrappers tend to implement inconsistently.

First, a protocol gives you *capability advertisement*. Instead of hard-coding every remote action into local wrapper classes, you can discover or ingest capabilities described in a standard shape.

Second, a protocol gives you *request and result schemas*. The boundary is no longer "call this URL with whatever JSON this vendor expects." It becomes "invoke this named tool under a published schema."

Third, a protocol gives you *transport-neutral semantics*. The details vary by protocol and implementation, but the architectural point is that capability contracts become more durable than the current transport or deployment topology.

That durability is often the real reason to adopt a protocol. Teams sometimes describe protocol integration as a convenience feature. In practice, it is a governance feature. It lets you separate *what* a capability is from *where* and *how* it is served.

Protocol contracts are not just remote function calls

You should not think of MCP or ACP as fancy RPC libraries. That analogy is too weak. A remote function call assumes you already know the callee, the method, the arguments, and the trust relationship. Protocol-based interoperability introduces a richer boundary: capability discovery, schema-driven inputs, structured results, transport concerns, and another party's policy domain.

MCP is the clearest example in this space. It formalizes tool definitions with JSON-schema-shaped input descriptions and optional output descriptions. Tool results can include both general content and structured content. That matters because it supports a cleaner split between

human-readable output and machine-usable data. When you bridge an MCP tool into Koog, you are not merely tunneling a raw HTTP request. You are importing a capability whose input contract and result envelope are designed to be agent-consumable.

That is a major improvement over many bespoke wrappers, but it does not make the trust problem disappear. A protocol can standardize invocation and still leave you with questions about authentication, authorization, rate limits, side effects, and data handling. Standardization reduces friction; it does not remove responsibility.

Three integration models

Once protocols enter the picture, you usually choose among three broad patterns.

Local custom tools. You write Koog tools in-process and bind them to your own services. This gives you the strongest control over latency, runtime behavior, and governance. It also couples capability delivery to your runtime and deployment choices.

Remote capabilities behind internal façades. You still expose local tools to the agent, but those tools call internal services that may in turn speak to remote systems or even protocol endpoints. This preserves a local governance layer and often fits organizations that want strong internal review before exposing remote capabilities.

Protocol-native remote capability providers. The agent consumes capabilities advertised through MCP, ACP, or a similar protocol, with the local runtime acting primarily as a client, adapter, or orchestrator.

The third pattern is not automatically superior. It is better when the organizational boundary is real, when remote ownership matters, or when portability across runtimes and products is strategically valuable. It is worse when you need tight latency control, uniform policy enforcement, or a very small trusted surface.

When protocol interoperability pays off

Protocols earn their complexity when one or more of the following pressures are present.

- *Cross-team capability sharing.* If multiple teams want to expose tools to multiple agent runtimes, a standard protocol prevents every consumer from building a different wrapper.
- *Cross-runtime interoperability.* If capability producers and consumers live in different languages, frameworks, or deployment models, protocols give you a stable meeting point.
- *Independent operational ownership.* A remote capability provider can evolve, scale, and deploy on its own schedule as long as it preserves the published protocol contract.
- *Centralized governance at the provider boundary.* Remote providers can enforce their own policy, authentication, and auditing rather than relying on every consumer to implement those concerns consistently.
- *Capability portability.* A protocol-based tool provider can be reused by multiple agent hosts or products without requiring each one to embed the same business logic.

These are structural advantages, not just implementation conveniences. If none of them matter, local tools are usually simpler and safer.

What protocol interoperability costs

Every remote protocol boundary introduces at least four costs.

Latency. Even a well-behaved local network hop is slower than an in-process call, and the distribution tail matters more than the average.

If an agent action requires several remote protocol invocations, user-perceived responsiveness can degrade quickly.

Trust redistribution. With a local tool, you own the implementation and often the service beneath it. With a protocol provider, you are trusting another runtime, deployment unit, and operational team. The contract may be standardized while the behavior remains remote and partially opaque.

Reduced local control. You can wrap, cache, and constrain remote tools, but you do not control their internal retries, rollout timing, or provider-specific failure modes unless you have an explicit governance agreement.

Compatibility management. Protocols reduce bespoke integration code but replace it with version negotiation, schema evolution, transport compatibility, and provider-behavior monitoring.

These costs are acceptable when the organizational leverage is strong enough. If you adopt a protocol only to avoid writing a small local wrapper, the trade rarely closes in your favor.

MCP as a protocol-level tool boundary

MCP is particularly relevant because its vocabulary lines up well with the tool design principles already established. It treats tools as described capabilities rather than arbitrary code hooks. A tool advertises its input schema, may describe structured output, and participates in a request-response exchange that is explicitly about tool use. That is a better fit for agent systems than tunneling free-form vendor API calls.

For Koog users, the practical appeal is clear: an MCP tool server can become a source of tools for a Koog agent through registry import rather than through handwritten wrappers for every remote action. That does not mean you should import every available remote capability directly into production. It means the protocol gives you a standardized ingress

point for externally owned tools.

Koog's MCP integration is currently beta, so you should pin your chosen version and transport strategy carefully. The documented provider construction methods include process-based and SSE-based creation, and preview materials also expose streamable HTTP. That variation is a reminder to keep protocol adoption operationally explicit. Do not assume every transport shown in one source set is stable in every release you might deploy.

ACP as an agent-to-agent boundary

ACP addresses a related but somewhat different need. Where MCP is strongly tool-oriented, ACP is about agent capability exchange across an inter-agent boundary. In Koog, ACP support is also beta and JVM-only through the ACP feature module and `AcpAgent`. That JVM-only constraint matters if you are working in a multiplatform environment and were hoping to treat ACP as a uniform boundary everywhere.

Use ACP when the remote party is not merely exposing a low-level tool but acting as an agentic component with its own reasoning, policies, or orchestration. That is a larger trust move than consuming a tool-oriented protocol endpoint. You are no longer asking only whether the remote side can execute a bounded action. You are asking whether you are willing to let another agent-shaped system interpret intent inside its own policy domain.

That distinction matters operationally. A remote tool provider can often be reviewed as a capability server. A remote agent provider may require governance closer to service delegation, vendor assessment, and result validation.

A practical operational flow

Protocol interoperability becomes easier to reason about if you follow the full request path.

The agent determines that it needs a capability not implemented locally. It identifies that capability either because the remote provider has already been registered as a local tool source or because capability discovery is part of the runtime setup. The request is serialized according to the protocol contract, including the named capability and schema-shaped input. The remote provider receives the request and executes under its own policy, transport, and operational constraints. The result returns in protocol form, possibly containing both human-readable content and structured fields. The local runtime then decides whether to trust the result directly, normalize it further, retry, degrade gracefully, or escalate the failure.

That flow has two distinct trust decisions. The first is *capability trust*: should this remote capability be made available to the model at all? The second is *result trust*: given this response, should the local system treat it as authoritative enough to continue acting?

A local custom tool often collapses those decisions because both are under one codebase and one governance regime. Protocols separate them.

Discovery is power, and therefore risk

One attractive feature of protocol-based systems is dynamic capability discovery. That feature is also one of the main governance risks. If a provider can advertise tools dynamically, you must decide whether your agent may see every advertised capability, whether capabilities require local allowlisting, and whether the descriptions are trustworthy enough for autonomous selection.

Dynamic discovery reduces operational friction in shared ecosystems, but it can quietly expand your action surface. A provider upgrade that adds a new tool may immediately change what the agent can attempt unless you constrain registration. For production systems, you should prefer reviewed capability import over unconstrained discovery. Dis-

covery is useful for integration and development; allowlisting remains safer for stable environments.

This is the same lesson as local registry management, amplified by network boundaries. A protocol may standardize advertisement, but you still need a publication policy.

Schema alignment and semantic drift

A protocol can standardize field structure without standardizing business meaning. This is where many integrations fail. Two providers may both offer a tool named `cancel_order` with valid JSON-schema descriptions and still disagree about whether "cancel" means immediate reversal, queued request, or best-effort attempt. Structured inputs help, but semantic drift remains possible.

That means protocol adoption does not free you from capability review. You still need to examine names, descriptions, result semantics, and side-effect expectations. If the remote contract exposes optional `outputSchema`, use it to reduce ambiguity, but do not assume that schema alone defines business guarantees such as idempotency, compensation, or authorization behavior.

When the capability is business-critical, wrap the remote tool in a local normalization layer even if the protocol already gives you structured results. The protocol gets the data to you; your boundary still decides what meaning your system attaches to it.

Transport choices are architectural choices

It is easy to treat transport as a deployment detail. With protocol-based capabilities, transport affects reliability, debugging, security posture, and operational ergonomics. A process transport can be attractive for local or tightly managed integrations because lifecycle is explicit and latency is low. SSE or stream-oriented HTTP can fit networked providers better but add more infrastructure variables: reverse prox-

ies, idle timeouts, TLS termination, authentication propagation, and connection management.

Because Koog's transport support for MCP is version-sensitive, you should avoid over-generalizing from one example. Pick a transport deliberately, pin versions, and test failure modes specific to that transport. For example, networked transports need explicit handling for reconnect behavior and partial session loss. A local process bridge has a different operational profile: startup failures, process supervision, and host resource isolation matter more.

Do not let the protocol abstraction hide transport reality. It helps with portability but does not make network physics disappear.

Security and governance move outward, not away

Teams sometimes adopt protocols hoping to simplify security review through standardization. Standardization helps, but the review burden moves outward instead of vanishing.

You need to assess provider identity, authentication requirements, authorization scope, data exposure, audit behavior, and result trustworthiness. You also need to decide whether the model may call remote capabilities directly, whether orchestration should gate them, and whether certain tools require human approval or additional policy checks before invocation.

For remote protocol providers, governance questions become sharper:

- Who owns the provider and its change process?
- How are new capabilities introduced and reviewed?
- What data leaves your trust boundary on invocation?
- Can the provider trigger external side effects you do not fully observe locally?

- What is the fallback plan when the provider is degraded?

A protocol gives you a better language for these questions. It does not answer them.

When to wrap a protocol provider behind a local tool

A useful compromise is to consume a protocol provider but present selected capabilities through local, domain-specific tools. That pattern is often the safest way to combine interoperability with governance.

You might do this when the remote provider exposes many generic capabilities but your application needs only a narrow subset. You might also do it when remote result semantics need normalization, when local audit requirements are stricter than the provider's, or when your product should remain stable even if the provider's capability catalog evolves.

The local wrapper is then not redundant. It is your policy and meaning boundary. The protocol remains valuable because it prevents you from building bespoke low-level integrations, but the agent still sees a reviewed, domain-shaped interface rather than the raw remote catalog.

Failure modes and anti-patterns

Several protocol anti-patterns appear repeatedly.

- *Treating remote tools as local function calls.* This hides latency, partial failure, and trust boundaries. The code looks clean while the operational model becomes misleading.
- *Importing every advertised capability.* Dynamic discovery without review turns capability growth into silent risk expansion.
- *Ignoring capability versioning.* Even when a protocol stabilizes the outer shape, providers evolve names, descriptions, and semantics. Pin versions and review changes deliberately.

- *Assuming uniform provider quality.* Protocol compliance does not guarantee sound business behavior. A well-formed schema can still describe a poorly bounded or weakly governed capability.

- *Letting protocol sprawl bypass internal architecture.* If every team exposes remote capabilities directly and agents consume them ad hoc, you may end up with less governance than you had with ordinary service APIs.

- *Using protocol adoption as a substitute for contract design.* A remote tool with vague semantics remains vague even if delivered over MCP.

Decision signals that justify protocol adoption

Choose protocol-based interoperability when the capability boundary crosses genuine organizational or runtime lines, when reuse across multiple consumers is likely, when remote providers can own their policy and operations responsibly, and when the benefits of standardized discovery and invocation outweigh the extra latency and trust complexity.

Stay with local custom tools when you need tight latency control, one-team ownership, strict in-process governance, or a highly curated capability surface that does not benefit from remote advertisement.

Use a hybrid model when remote capabilities are real but your application still needs a local review and normalization layer. That is often the practical equilibrium in production systems: protocols reduce bespoke integration work, while local tools preserve the product's own safety and semantic discipline.

Protocol-level interoperability works best when you treat MCP and ACP as boundary choices rather than convenience APIs. They are most valuable when they make ownership clearer, contracts more portable,

and governance more explicit than a pile of custom wrappers ever could.

4.5. Java and Kotlin Interop at the Tool Layer

Most JVM teams do not get to choose a single-language world. You may write new agent code in Kotlin because Koog is Kotlin-first, while your core business services, framework integration, and operational libraries remain heavily Java-based. That is not a temporary inconvenience. It is the normal shape of a production estate. The tool layer becomes the place where those worlds either meet cleanly or grind against each other.

The key design move is to treat the tool layer as a translation boundary, not as a place to erase language differences. You do not need Java to consume every Kotlin idiom naturally, and you do not need Kotlin to inherit every Java convention. You need a small, stable collaboration surface where tools remain expressive for Kotlin authors and predictable for Java callers or service implementations.

A concrete pattern appears immediately when you expose a Kotlin-defined tool contract backed by a Java-friendly service interface. Keep the Koog-specific tool definition in Kotlin, and define the business-facing port in a shape both languages can handle comfortably.

```
import ai.koog.agents.core.tools.Tool
import kotlinx.serialization.Serializable
import kotlin.reflect.typeOf

@Serializable
data class LookupCustomerArgs(
 val customerId: String
)

@Serializable
data class LookupCustomerResult(
 val customerId: String,
 val fullName: String,
```

```
 val tier: String
)

interface CustomerLookupService {
 suspend fun lookupCustomer(
 customerId: String
 ): LookupCustomerResult
}

class LookupCustomerTool(
 private val service: CustomerLookupService
): Tool<LookupCustomerArgs, LookupCustomerResult>(
 argsType = typeOf<LookupCustomerArgs>(),
 resultType = typeOf<LookupCustomerResult>(),
 name = "lookup_customer",
 description = "Look up customer account details."
) {
 override suspend fun execute(
 args: LookupCustomerArgs
 ): LookupCustomerResult {
 require(args.customerId.isNotBlank()) {
 "customerId must not be blank"
 }
 return service.lookupCustomer(args.customerId)
 }
}
```

This looks ordinary because it should. The tool expresses Koog's contract model directly. The port behind it expresses domain behavior. The complication starts when that port must be implemented or invoked from Java. A raw Kotlin `suspend` boundary is awkward for Java because the compiler lowers suspending functions into coroutine state-machine forms that are natural in Kotlin and unnatural in Java source. That mismatch is the heart of interop design at the tool layer.

Choose the boundary before you choose the syntax

Mixed-language teams often start by asking how Java can call a Kotlin tool or how Kotlin can expose a Java service. That is too low-level. Start by deciding what belongs on each side of the boundary.

Kotlin is the natural home for Koog DSLs, tool definitions, coroutine-

aware orchestration, and compact data modeling. Java is often the natural home for mature business services, established enterprise frameworks, and large service estates with existing testing and deployment conventions. The best interop design preserves those strengths rather than trying to flatten them into one style.

That leads to a simple rule: keep Koog-facing code Kotlin-first, keep shared business ports conservative, and keep Java-facing boundaries explicit. When you do that, language differences stay localized. When you do not, the whole tool layer becomes a negotiation over defaults, nullability, and coroutine internals.

Do not subclass Kotlin tool APIs from Java

Koog supports both Kotlin and Java on the JVM, but the authoring path differs by language. In Kotlin, class-based tools built on `Tool<Args, Result>` or `SimpleTool` are the natural fit. In Java, that inheritance path is not the right boundary. Subclassing Kotlin `Tool` or `SimpleTool` from Java is not supported because the suspend-based execution model does not map cleanly to normal Java subclassing.

That limitation is not arbitrary. A suspending Kotlin method is not just a method returning a future-like object. It compiles into a continuation-based state machine with conventions that ordinary Java subclasses do not express ergonomically. You can call such APIs from Java with enough ceremony, but you should not build your public authoring model around that friction.

For Java-authored tools on the JVM, prefer annotation-based tool exposure rather than subclassing the Kotlin tool base classes. That keeps Java on a Java-friendly surface and avoids forcing developers to understand coroutine internals just to publish a tool.

Use thin Kotlin tool wrappers over stable service ports

A robust interop pattern is to keep the tool wrapper in Kotlin and let the

underlying service be implemented in either language through a deliberately shaped port. That port should avoid Kotlin-only conveniences that become awkward in Java, such as default parameters, DSL-heavy builders, and public nullable semantics that rely on Kotlin's type system more than on explicit contracts.

When the service is Java-heavy, consider exposing one of these shapes at the boundary:

- a synchronous interface if the work is naturally blocking or already integrated into synchronous infrastructure,
- a future- or callback-based interface if asynchronous completion must cross into Java ergonomically,
- a Kotlin-facing suspend adapter that wraps a Java service internally.

The wrong move is to let raw suspending APIs leak into Java-facing service contracts by default. Even when Java *can* interoperate at the bytecode level, that does not make the resulting source API pleasant, discoverable, or maintainable.

A practical adapter shape

Suppose you already have a Java service used across a large application. Do not rewrite it into coroutines just to satisfy the tool layer. Wrap it.

```
public interface CustomerDirectory {
 CustomerRecord findById(String customerId);
}
```

You can adapt that service in Kotlin and keep the suspension boundary local to the tool side.

```
class CustomerLookupAdapter(
 private val directory: CustomerDirectory
): CustomerLookupService {
```

```
 override suspend fun lookupCustomer(
 customerId: String
 ): LookupCustomerResult {
 val record = directory.findById(customerId)
 return LookupCustomerResult(
 customerId = record.getCustomerId(),
 fullName = record.getFullName(),
 tier = record.getTier()
 )
 }
}
```

This adapter is intentionally boring. It prevents Koog concerns from leaking into the Java service and prevents Java calling conventions from distorting the tool contract. If you later move the Java service to an asynchronous client or replace it entirely, the adapter is where that change belongs.

Java-friendly tool publication

When Java teams need to expose tools directly, use the JVM-oriented annotation model rather than forcing them through Kotlin inheritance. That lets you expose existing service methods with tool metadata and keep the service implementation close to ordinary Java coding style.

The exact annotation names matter here because this is framework API, not illustrative pseudocode. Koog's JVM annotation-based model uses `@Tool` and `@LLMDescription`. The practical guidance is simple: annotate thin façade methods, not deep infrastructure classes. Keep those methods domain-oriented and narrow, just as you would for Kotlin class-based tools.

The underlying principle remains the same across languages. A tool is a governed capability boundary. Interop changes how you publish that boundary, not what a good boundary looks like.

Keep shared DTOs boring and explicit

Shared data-transfer types are where many mixed-language designs

quietly decay. Kotlin encourages concise models with non-null properties, default values, sealed hierarchies, and collection literals that feel natural on the Kotlin side. Java often prefers mutable beans, overloaded constructors, and framework-driven reflection. If you let every local preference surface at the boundary, you create DTOs that are pleasant in one language and error-prone in the other.

For tool-layer interop, prefer explicit DTOs with these properties:

- stable field names,
- simple scalar and collection types,
- minimal inheritance,
- explicit nullability semantics,
- serializer-friendly structure.

This is especially important because tool arguments and results are serialized. The DTO is not just a convenience object inside one runtime. It is part of the contract that may be logged, persisted, traced, or handed across different serialization stacks. A clever type hierarchy that saves a few lines in Kotlin can become a costly boundary object once Java reflection, serializer configuration, or schema evolution enters the picture.

Nullability must be designed, not inferred

Kotlin's type system gives you null-safety as a first-class language feature. Java does not. Mixed-language teams often assume this means Kotlin can simply "protect" the boundary. It cannot, at least not by itself. Once Java enters the picture, nullability becomes an API design choice that must be made explicit.

If a field is required, make it required in the serialized contract and validate it early. If a field is optional, define what `null` means oper-

ationally. Does it mean unknown, inapplicable, not yet computed, or absent because of degraded upstream data? Those are different states, and collapsing them into a nullable reference often creates ambiguity in both languages.

At public boundaries, avoid relying on Kotlin callers inferring intent from type syntax while Java callers receive only looser conventions. Prefer domain names and result envelopes that make absence meaningful. A result field like `nextBillingDate: String?` is tolerable if the contract explains when it may be absent. A field like `details: String?` is usually a sign of semantic weakness, not just interop trouble.

Default parameters are not a public interop strategy

Kotlin default arguments are elegant for Kotlin callers and a common trap for Java-facing APIs. Java does not understand Kotlin default parameters directly as a source-level feature. Kotlin can generate overloads with `@JvmOverloads`, but that mechanism is not a general solution.

First, you should use it deliberately, not reflexively. Generated overloads can be useful for small helper façades, but they also enlarge the bytecode surface and can obscure which combinations are semantically valid. Second, `@JvmOverloads` cannot be used on abstract methods, including interface methods. That alone makes it unsuitable as the main strategy for Java-friendly public ports.

If a public boundary needs optional parameters and Java callers matter, prefer one of these approaches:

- define explicit overloads on a concrete façade,
- introduce a request object with named fields,
- separate distinct operations into separate methods instead of

stacking defaults onto one method.

This is not only about Java convenience. Request objects often make the tool boundary clearer and more serializer-friendly anyway.

Use `@JvmName` to keep signatures intentional

Kotlin allows declarations that compile to JVM signatures Java may find confusing or conflicting. Overloaded extensions, top-level functions, and property-generated accessors can create names that are natural in Kotlin and awkward in Java. `@JvmName` lets you control the exposed JVM-level name and avoid accidental clashes.

This is useful when you publish Java-facing façades around agent infrastructure. Use it to make names stable and unsurprising, not to create clever dual personalities. If Java developers are expected to call a Kotlin helper that assembles tool registries or starts agent execution, give that helper a clear JVM name instead of assuming the default generated form is good enough.

The deeper rule is consistent: interop works best when the JVM-visible API is designed intentionally rather than accepted as a byproduct of Kotlin source conventions.

Façades are often better than shared internals

A mixed-language codebase rarely benefits from exposing the same rich API to both Kotlin and Java unchanged. Kotlin callers may benefit from DSLs, extension functions, lambdas with receiver, and suspending composition. Java callers usually want straightforward classes, methods, and DTOs. Trying to preserve every Kotlin convenience at the Java boundary creates awkward Java APIs. Flattening every API into Java style wastes Kotlin's strengths.

A better pattern is a small Java-friendly façade over richer Kotlin internals. The façade can expose concrete methods, request objects, and

predictable return types. Internally, Kotlin code can still use more expressive composition to build registries, assemble agents, or manage coroutine-driven execution.

This separation also improves binary stability. Internal Kotlin idioms can evolve without forcing Java callers to absorb those changes. The façade becomes the contract you version and support.

Interop is not the same as service-boundary design

One of the easiest mistakes in mixed-language projects is to conflate language adaptation with architectural ownership. A Java-facing façade does not decide where business rules live. A Kotlin tool wrapper does not replace an application service. Those boundaries still matter.

Keep business logic in services, just as you would in a single-language system. Keep tool contracts narrow and domain-oriented. Keep infrastructure concerns below the service layer. Interop should help each layer remain accessible to the right language communities; it should not redraw the architecture around language lines alone.

If the Java team owns a domain service, let it own that service cleanly and adapt it at the tool edge. If the Kotlin team owns the agent runtime and orchestration, let it do so without turning every Java consumer into a coroutine expert. That division is far more sustainable than trying to make every part of the stack feel equally native in both languages.

Testing across the boundary

Interop bugs often appear as ordinary business defects: a missing field, a confusing overload, an unexpected `null`, or a reflection failure during serialization. You should test the boundary from both sides.

For Kotlin-defined, Java-consumed APIs, write at least one Java compilation or usage test that exercises the intended call shape. For Java-backed services adapted into Kotlin tools, test the Kotlin adapter inde-

pendently from the Java implementation so you can isolate translation errors from business-rule errors.

This dual testing discipline is especially valuable for DTOs and public façades. A data class that looks obvious in Kotlin may be annoying to construct from Java. A serializer-friendly constructor strategy in one library may interact badly with another stack. Interop testing catches these issues early, before the tool boundary becomes a long tail of small but costly surprises.

Common failure modes

Several interop failures recur in tool-layer systems.

- Leaking suspend semantics into Java APIs. The code compiles, but the Java usage is obscure and discourages correct adoption.
- Publishing Kotlin-first collections and DSLs as public Java APIs. Java callers then fight type inference, builder patterns, or generated names that were never meant for them.
- Relying on default parameters as the only Java convenience. This works for some concrete functions and fails exactly where stable public abstractions matter most.
- Exposing persistence entities as tool DTOs. This creates trouble in both languages and couples serialization to storage concerns.
- Using reflection-heavy magic across mixed tooling. What works under one build plugin or serializer configuration may fail under another, especially when Java frameworks and Kotlin metadata interact.
- Letting the agent layer become the language integration layer. Tools then absorb responsibilities that belong in façades, adapters, or services.

Choosing among interop strategies

Three strategies are common, and each has a place.

A **Kotlin-owned tool layer with Java service adapters** works best when Koog usage is concentrated in Kotlin and the existing estate is mostly Java services. This keeps agent concerns localized and usually yields the cleanest tool contracts.

A **Java-friendly façade over Kotlin agent infrastructure** works best when Java applications need to invoke agent functionality but should not depend directly on Kotlin-flavored APIs. The façade becomes the stable operational surface.

An **isolated Kotlin module with serialized request-response access** can work well when the language boundary is too wide for fine-grained API sharing. Instead of making Java call many Kotlin APIs directly, you expose one coarser service boundary. This sacrifices some ergonomics for stronger isolation and simpler binary compatibility.

Choose based on team skill distribution, debugging needs, deployment model, and how much of the agent runtime truly needs to be visible outside Kotlin. Do not choose based on a desire for symmetrical elegance. Symmetry is less valuable than a boundary each side can use confidently.

Version and compatibility discipline

Interop surfaces deserve stricter compatibility discipline than internal Kotlin code. Kotlin compiler upgrades, generated bytecode details, and library evolution can all affect the Java experience even when the Kotlin source looks stable. That does not mean you should freeze everything. It means you should be selective about what becomes a supported public boundary.

Keep public façades small. Keep DTOs explicit. Avoid exposing experi-

mental or rapidly evolving framework internals to Java callers. Where Koog provides a clear Java-oriented path, prefer that path rather than reverse-engineering Kotlin internals into Java-friendly shapes yourself.

The tool layer becomes far easier to maintain when each language sees an API that respects its strengths without forcing it to mimic the other. Kotlin can remain the expressive home for Koog definitions and orchestration. Java can remain productive around services, façades, and stable DTOs. The boundary between them should feel deliberate, narrow, and uneventful enough that neither side has to think about the other language more than necessary.

Chapter 5

Type-Safe Graph Workflows for Predictable Agents

Complex agent systems fail less often when their control flow is modeled as software rather than implied inside prompts. This chapter shows how Koog's typed graph workflows turn orchestration into a first-class engineering artifact, letting teams reason about state, retries, tool boundaries, and business accountability with the same rigor they apply to other production systems.

5.1. Why Graph Workflows Outperform Linear Prompt Chains

A sequential prompt chain often survives its first demo because the happy path is short, the operator watches every run, and failures are

cheap. You send one prompt, parse the reply, feed that reply into the next prompt, and repeat until the model emits something that looks like an answer or an action plan. For exploratory work, that can be enough. The trouble appears when the system stops being a demo artifact and starts owning real obligations: route requests differently by policy, call external services, validate intermediate results, recover from partial failure, explain why a decision was made, and produce evidence that the process stayed inside approved boundaries. At that point, the prompt chain is still executing, but the architecture is no longer visible in the code. It has dissolved into transcripts, parser assumptions, and ad hoc conditionals around model output.

Koog gives you a different execution model. Instead of treating orchestration as something implied by prompt wording and glue code, you declare it as a graph of typed execution units. The graph becomes the control surface. Nodes define what work can happen. Edges define which transitions are legal. State carries durable context in a form your program can inspect directly rather than reconstruct from a conversation history. That shift matters most when predictability is more important than improvisation.

A minimal graph-based strategy already exposes the architectural difference. Even without introducing validation or recovery yet, you can see the workflow shape as code rather than as an accident of prompt order.

```
import ai.koog.agents.core.agent.strategy.strategy
import ai.koog.agents.core.agent.strategy.node.node
import ai.koog.agents.core.agent.strategy.node.nodeFinish
import ai.koog.agents.core.agent.strategy.node.nodeStart
import ai.koog.agents.core.agent.strategy.edge.edge

val normalize = node<String, String> { input ->
 input.trim()
}

val classify = node<String, String> { input ->
 if (input.startsWith("refund")) "billing" else "general"
```

```
}

val workflow = strategy<String, String>("request-routing") {
 edge(nodeStart forwardTo normalize)
 edge(normalize forwardTo classify)
 edge(classify forwardTo nodeFinish)
}
```

This example is intentionally small, but it already makes one property explicit that prompt chains usually hide: there is a defined start, a defined finish, and named execution steps between them. In a prompt chain, you typically discover the workflow only by reading a growing pile of prompts, helper functions, and string parsing code. In a graph, the workflow is the primary artifact.

Where linear prompt chains fail under operational pressure

The core weakness of a linear prompt chain is not that it is sequential. Many real processes are sequential. The weakness is that sequence is often the only control structure it has. Once the system must do anything more complex than "take output from step n and feed it to step $n + 1$," engineers start encoding orchestration rules in places that are hard to inspect and harder to verify.

One common failure mode is *hidden state*. A prompt chain often uses the transcript itself as the source of truth. Important facts live inside natural-language exchanges: whether validation already passed, which tool failed, whether a user was authenticated, whether a side effect already occurred, or whether a retry has already been attempted. The system then relies on the next prompt to infer those facts from textual context. That works until the prompt changes, the model compresses a detail differently, or a downstream parser expects wording that no longer appears. Hidden state is especially dangerous in business workflows because the state you most need to trust is the state that must survive ambiguity.

A second failure mode is *implied control flow*. In a linear chain, branching is rarely declared as a first-class structure. Instead, the model may be asked to emit phrases such as "APPROVED" or "NEEDS_REVIEW," and a host-language conditional routes execution based on those strings. The workflow is then split between prompt instructions, parser conventions, and application code. If you need to answer a basic audit question—why did this request go down the escalation path rather than the automated path—the answer is scattered across model output, branching code, and whatever prompt template happened to be deployed at that moment.

A third failure mode is *error handling by accretion*. Prompt chains tend to acquire retries, fallbacks, and special cases as local patches. If a model returns malformed JSON, call it again with a sterner instruction. If a tool call times out, ask the model to retry. If validation fails twice, switch to a different prompt. If the external system partially completes the action, add cleanup logic in the exception handler. Each patch may be reasonable in isolation, but the control policy remains implicit. You end up with a system whose happy path is linear and whose real path is a mesh of unstated exceptions.

A fourth failure mode is *weak inspectability*. Debugging a production agent is not the same as evaluating a prompt interactively. Service owners need to know which execution unit ran, with what typed input, what output it produced, what state changed, which transition fired, and whether the same run can be reproduced under test. Linear chains are poor at answering those questions because the transcript is doing too much work. It contains intent, data, routing hints, and recovery instructions all at once. When the transcript is overloaded in that way, you can inspect everything and still understand very little.

The operational stakes are architectural, not stylistic

Graph workflows matter because production concerns attach to con-

trol flow, not just to model quality. A compliance reviewer does not only care whether the model usually makes a reasonable recommendation. The reviewer cares whether requests requiring approval can bypass approval, whether exceptions are visible, whether retries are bounded, whether side effects happen before validation, and whether policy-critical steps can be enumerated and tested. Those are orchestration questions.

The same applies to debugging and service ownership. Suppose a support workflow classifies inbound requests, enriches them with customer data, checks refund eligibility, and may trigger an external refund API. In a prototype, you can implement that as a handful of prompts and helper functions. In production, you need stronger guarantees. A malformed enrichment result must not quietly steer the request into the wrong branch. A partial external failure must not leave the system uncertain whether money moved. A reviewer must be able to see where eligibility was computed and what conditions allowed the side-effecting call. Once you state those requirements directly, the inadequacy of a free-form prompt chain becomes obvious. The system needs a control model, not just better prompts.

Vocabulary that makes the execution model explicit

The rest of the chapter depends on a small set of terms used precisely.

A *node* is a unit of execution. It accepts some input, performs work, and produces output. The work may be deterministic computation, a model call, a tool invocation, a validation step, or a state transformation. The key property is not what the node does internally, but that the node has a contract. You can name it, test it, observe it, and place it intentionally in the workflow.

An *edge* is a legal transition between nodes. It states that after one node completes, execution may continue to another node. In richer workflows, edges can be conditional. That turns routing into a declared

property of the graph rather than an emergent property of scattered if-statements and parser logic.

Workflow state is the durable context that matters to orchestration. It is not merely a message history. It contains the facts required to decide what can happen next and to explain what already happened. In a type-safe system, that state is shaped deliberately rather than reconstructed from text. Chapter material on state design and persistence expands this idea later; here the important point is that the graph carries state as data, not as conversational implication.

A *transition* is the runtime event of moving from one node to another through an edge. Transition semantics matter because they define the legal progression of work. If a request has not been validated, the graph should make it impossible, or at least explicit, to reach the side-effecting node that depends on validation.

Traversal is the runtime progression through the graph. This is the actual path a run takes from `nodeStart` toward `nodeFinish`. In a predictable system, traversal is not mysterious. You can inspect the path, explain it, and compare it against expected behavior.

Graphs separate orchestration from prompt wording

A prompt chain often forces prompt text to carry too many responsibilities. The prompt must define the task, preserve state, encode routing hints, ask for tool use, request retries, and sometimes even instruct the model how to recover from downstream failures. That bundling makes small prompt edits disproportionately risky because changing wording may alter not only the model's answer but also the system's control flow.

Graph workflows separate those concerns. The prompt can focus on the model task inside a node. The graph defines what happens before and after that node, what branches are permitted, and what must be

true before execution proceeds. This separation is one of the main reasons graph workflows age better than prompt chains. You can revise a node's prompt without accidentally rewriting the topology of the whole process. You can change a routing rule without editing natural-language instructions that also govern tool selection and output format.

That separation becomes even more important when tools enter the picture. Tool access alone does not make a system well orchestrated. In fact, it can make unpredictability worse if the model is free to decide when to call tools, how many times to call them, and how to recover when their outputs are malformed or incomplete. Tool construction belongs elsewhere. The orchestration point here is narrower and more important: a tool-using system is only predictable when tool execution sits inside explicit control flow.

Koog exposes that control flow directly through graph-based strategies. Official graph examples show named strategies, typed custom nodes, start and finish boundaries, conditional transitions, and predefined nodes for common agent behaviors such as model requests and tool execution. The significance is architectural rather than cosmetic. The framework treats workflow structure as a first-class program element.

Typed graphs reduce ambiguity at the exact points where prompt chains drift

When practitioners describe a prompt chain as "fragile," the fragility usually comes from one of three mismatches.

The first is a *schema mismatch*. One step emits text that the next step interprets as structured data. If the wording changes, the parser fails, or worse, succeeds incorrectly. Typed nodes reduce this risk by making inputs and outputs explicit in code rather than conventional in prose.

The second is a *state mismatch*. One step assumes a fact is known be-

cause it appeared earlier in the conversation; another step assumes the same fact was validated; neither assumption is encoded as a durable workflow invariant. Explicit workflow state gives those facts a stable home.

The third is a *routing mismatch*. The model suggests a branch, but the business process requires that the branch be allowed only under typed, validated conditions. In a graph, you can force routing to pass through explicit checks. In a prompt chain, routing often happens because a model emitted a plausible sentence that some downstream code interpreted optimistically.

These mismatches are not edge cases. They are the normal way prompt-chain systems degrade as requirements accumulate. The graph model directly targets them by moving contracts out of free-form language and into explicit program structure.

Branching is where prompt chains stop being readable

The moment you need branching, a prompt chain stops being a chain. It becomes a graph in disguise. You may still write the code as a sequence of prompt calls with conditionals around them, but the real control structure now has forks, joins, loops, and terminal states. Because that structure is not modeled directly, readability collapses.

Consider a routine business scenario: classify a request, gather evidence, validate policy constraints, execute an external action if permitted, and otherwise route to human review. In a linear implementation, each branch point is hidden behind parser output and application logic. To understand the full behavior, you have to inspect every place where strings are interpreted as route signals. To verify that the human-review path is always available when validation fails, you have to reason across multiple prompts and handlers. To prove that the external action never runs before validation, you must trust that no prompt or exception path bypasses the intended ordering.

A graph workflow makes branching visible. The legal paths are explicit. You can point at a node and ask which successors are allowed. You can point at a branch condition and decide whether it is based on validated data or model suggestion. You can encode that some transitions simply do not exist. That last point is more powerful than it first appears. Reliability often improves not because the graph adds more logic, but because it removes undeclared movement.

Recovery logic belongs in topology, not in apologetic prompts

A familiar anti-pattern in prompt-chain systems is to encode recovery in natural language: "If the tool call fails, retry once. If the output is invalid, ask the tool again. If confidence is low, escalate." That instruction may influence the model's behavior, but it does not define a reliable recovery policy. The retry count is soft. The failure classification is fuzzy. The distinction between transport failure, semantic failure, and policy rejection is blurred. The resulting behavior is difficult to test because the recovery mechanism is partly inside the model's reasoning and partly outside it.

Graph workflows give recovery a place to live. A retry can be represented as a deliberate loop with bounded conditions. A validation failure can route to a repair or escalation node. A side-effecting step can have an explicit compensation path. The chapter later drills into retries and compensating actions in detail; the architectural point here is simpler: once failure handling matters, the topology itself becomes part of your reliability mechanism.

This also improves runtime accountability. If a request was retried twice and then escalated, that path should be visible as graph traversal, not inferred from a transcript full of polite self-corrections. If a compensating action ran after a partial failure, that should appear as an explicit branch, not as an exception-side cleanup hidden in glue code.

Graphs support stronger testing because they expose contracts

Prompt chains are notoriously difficult to test beyond prompt-response snapshots. Snapshot testing can catch regressions in wording or output structure, but it does not usually tell you whether the workflow remains legal under all required conditions. Graph workflows support deeper testing because each node and edge embodies a contract.

You can test nodes independently with controlled inputs. You can test routing predicates without invoking the entire model stack. You can test that invalid state cannot reach restricted nodes. You can simulate failure classes and verify that traversal reaches the correct terminal condition. You can inspect traces in terms of node execution and state changes rather than reconstructing a narrative from text.

This style of testing does not eliminate uncertainty from model-based systems, but it localizes it. The model remains stochastic where it should remain stochastic: inside the node that requires model behavior. The orchestration layer becomes more deterministic, inspectable, and auditable.

Why explicit graphs deserve architectural status

It is tempting to treat graphs as a visualization convenience placed on top of ordinary agent code. That understates their value. In a business-critical system, orchestration is part of the correctness boundary. If the system can approve the wrong action, skip validation, repeat a side effect, or lose track of partial completion, the failure is not merely "bad model output." It is a control-flow defect. Control-flow defects deserve a first-class representation.

Koog's graph-based agents fit that need because they model the agent as a structured, type-safe state machine rather than as an unstruc-

tured sequence of prompt interactions. Strategies can be expressed as graphs, custom behavior can be encapsulated in typed nodes, and transitions can be declared explicitly. That architecture aligns the runtime model with the design model. The artifact you review in code is closer to the artifact that executes in production.

There is still room for simpler approaches. A short-lived prototype, a purely conversational assistant with no side effects, or an exploratory internal tool may not need a graph. The decision signal is straightforward: if you do not care which path was taken, whether an intermediate condition was validated, or how a partial failure is recovered, a prompt chain may be adequate. Once you do care, the hidden graph inside the prompt chain should become an actual graph in your program.

Common anti-patterns that signal a prompt chain has outgrown itself

Several symptoms reliably indicate that a linear chain is carrying orchestration work it can no longer manage cleanly.

One is *procedural prompts*. The prompt starts reading like a workflow engine specification: first do this, then if that fails do this other thing, unless condition X, in which case ask for clarification, then decide whether to retry. When a prompt becomes procedural in that way, you are using natural language to emulate a graph.

Another is *string-based business routing*. The model emits route labels or free-form justifications, and the application treats them as authoritative without typed validation. That shifts business control into the least constrained part of the system.

A third is *transcript-as-database*. The team depends on the message history to answer whether a fact is known, a permission was granted, or a side effect already occurred. That is a strong sign that workflow state needs to be made explicit.

A fourth is *retry policy in prose*. If bounded retries, escalation limits, or fallback conditions are described mainly in prompts, the system lacks a reliable recovery surface.

A fifth is *audit by log archaeology*. When engineers have to reconstruct execution from raw prompt and response logs because the runtime has no explicit concept of nodes, transitions, and state changes, the orchestration model is too implicit.

Each of these anti-patterns points to the same corrective move: lift control flow out of prompt conventions and into the graph itself.

The graph is the durable answer to orchestration complexity

As agent systems grow, the hard part is rarely producing one more model call. The hard part is deciding what may happen next, on what evidence, under which constraints, with what recovery behavior, and how you will prove that the system stayed inside those rules. Linear prompt chains blur those decisions into text and glue code. Graph workflows turn them into inspectable software structure.

That is why graph workflows outperform linear prompt chains in business-critical settings. They make the legal execution space explicit. They give state a durable shape. They separate model behavior from orchestration policy. They let you encode branching, validation, and recovery as declared program structure rather than as a collection of hopeful prompt instructions. When predictability becomes a requirement instead of a preference, the graph stops being an implementation detail and becomes the system's operational backbone.

5.2. Modeling Typed Nodes, Edges, and Workflow State

Once you decide that orchestration belongs in an explicit workflow graph, the next design problem is sharper: what exactly are you typing, and what does that type information buy you? Many teams stop too early. They draw boxes and arrows, name a few steps, and assume they now have a disciplined workflow. In practice, a graph without clear contracts can still hide the same ambiguity that made prompt chains brittle. The names are cleaner, but the execution model remains soft. A node takes "some context," produces "some result," and routing decisions still depend on loosely structured text or generic maps. That style improves presentation more than correctness.

Koog's graph DSL is more useful when you treat it as a way to encode execution contracts. A node is not just a labeled step. It is a function boundary with declared input and output types. An edge is not just a visual connector. It states that a particular downstream node may legally receive a particular upstream result. Workflow state is not an ever-growing bag of facts. It is a typed carrier for information that must survive across nodes and subgraphs. When those three ideas line up, the graph becomes a software design discipline rather than a diagramming exercise.

A compact example makes the distinction concrete. The code below is intentionally small, but it shows the core mechanics you need to reason about: typed nodes, typed storage, and edges that wire declared outputs into legal downstream inputs.

```
import ai.koog.agents.core.agent.context.AIAgentStorage
import ai.koog.agents.core.agent.context.createStorageKey
import ai.koog.agents.core.agent.strategy.edge.edge
import ai.koog.agents.core.agent.strategy.node.node
import ai.koog.agents.core.agent.strategy.node.nodeFinish
import ai.koog.agents.core.agent.strategy.node.nodeStart
import ai.koog.agents.core.agent.strategy.strategy
```

```
data class IntakeRequest(
 val requestId: String,
 val email: String,
 val amount: Int
)

data class ValidatedRequest(
 val requestId: String,
 val email: String,
 val amount: Int
)

data class Eligibility(
 val approved: Boolean,
 val reason: String
)

val validatedKey =
 createStorageKey<ValidatedRequest>("validated-request")

val validateRequest =
 node<IntakeRequest, ValidatedRequest> { input ->
 require(input.email.contains("@"))
 require(input.amount > 0)
 ValidatedRequest(
 requestId = input.requestId,
 email = input.email.trim(),
 amount = input.amount
 )
 }

val persistValidated =
 node<ValidatedRequest, Eligibility> { input ->
 storage.set(validatedKey, input)
 if (input.amount <= 500) {
 Eligibility(true, "within-limit")
 } else {
 Eligibility(false, "manual-review")
 }
 }

val finishApproved =
 node<Eligibility, String> { input ->
 check(input.approved)
 "approved:${input.reason}"
 }
```

```
val finishReview =
 node<Eligibility, String> { input ->
 check(!input.approved)
 "review:${input.reason}"
 }

val workflow = strategy<IntakeRequest, String>("refund-intake") {
 edge(nodeStart forwardTo validateRequest)
 edge(validateRequest forwardTo persistValidated)
 edge(
 persistValidated forwardTo finishApproved
 onCondition { result -> result.approved }
 )
 edge(
 persistValidated forwardTo finishReview
 onCondition { result -> !result.approved }
 )
 edge(finishApproved forwardTo nodeFinish)
 edge(finishReview forwardTo nodeFinish)
}
```

You can read this workflow at two levels. At the runtime level, it validates input, stores the normalized request, computes eligibility, and routes to one of two terminal paths. At the contract level, it does something more important. It declares that raw intake data must become a `ValidatedRequest` before any eligibility decision is allowed. It records that validated data in typed storage rather than leaving it implicit in a transcript. It routes on a typed `Eligibility` object rather than on free-form model text. That is the real gain. The graph no longer says only what happens; it encodes what must already be true before each step can happen.

Treat each node as a contract boundary

You get the most leverage from typed workflows when you resist the temptation to define nodes as arbitrary chunks of code. A useful node has a crisp contract along four dimensions: input shape, output shape, side effects, and continuation semantics.

Input shape answers a narrow question: what must a caller provide

for this node to execute meaningfully? If the node needs a validated customer identifier, do not accept a generic request object and hope the right field has already been cleaned. Accept the validated type. That forces normalization to happen upstream and makes illegal entry points harder to express.

Output shape answers the complementary question: what can downstream nodes rely on after this node completes? The output should expose exactly the facts the graph needs to carry forward, not a blob of convenience data. If a node produces an approval decision, capture whether it was approved, why, and any identifiers needed later. Do not return an ambiguous string such as `"ok"` and expect downstream code to infer the rest.

Side effects deserve explicit thought even when the type system cannot encode them directly. A node that performs an external action is not equivalent to a node that normalizes data. If you let both kinds of work accumulate inside a single node, the type signature stays small while the semantic contract becomes opaque. Keep computational nodes, validation nodes, and side-effecting nodes conceptually distinct even when their Kotlin types look similar.

Continuation semantics define what a successful result means for routing. Some nodes simply transform data. Others certify a property that unlocks additional paths. A node that verifies account ownership changes the legal workflow surface; after it succeeds, certain edges become meaningful that were not meaningful before. You want those semantic boundaries reflected in node outputs rather than buried in comments or conventions.

This is why the simple official custom-node form, `node<String, Int> { input -> input.length }`, matters more than it first appears. The syntax is compact, but it forces you to name the transformation boundary directly. That habit scales. Whether the node computes a string

length or executes a policy decision, the code still asks you to declare what goes in and what comes out.

Edges encode legal movement, not just sequence

A typed node graph still fails if the edges are treated casually. Engineers often draw edges as if they were mere sequencing operators: run A, then B, then C. In a workflow with consequential branching, an edge does more than preserve order. It states a legal movement between states of work.

That distinction becomes important as soon as two downstream nodes require different preconditions. Suppose one node can issue a refund only after eligibility is established, while another can request manual review from a less complete record. If both nodes accept the same broad request type, the graph gives you little help. The routing may be visually explicit, but the state model does not constrain misuse. The safer design is to shape outputs so that downstream contracts differ materially. A validated-and-approved object should not be indistinguishable from a merely parsed request.

Koog's routing helpers support this style directly. An edge can be conditional through `onCondition`, and the condition executes against typed output from the upstream node. That lets you move business routing away from string parsing and into explicit predicates over structured data. When the condition operates on a meaningful domain type, your graph starts to express legal movement rather than only possible movement.

You should also be careful about where routing logic lives. If a node both computes a result and hides the routing decision internally, the edge layer becomes decorative. The graph may look explicit while the real control flow stays embedded in node code. Keep the boundary clean: nodes produce information, and edges consume that information to decide where execution may go. There are exceptions for tiny

workflows, but they stay exceptions if you value inspectability.

Workflow state should carry invariants, not leftovers

State is where many agent systems quietly decay. Once a workflow needs information across multiple steps, teams often add a generic context object, a mutable map, or a growing chat transcript and call it done. That keeps development moving for a while, but it destroys the very benefit graph orchestration promised. The runtime can carry anything, so downstream code must defensively assume everything and trust nothing.

Koog's storage model gives you a better primitive. `AIAgentStorage` and `AIAgentStorageKey<T>` let you pass data across nodes and subgraphs with explicit types. The crucial design question is not merely how to store values, but what deserves to become workflow state.

A good rule is simple: put information into workflow state when later routing, validation, recovery, or audit depends on it. If a fact only matters inside one node and never influences legal downstream behavior, keep it local. If a fact must survive node boundaries because another node depends on it or because operators need to inspect it after the run, give it a typed storage key.

That rule yields three practical categories.

Durable control facts belong in workflow state. Examples include validated identifiers, approval outcomes, normalized tool results, retry counts, and correlation IDs. These facts affect what the graph can legally do next or what a support engineer must later explain.

Ephemeral computation details usually stay local. Intermediate parser objects, temporary string fragments, and library-specific helper results often do not need graph-wide visibility. Promoting them to workflow state makes traces noisier and state contracts weaker.

Externalized long-lived memory belongs elsewhere. User memory, retrieval corpora, and durable checkpointing systems have different lifecycles and consistency requirements. They matter to the overall architecture, but they are not substitutes for workflow state. Workflow state should stay narrow enough to explain the current run.

That separation is one of the main discipline points in graph design. If you put too little into workflow state, downstream behavior depends on reconstructing facts from text or from hidden local variables. If you put too much into it, every node starts depending on a giant mutable object whose invariants nobody can state cleanly.

Type safety is a control mechanism

In many codebases, type safety gets framed as a developer convenience: better autocomplete, cleaner refactors, fewer runtime casts. Those benefits are real, but they are not the most important reason to type your workflow graph. In orchestration-heavy systems, type safety is a control mechanism.

A typed node prevents an entire class of invalid execution simply by refusing to accept the wrong shape of data. A typed storage key prevents accidental reuse of unrelated values under the same conceptual name. A typed edge condition forces routing decisions to inspect structured results rather than fragile string conventions. Each of those restrictions narrows the legal state space of the running system.

That matters more in agent workflows than in ordinary request-response code because model output expands ambiguity. If a model proposes a route, a tool call, or a policy interpretation, you need a place to constrain that proposal before it acquires business force. Types give you one layer of that constraint. They do not replace validation or policy logic, but they make it harder for unchecked text to leak directly into control flow.

The effect compounds during maintenance. If you split one node into two, the compiler can expose exactly which edges and downstream consumers now need adjustment. If you add a required field to a validated type, you learn which node contracts are incomplete. If you narrow an eligibility model from open text to a closed set of structured reasons, the routing layer becomes correspondingly more robust. Without types, those changes spread by convention and are discovered late through broken runs.

Use transformation boundaries deliberately

Not every edge can connect matching types directly, and that is healthy. A graph often needs to adapt one output into a narrower or differently shaped downstream input. Koog exposes `transformed {... }` for that purpose. The design lesson is not merely that type adaptation is possible, but that adaptation should be explicit and local.

When a node emits a rich result object and the next node needs only one normalized field, perform that projection at the edge boundary rather than forcing the downstream node to understand the whole upstream payload. That keeps dependencies narrow. It also makes the graph more legible. You can tell where information is intentionally reduced, renamed, or reinterpreted.

Use this carefully. A transformation is a contract boundary, not a dumping ground for arbitrary business logic. If the adaptation becomes complex enough to have failure modes, validation rules, or side effects of its own, promote it to a node. Keep `transformed {... }` for lightweight shape adaptation that clarifies the handoff between adjacent workflow steps.

A small graph can already express meaningful constraints

The moment you combine typed nodes, conditional edges, and typed storage, the graph starts to enforce design intent rather than merely

document it. The refund example above has only a few nodes, but it already excludes several invalid paths:

- A request cannot reach eligibility evaluation before passing structural validation.
- The validated request is stored in typed form before any approval path is chosen.
- The approved and manual-review terminals are chosen from a typed `Eligibility` result rather than from ambiguous text.
- Terminal nodes assert the branch invariant they depend on, making branch misuse fail fast during development and testing.

Those are modest constraints, yet they eliminate a large class of production bugs that otherwise surface as vague workflow anomalies. The graph is doing real control work.

Granularity is a design decision, not a purity test

Once you start modeling contracts explicitly, it is easy to overcorrect and turn every micro-step into a separate node. That produces a graph that is technically typed and practically unreadable. The goal is not maximum fragmentation. The goal is to surface control boundaries that matter.

A node deserves to exist when one or more of the following is true:

- it changes the legal downstream workflow surface;
- it performs a meaningful validation or normalization step;
- it introduces or depends on a side effect;
- it deserves independent observation or testing;

- it creates a reusable boundary that will likely survive refactoring.

A step does *not* necessarily deserve its own node just because you can give it a name. Splitting trivial computations into separate graph elements often dilutes the signal. If two operations are inseparable from the viewpoint of routing, testing, and observability, keep them inside one node. For example, trimming whitespace and normalizing case for an identifier usually belong together unless one of those operations has distinct policy meaning.

The same caution applies to state shape. Do not create a new workflow type for every local variation in data unless the variation changes what the graph is allowed to do. Types should encode business-relevant distinctions, not every implementation detail. The right granularity makes illegal movement hard and legal movement obvious.

Keep business invariants in types and predicates, not in comments

Comments can describe invariants, but they cannot enforce them. If a refund path requires both validated identity and a positive policy decision, encode those facts in your data model and edge predicates. Do not rely on notes such as "caller must ensure validation happened first." If callers can violate the assumption, they eventually will.

This does not mean you need a maximally rich algebraic model for every workflow. It means you should identify the invariants that materially affect correctness and represent them in a way the code can check. A boolean approval flag may be enough in one workflow. Another may need a sealed domain model separating *Approved*, *Rejected*, and *NeedsReview*. Choose the narrowest representation that makes the invalid states difficult to express.

The same principle guides storage keys. If you store values under generic names such as `"result"` or `"status"`, you are reintroducing

stringly typed ambiguity through a side door. Prefer keys whose names and types reflect the business invariant they carry. You want a future maintainer to infer why that value exists and which nodes may legitimately depend on it.

Typed graphs improve traces because the runtime matches the design

Observability becomes easier when the runtime model mirrors the design model. A trace built from node executions, typed inputs, typed outputs, storage writes, and edge transitions is much easier to interpret than a trace built from mixed prompt logs and helper-function messages. You can ask precise questions: which node produced the eligibility result, what validated request was in storage at that point, which condition routed execution to review, and whether an unexpected transition occurred.

That clarity helps both debugging and governance. During debugging, you can isolate whether the fault came from node logic, malformed state, or an incorrect edge condition. During governance review, you can show that certain routes are unreachable without specific typed preconditions. The graph becomes a defensible artifact rather than a conceptual sketch.

Common modeling mistakes erode the benefit quickly

Several mistakes repeatedly undermine typed workflow designs.

The unbounded state object. Teams create a giant context model and keep adding fields until every node can read and write everything. This preserves compile-time typing while destroying semantic discipline. The graph appears structured, but real dependencies are hidden inside a shared mutable object.

Stringly typed payloads in typed clothing. A node returns a typed wrapper around a raw string blob, and downstream nodes still parse

conventions out of it. The type system is present, but the contracts remain informal.

Nodes that both compute and route opaquely. If a node decides internally which next step should run and emits a generic result, the edge layer stops expressing actual control flow. Keep routing logic visible where possible.

Duplicated business logic in edges and nodes. If a node validates the same approval rule that an edge predicate also checks, the workflow becomes harder to evolve. Put each invariant in one clear place, then route on the resulting structured state.

Over-modeling every transient condition. Excessive type proliferation can make the workflow harder to read than the original prompt chain. Model the distinctions that change legality, observability, or recovery, and keep incidental variation local.

These mistakes are easier to avoid if you keep one question in front of you during design: what facts must be true for this transition to be legal? If the answer is vague, the model is too loose. If the answer requires understanding ten unrelated state fields, the model is too broad.

Design the graph so refactoring pressure stays local

One practical test of a good typed workflow is how refactoring pressure propagates. When you change a policy rule, you want the impact to stay close to the node or edge that owns the rule. When you normalize a new field, you want the compiler to point at the specific downstream contracts that now need attention. If a simple policy change forces edits across every node because they all share one broad request object, the workflow is not sufficiently decomposed.

This is where typed nodes and explicit state pay long-term dividends. They create natural seams. You can replace a node implementation, split a node into two, or refine a branch output without rewriting the

entire graph. The graph is still a connected system, but the contracts localize change. That is the difference between a workflow that can evolve under production pressure and one that slowly hardens into a fragile tangle.

A well-shaped graph does not eliminate complexity. It places complexity where you can inspect it, test it, and keep it honest. Typed nodes define work units with real contracts. Typed edges make legal progression explicit. Workflow state carries the facts that matter to control and audit. When those parts reinforce each other, the graph stops being merely descriptive and starts acting as a correctness boundary for the agent system.

5.3. Implementing Branching, Retries, and Compensating Paths

A workflow becomes operationally interesting the moment the happy path stops being the only path. A request arrives with incomplete data, a model proposes an ambiguous route, a provider times out, validation fails after a tool already changed external state, or a second attempt succeeds only because the first attempt partially succeeded in a way your code did not record. These are not exceptional edge cases in production systems. They are the normal shape of execution once an agent interacts with external services and business rules.

That is why failure handling belongs in the graph itself. If branching, retries, and cleanup logic live only in prompts or in scattered exception handlers, the workflow stops being inspectable at the exact moment you most need inspection. A predictable agent does not merely *have* recovery behavior. It makes recovery behavior explicit, bounded, and reviewable.

A concrete graph pattern makes that visible. The example below mod-

els a refund workflow with four distinct outcomes: approve automatically, request manual review, retry a transient failure, or compensate after a partial side effect. The code uses typed nodes, typed storage, and explicit edge conditions so that branch selection is driven by structured state rather than by free-form text.

```
import ai.koog.agents.core.agent.context.createStorageKey
import ai.koog.agents.core.agent.strategy.edge.edge
import ai.koog.agents.core.agent.strategy.node.node
import ai.koog.agents.core.agent.strategy.node.nodeFinish
import ai.koog.agents.core.agent.strategy.node.nodeStart
import ai.koog.agents.core.agent.strategy.strategy

data class RefundRequest(
 val requestId: String,
 val customerId: String,
 val amount: Int
)

sealed interface RefundDecision {
 data class Approved(
 val request: RefundRequest
 ): RefundDecision

 data class NeedsReview(
 val request: RefundRequest,
 val reason: String
 ): RefundDecision
}

sealed interface ChargeResult {
 data class Success(
 val externalId: String
 ): ChargeResult

 data class TransientFailure(
 val message: String
 ): ChargeResult

 data class PartialCommit(
 val externalId: String
 ): ChargeResult
}

val retriesKey = createStorageKey<Int>("refund-retries")
val chargeIdKey = createStorageKey<String>("refund-charge-id")
```

```
val decideRefund =
 node<RefundRequest, RefundDecision> { input ->
 if (input.amount <= 500) {
 RefundDecision.Approved(input)
 } else {
 RefundDecision.NeedsReview(
 request = input,
 reason = "amount-exceeds-auto-limit"
 )
 }
 }

val executeRefund =
 node<RefundDecision.Approved, ChargeResult> { input ->
 val priorRetries = storage.get(retriesKey) ?: 0
 when (priorRetries) {
  0 -> ChargeResult.TransientFailure("gateway-timeout")
  1 -> ChargeResult.PartialCommit("txn-4821")
  else -> ChargeResult.Success("txn-4821")
 }
 }

val recordRetry =
 node<ChargeResult.TransientFailure, RefundDecision.Approved> {
 val current = storage.get(retriesKey) ?: 0
 storage.set(retriesKey, current + 1)
 RefundDecision.Approved(
  RefundRequest("req-unknown", "cust-unknown", 0)
 )
 }

val recordPartial =
 node<ChargeResult.PartialCommit, ChargeResult.PartialCommit> {
 storage.set(chargeIdKey, it.externalId)
 it
 }

val compensateRefund =
 node<ChargeResult.PartialCommit, String> { input ->
 "compensated:${input.externalId}"
 }

val finishApproved =
 node<ChargeResult.Success, String> { input ->
 "approved:${input.externalId}"
 }

val finishReview =
```

```
 node<RefundDecision.NeedsReview, String> { input ->
 "review:${input.reason}"
 }

val workflow = strategy<RefundRequest, String>("refund-flow") {
 edge(nodeStart forwardTo decideRefund)

 edge(
  decideRefund forwardTo executeRefund
  onCondition { it is RefundDecision.Approved }
  transformed { it as RefundDecision.Approved }
 )

 edge(
  decideRefund forwardTo finishReview
  onCondition { it is RefundDecision.NeedsReview }
  transformed { it as RefundDecision.NeedsReview }
 )

 edge(
  executeRefund forwardTo recordRetry
  onCondition { it is ChargeResult.TransientFailure }
  transformed { it as ChargeResult.TransientFailure }
 )

 edge(
  executeRefund forwardTo recordPartial
  onCondition { it is ChargeResult.PartialCommit }
  transformed { it as ChargeResult.PartialCommit }
 )

 edge(
  executeRefund forwardTo finishApproved
  onCondition { it is ChargeResult.Success }
  transformed { it as ChargeResult.Success }
 )

 edge(recordRetry forwardTo executeRefund)
 edge(recordPartial forwardTo compensateRefund)
 edge(finishApproved forwardTo nodeFinish)
 edge(finishReview forwardTo nodeFinish)
 edge(compensateRefund forwardTo nodeFinish)
}
```

This example is deliberately compact, but it exposes the architectural move that matters. The graph does not treat failure as an out-of-band

concern. A transient failure is a typed output. A partial commit is a typed output. Compensation is a named path. Manual review is a named path. The legal execution surface includes unhappy paths by design.

The example also shows an important limitation that you should keep in view: Koog's graph model clearly supports loops and conditional branching, but the retrieved documentation does not confirm a standalone built-in retry-policy DSL for arbitrary graph nodes. Until you verify a newer primary source, model retries conservatively as explicit graph loops, node-local logic, or bounded repetition inside higher-level graph constructs. That approach is not a workaround; it is often the clearer design anyway, because the retry behavior remains visible in the topology.

Branching starts with controlled interpretation

Most branch errors in agent systems do not come from syntax mistakes. They come from letting the model improvise control decisions that belong to the business domain. If a workflow asks a model to "decide whether to escalate" and then directly routes on the returned prose, the system has effectively delegated policy to unconstrained text generation. That can work in a prototype. It is a liability in production.

You want a narrower pattern. Let the model produce information, proposals, or classifications inside a typed boundary, then route on validated outputs or explicit predicates. The decision point stays in the graph even if a model contributed evidence to it.

There are three common sources of branching, and they need different handling.

Deterministic business branching should come from explicit predicates over validated state. Examples include amount thresholds, account status, approval flags, presence of required documentation,

and tool-returned status codes after normalization. Use graph edges for this because the route is governed by domain rules, not by model creativity.

Model-suggested branching is appropriate when the model performs interpretation that would be expensive to encode manually, such as classifying a user request into one of a controlled set of intents. Even then, do not route on raw text. Route on a typed, validated representation of the model's proposal. If the proposal does not fit the expected schema or confidence regime, add an explicit review or repair path.

Operational branching responds to runtime conditions such as timeout, malformed output, rate limiting, or partial side effects. These conditions are not semantic interpretations of user intent. They are control-flow events and should be modeled as such.

When these categories are mixed together, workflows become opaque. A timeout and a policy rejection are treated as equivalent "errors." A model classification result is handled by the same fallback path as a transport failure. You lose the ability to reason about what kind of problem occurred and what kind of response is justified. Typed branch outputs keep those categories separate.

Model branch conditions around failure classes, not generic error flags

A single `failed: Boolean` field is almost always too weak for production control flow. It tells you that something went wrong, but not what kind of wrongness occurred. Recovery decisions depend on that distinction.

A useful failure classification usually separates at least these cases:

- **Transient infrastructure failure:** timeout, temporary unavailability, provider throttling, dropped connection.

- **Semantic failure:** malformed model output, invalid tool payload, failed validation, unsupported request shape.
- **Business rejection:** request denied by policy, approval not granted, precondition not satisfied.
- **Partial side effect:** an external action may have happened, but the local workflow cannot yet prove final success.

Each class should drive different routing. Transient failures may justify retry. Semantic failures often justify repair, normalization, or escalation. Business rejections should usually terminate or route to review without retrying. Partial side effects require the most care because the system may now need compensation rather than repetition.

If you model all of these as the same generic failure signal, your graph cannot express recovery policies with enough precision. The result is usually over-retry, silent masking, or accidental double execution.

Retries should be explicit, bounded, and stateful

Retries are the easiest recovery mechanism to add and the easiest to misuse. A retry can improve availability when the underlying problem is transient. It can also amplify load, duplicate side effects, hide systemic defects, and make debugging harder if the workflow does not record what happened. Treat retries as graph behavior, not as a convenience wrapper.

A useful retry design answers five questions explicitly.

What kind of failure is retryable? Only failures with a plausible chance of succeeding on a later attempt should loop. A validation error caused by malformed business data is not improved by repetition. A provider timeout might be.

Where is the retry count stored? The graph needs visible state for retry count or attempt history when the count affects legal movement.

If the count changes whether you escalate, stop, or compensate, record it in workflow state rather than burying it inside a local library callback.

What is the retry bound? Unbounded retries are almost never acceptable in a business workflow. They turn a temporary defect into hidden latency or resource exhaustion. A bounded loop makes the limit part of the design.

What happens after the last allowed retry? A retry path without a terminal failure path is not a policy. It is a delay. After the final attempt, route somewhere explicit: manual review, failure terminal, fallback provider, or compensation path, depending on the operation.

Is the retried node safe to re-execute? This question matters more than the retry count. If re-execution can duplicate an external side effect, you need an idempotency strategy, a reconciliation step, or compensation logic before you can safely retry.

In Koog, the conservative and clear way to express retries is often to route transient failure outputs to a node that records attempt state and then loops back to the action node. That keeps the retry policy visible in the graph. In verification-oriented flows, bounded repetition may also live inside a subgraph configuration such as `assistantResponseRepeatMax`, but that is a narrower pattern than a general retry framework and should be described accordingly.

Decide whether retry belongs inside a node or in the graph

Not every retry needs a visible edge. Some retries are transport-local implementation details. If an HTTP client transparently retries an idempotent GET after a dropped connection, exposing that as a graph branch may add noise without meaningful control value. Other retries absolutely belong in the graph because they alter business-observable behavior.

Use a graph-visible retry when one or more of the following is true:

- the retry count affects routing or escalation;
- operators must be able to explain why a request took longer or changed path;
- the retried action has business significance or side-effect risk;
- different failure classes route differently after retry exhaustion;
- audit or policy requirements demand explicit control of repetition.

Keep the retry local to a node when the behavior is purely mechanical, idempotent, and does not change the workflow's externally meaningful path. Even then, document the assumption. Hidden retries can surprise you during incident analysis if they distort latency or provider usage patterns.

Fallback paths are not the same as retries

A fallback is an alternate strategy, not another attempt at the same strategy. That distinction matters because fallbacks often change correctness assumptions, output quality, or governance posture.

For example, if a primary enrichment service times out, a retry may ask the same service again. A fallback might instead use a cached record, a secondary provider, or a reduced-capability manual-review path. Those options have different semantics. A cache-based fallback may lower freshness. A secondary provider may return differently shaped data. A manual-review fallback changes latency and accountability. If you treat fallback as just "retry somewhere else," the graph hides these trade-offs.

Model fallback as a distinct branch when the alternate path changes guarantees or downstream interpretation. Give the fallback path its own validation and terminal semantics. Do not quietly substitute one

strategy for another unless the substitution is contractually acceptable and operationally visible.

Partial completion changes the problem from recovery to consistency

The hardest workflows are not the ones that fail before doing anything. They are the ones that may have changed the outside world before the workflow loses certainty. A payment gateway times out after accepting the request. A ticketing system creates a record but the confirmation response is lost. A refund is issued, but the local node crashes before persisting the external identifier. In these cases, "try again" may create duplication, and "give up" may leave the system inconsistent.

This is where compensating paths become necessary. Compensation is not a nicer exception handler. It is a deliberate branch that attempts to restore business consistency after a side effect has already crossed a meaningful boundary.

A useful way to think about this borrows from saga design. Some steps are **retryable transactions**: if they fail transiently, repeating them is safe and sensible. Some steps are **compensable transactions**: if they complete and later logic fails, you can execute a compensating action that semantically undoes the business effect. Some systems also have a **pivot** moment: after a certain point, the workflow moves from reversible preparation into committed action, and the downstream design must assume that the business state has materially changed.

This vocabulary helps you reason about graph topology. Preparation and validation steps usually live before the pivot. Retryable actions need explicit loop conditions. Compensable side effects need a stored identifier and a compensation branch. Irreversible actions often need stronger gating before execution because the graph cannot truly undo them afterward.

Store the facts compensation will need before you need them

A compensation path fails surprisingly often for a simple reason: the workflow did not preserve the data required to execute it. If a tool call returns an external transaction identifier, store it as soon as the graph believes the side effect may have occurred. Do not assume you can recover it later from logs or by replaying the same call.

This is where workflow state design directly affects reliability. The graph should persist compact, typed compensation facts such as external IDs, timestamps, prior status, or normalized provider references at the moment they become relevant. These values are not implementation leftovers. They are part of the recovery contract.

In the example above, the partial-commit path stores `chargeIdKey` before entering compensation. That is a small pattern with large consequences. A compensation node without the needed identifier is not really a recovery mechanism; it is a best-effort apology.

Compensation should be first-class and testable

If compensation matters, give it the same design dignity as the happy path. Name the node. Type its input. Record the exact state it depends on. Give it explicit terminal behavior. Test it independently.

That last point is often neglected. Teams test the success path thoroughly, then assume compensation will work because the code is short. In reality, compensation logic is exercised less often, depends on already-damaged state, and frequently interacts with systems under stress. It deserves stronger tests, not weaker ones.

A practical test matrix for compensation usually includes:

- compensation after a confirmed partial side effect;
- repeated compensation attempts for idempotency or safe dedu-

plication;

- compensation when the external system reports the action was already reversed;
- failure of the compensation itself, including escalation behavior;
- observability of the compensation branch in traces and final status.

If you cannot describe how compensation behaves under those conditions, the workflow does not yet have a reliable cleanup story.

Do not mix business recovery with transport recovery

A subtle anti-pattern appears when low-level client retries, business retries, and compensation logic all pile into the same node without clear boundaries. The node calls an external API, the HTTP client retries twice, the node retries once more if it sees a timeout, and the workflow retries again if the output is still inconclusive. Meanwhile, the same node also decides whether the business request should be escalated or rejected.

This design is difficult to reason about because three recovery layers are interleaved:

1. **Transport recovery** handles mechanics such as transient network interruptions.
2. **Operation recovery** handles whether the domain action itself should be attempted again.
3. **Business recovery** handles whether the workflow should escalate, fallback, compensate, or terminate.

Keep these layers separate enough that you can explain them. Let the client library handle simple idempotent transport concerns. Let the

graph own business-visible retry and fallback policy. Let compensation handle semantic recovery after side effects. When all three concerns collapse into one opaque block, operators cannot tell whether the system is resilient or merely repetitive.

Branch visibility is a governance feature

It is tempting to hide complexity inside nodes in the name of a cleaner graph. Resist that instinct when the complexity affects policy or accountability. A branch that determines whether money moves, whether a user receives an automatic answer, whether a failed action is retried, or whether a compensating action is issued should usually be visible in the workflow topology.

Visible branches improve more than readability. They create stable hooks for monitoring, testing, and review. You can count how often the transient-failure loop runs. You can alert on compensation frequency. You can inspect whether manual-review paths spike after a model or tool change. None of that requires understanding the internals of every node if the graph itself names the important control points.

Decision signals for branch, retry, fallback, or compensation

When choosing a recovery shape, the most useful decision signals are usually these:

Failure semantics. If the problem is transient and no side effect escaped, retry is a candidate. If the problem is semantic or policy-based, route to repair or review instead.

Idempotency. If repeating the action can safely produce the same result, retries are cheaper. If repetition can duplicate value transfer or external mutation, you need stronger guards, reconciliation, or compensation.

Business reversibility. If the action can be semantically undone,

compensation is viable. If it cannot, move more validation and approval before the pivot.

Observability needs. If operators must explain the path, keep the branch explicit. Hidden recovery is expensive to debug even when it works.

Cost of masking defects. Fallback can improve availability, but it can also hide broken primaries and degrade output quality quietly. Use it when the alternate path is genuinely acceptable, not just because it avoids a visible error.

These criteria are more useful than any generic rule such as "always retry three times" or "always fall back to manual review." The right topology depends on the semantics of the action and the cost of getting it wrong.

Common failure patterns map naturally onto graph shapes

Certain recurring patterns are worth recognizing because they suggest stable graph forms.

A **validate-then-act** pattern is appropriate when you can cheaply reject or reroute invalid input before touching external systems. Keep the validation node separate so the graph makes the precondition explicit.

An **attempt-record-retry** loop fits transient failures when retry count changes downstream behavior. The recording node updates state, and the loop remains bounded and inspectable.

A **attempt-detect-compensate** path fits partial completion. The first action records evidence of possible external mutation, and the compensation branch terminates in a safe, reviewable state rather than pretending nothing happened.

A **classify-and-review** branch fits model-suggested routing when confidence or policy requires human oversight for certain classes. The

graph should route on a structured classification output, not directly on conversational text.

These patterns are not framework magic. They are disciplined uses of nodes, state, and edges to make failure handling part of the executable design.

A predictable workflow makes bad paths legible

The happy path tells you whether a workflow can succeed. The unhappy paths tell you whether it can be trusted. When branching, retries, fallbacks, and compensations are explicit graph elements, the system stops treating failure as hidden turbulence inside prompts and helper code. It names the conditions, records the state, bounds the repetition, and gives recovery its own topology.

That is the practical standard to aim for. A workflow should make it obvious which decisions are business decisions, which failures are retryable, where reversibility ends, and what the system does after uncertainty enters the run. Once those paths are encoded directly in the graph, recovery behavior becomes something you can inspect, test, and defend rather than something you merely hope the agent will improvise correctly.

5.4. Orchestrating Tools, Validation, and Human Gates Inside Workflow Graphs

The workflow stops being abstract once it touches a system outside the model. A search index must be queried, a CRM record must be fetched, a compliance rule must be enforced, or a refund request must wait for human approval before anything irreversible happens. At that point, "the agent" is no longer a single reasoning component. It is an execution topology composed of model steps, tool calls, validators, and

sometimes people. If you let any one of those components coordinate the others implicitly, you lose the control surface that made the graph valuable in the first place.

The practical rule is simple: treat model invocations, tool execution, validation, and approval as separate orchestration concerns, even when they cooperate tightly at runtime. Tool design belongs elsewhere. What matters here is how you sequence those capabilities so that each step has a clear contract, the graph owns the legal progression, and traces show not just what the model said, but what the system actually did.

Koog exposes this orchestration model directly. Official graph-oriented APIs include predefined nodes such as `nodeLLMRequest()`, `nodeExecuteTools()`, and `nodeLLMSendToolResults()`, which make the tool loop explicit instead of burying it inside a prompt. That is the right starting point because it separates three distinct responsibilities:

- the model decides whether tool calls are needed and formulates them;
- the runtime executes those tool calls under explicit policy;
- the model receives tool results and continues generation with the new evidence.

A minimal orchestration slice already shows why this matters.

```
import ai.koog.agents.core.agent.strategy.edge.edge
import ai.koog.agents.core.agent.strategy.node.nodeFinish
import ai.koog.agents.core.agent.strategy.node.nodeStart
import ai.koog.agents.core.agent.strategy.node.nodeExecuteTools
import ai.koog.agents.core.agent.strategy.node.nodeLLMRequest
import ai.koog.agents.core.agent.strategy.node.nodeLLMSendToolResults
import ai.koog.agents.core.agent.strategy.strategy

val llm = nodeLLMRequest()
val executeTools = nodeExecuteTools()
val sendResults = nodeLLMSendToolResults()
```

```
val toolWorkflow = strategy<String, String>("tool-orchestration") {
 edge(nodeStart forwardTo llm)
 edge(llm forwardTo executeTools onToolCalls { true })
 edge(executeTools forwardTo sendResults)
 edge(sendResults forwardTo executeTools onToolCalls { true })
 edge(sendResults forwardTo nodeFinish onTextMessage { true })
}
```

The loop is explicit. The model does not "just use tools." It emits tool calls, the graph routes to a tool execution node, results come back, and the model either asks for more tools or produces a terminal text response. That may look obvious, but it corrects a persistent design mistake: assuming that tool use is a property of the model rather than a property of the workflow. Once you make the loop visible, you can constrain it, instrument it, test it, and add validation or approval where needed.

Keep the orchestration boundary sharp

A tool is an action surface. A validator is a correctness surface. A human gate is an accountability surface. A model node is an interpretation or synthesis surface. If you allow one of those layers to absorb the others, the workflow quickly becomes opaque.

The most common collapse is to ask the model to do everything: interpret intent, choose tools, decide whether tool outputs are valid, determine whether a human is needed, and generate the final action proposal. That looks elegant in a prototype because the prompt is short and the graph is small. It fails under operational pressure because each responsibility has different failure modes and different review requirements.

A better pattern is to let each component do one kind of work and no more than that:

- **Model nodes** interpret ambiguous input, synthesize tool argu-

ments, summarize results, or draft proposals.

- **Tool nodes** perform external actions or data retrieval.
- **Validation nodes** normalize, verify, and reject malformed or policy-incompatible results.
- **Human gates** authorize, deny, or amend a proposal before the graph crosses a meaningful business boundary.

This separation does not make the workflow slower by definition. It makes the control boundaries visible. That visibility is what lets you optimize safely later.

A model step should propose, not silently commit

Many failures in tool-using systems happen because a model node is allowed to blur proposal and commitment. The model infers intent, constructs arguments, and implicitly decides that the side effect should occur. In a high-stakes path, that is too much authority for one step.

The safer pattern is proposal first, commitment later. Let the model node produce a structured proposal or tool-call intent. Then route that proposal through validation and, if necessary, through a human or policy gate before any side-effecting tool executes. Even when the graph ultimately allows automatic execution, you still benefit from separating proposal from commitment because the validator now has a stable object to inspect.

This is particularly important when the model is asked to map natural language into operational parameters. A request like “refund the duplicated charge from last month” may require customer identification, transaction disambiguation, amount normalization, policy checks, and justification capture. You do not want the model’s first interpretation to become an action without intermediate scrutiny.

Validation belongs in the graph, not in prompt wording

Validation is frequently under-modeled. Teams write prompts such as "only return valid JSON" or "ensure the selected record matches the user" and count that as a validation strategy. Those instructions may improve behavior, but they do not create an enforceable contract. If the model drifts, the workflow has no explicit correctness barrier.

A proper validator node has a narrower mission. It receives typed output from a model or tool step and confirms that downstream assumptions are justified. That can include schema normalization, range checks, identifier consistency, business rule enforcement, or confidence gating. The key point is that validation changes the legal workflow surface. After a validator passes, a downstream node may legitimately execute. Before it passes, that node should not be reachable.

You should also keep different validation purposes distinct.

Structural validation checks whether the output is well formed enough to consume. Examples include parseability, required fields, and type conversion.

Semantic validation checks whether the output makes sense in the domain. Examples include whether a customer ID exists, whether a date range is admissible, or whether the selected account belongs to the current tenant.

Policy validation checks whether execution is allowed. Examples include approval thresholds, forbidden operations, or separation-of-duties rules.

These layers often appear together, but they are not interchangeable. A structurally valid tool result can still be semantically wrong. A semantically correct proposal can still violate policy. If you collapse all validation into one vague gate, operators lose the ability to tell whether the issue was malformed data, a domain mismatch, or a business prohibition.

Tool orchestration is a loop with checkpoints

The canonical tool loop is straightforward: request, execute, return results, continue. The engineering challenge is not understanding the loop. It is deciding where to place checkpoints inside it.

At minimum, you usually need to answer these questions:

- Should the model be allowed to call any registered tool, or only a constrained subset?
- Do tool arguments require validation before execution?
- Should multiple tool calls execute serially or in parallel?
- Must tool results be normalized before the model sees them again?
- Are there points where the graph should stop and ask for approval before continuing?

Koog's predefined nodes make the loop explicit enough that you can answer these questions in topology rather than in hope. If the model proposes tool calls, the graph can route to execution. If the execution phase needs a limited tool set, you can enforce that at the subgraph or registry boundary rather than relying on prompt discipline. If the model should receive only validated tool results, place a validator node between `nodeExecuteTools()` and `nodeLLMSendToolResults()`.

That last move is especially useful. Raw tool outputs often contain provider-specific structure, optional fields, or partial data that a downstream model can misinterpret. A normalization step can convert those outputs into a stable internal representation before the LLM sees them again. That reduces prompt burden and makes the system less sensitive to provider quirks.

Use parallel tool execution selectively

Koog supports parallel execution through `nodeExecuteTools(parallel = true)`. This is an operationally important capability, but it should not be treated as a default speed knob. Parallel tool execution changes both latency and correctness assumptions.

It improves latency when the requested tools are independent and the results do not need ordered interpretation. A common example is enriching a case with data from several read-only systems: account metadata, fraud score, recent support history, and shipping status. If those calls do not depend on one another, parallel execution can significantly reduce end-to-end time.

It becomes risky when tool calls have hidden coupling. Two tools may mutate overlapping state, compete for a scarce rate limit, or produce results that require sequential conflict resolution. Parallel execution also complicates failure handling. If one of three tools fails, does the graph wait, cancel the others, continue with partial results, or route to a recovery path? Those are workflow questions, not runtime trivia.

Use parallel execution when these conditions hold:

- tool calls are logically independent;
- the tools are safe to run concurrently;
- partial completion semantics are understood;
- result aggregation is explicit and testable;
- the latency gain is worth the extra recovery complexity.

If those conditions are not clear, prefer staged execution. A slightly slower but inspectable workflow is usually cheaper than a fast one whose partial-failure behavior is ambiguous.

Subgraphs are a strong boundary for sensitive phases

As workflows become larger, least privilege becomes part of orchestration rather than just a security slogan. Not every phase of a workflow should have access to every tool. A model step that drafts a recommendation may need retrieval tools but should not see side-effecting admin actions. A verification phase may need only checking tools and no mutation capability at all.

Koog subgraphs give you a useful way to express that boundary. A subgraph can be a self-contained unit with its own input, output, and limited tool surface. That is more than a code-organization convenience. It lets you narrow authority around sensitive workflow phases.

This matters operationally. If a planning or enrichment phase has access to powerful tools it does not strictly need, prompt errors and model drift have a larger blast radius. If a verification-oriented subgraph can access only the tools required to inspect a proposal, you have encoded least privilege directly into workflow topology.

Use subgraphs when you need one or more of the following:

- a bounded tool set for a specific phase;
- a reusable mini-workflow with stable contracts;
- an isolated approval or verification region;
- a cleaner trace boundary around a logically distinct activity.

This is also the right place to contain specialized orchestration patterns such as critique or verification loops. The graph stays readable at the top level while sensitive logic remains explicit and bounded.

Verification nodes should police outputs, not restate prompts

A verification step is valuable only if it has a different job from the generating step. If the same model prompt both generates an answer and "verifies" it using the same context and incentives, the verification layer may add little beyond latency. Effective verification changes either the information surface, the tool surface, or the acceptance criteria.

Koog exposes verification-oriented orchestration through `subgraphWithVerification()`, which guarantees a final verification tool call and produces a structured `CriticResult`. That pattern is useful because it turns critique into a runtime contract rather than an optional prompt suggestion. The graph knows a verification step must happen before the subgraph returns.

This gives you a disciplined place for checks such as:

- does the proposed action match policy constraints;
- do cited records actually support the generated explanation;
- are required fields present and internally consistent;
- should the result be accepted, revised, or escalated.

The benefit is not only correctness. It is inspectability. When verification returns a structured critic result, operators can distinguish "proposal failed policy verification" from "tool execution failed" or "user input was ambiguous." Those distinctions are essential for incident response and product iteration.

Human gates belong before irreversible boundaries

A human gate is expensive in latency and staffing, so you should place it deliberately. Many systems add human review too late. The workflow gathers data, synthesizes a proposal, executes side effects, and only then exposes the result to a reviewer. At that point, review is descriptive rather than preventive.

The right placement depends on what the human is meant to control.

If the human is validating interpretation, place the gate after proposal generation and validation but before any external mutation. If the human is approving risk, place the gate after relevant evidence is assembled and normalized, but still before the pivot transaction. If the human is auditing already-completed behavior, the gate may come later, but then you should be honest that this is oversight, not approval.

A useful human-gate contract usually includes:

- the proposed action in structured form;
- the evidence or summary justifying the proposal;
- the decision options available to the reviewer;
- the effect of each decision on the graph;
- the timeout or abandonment behavior if no decision arrives.

Without this structure, human review becomes a detached UI event rather than part of the workflow. The graph should be able to represent approval, rejection, request-for-clarification, and timeout as distinct transitions.

Model the human gate as a node or subgraph boundary

Koog documentation retrieved for this work does not clearly expose a dedicated built-in human-interrupt node. Treat human approval conservatively as a workflow pattern implemented with custom nodes or subgraphs and an external approval system. That is not a limitation in the design sense. It simply means you should model the gate explicitly in your own graph rather than assuming a magic runtime primitive.

The important point is to give the gate first-class status. A custom approval node can pause progression by emitting a request to an external

approval service, persisting the proposal identifier and relevant state, and later resuming into an approval outcome node. The exact integration mechanics depend on your surrounding platform, but the orchestration principle stays stable: the graph owns the decision boundary, not the UI and not the prompt.

Be careful with re-execution semantics if your runtime can resume work after interruption or failure. A resumed approval path may re-enter a node, so side-effecting steps before the gate should be idempotent or protected against replay. That concern becomes especially important in durable execution systems, even if the precise runtime behavior is handled outside the graph DSL itself.

Avoid god nodes that mix tools, validation, and routing

One of the fastest ways to erode inspectability is to build a single node that does all of the following:

- asks the model what to do;
- executes one or more tools;
- parses and validates the results;
- decides whether policy allows continuation;
- maybe asks a human for input;
- and returns a final answer.

This node may be small in source lines, but it has no useful contract. Its input is broad, its internal state is hidden, and its output compresses multiple control decisions into one opaque result. The graph around it becomes ceremonial because the real orchestration has moved inside the node.

Split these concerns unless you have a strong reason not to. If you need one node for performance or implementation convenience, document exactly which responsibilities it owns and which invariants it guarantees. Otherwise, the system will gradually recreate the prompt-chain anti-pattern inside a typed graph shell.

Validation placement changes both safety and cost

Where you validate matters. Validation before tool execution can prevent wasted calls and block dangerous actions early. Validation after tool execution may be necessary when the tool returns data that itself must be checked. In many workflows you need both.

A common sequence is:

- validate user input enough to form safe tool arguments;
- execute read-only tools to gather evidence;
- validate and normalize tool results;
- generate or refine an action proposal;
- run policy verification;
- request human approval if required;
- execute the side-effecting tool.

This order is not universal, but it illustrates a useful principle: place cheap, high-signal validation as early as possible, and place decision-gating validation immediately before the action it authorizes. If you move all validation to the end, bad state travels too far through the graph. If you validate too early without enough evidence, you risk rejecting requests that would become admissible after enrichment.

The trade-off is operational. Early validation saves cost and reduces blast radius. Later validation can be more accurate because more information is available. The right topology depends on the cost of downstream actions and the penalty for false acceptance versus false rejection.

Treat tool results as data products, not raw text

Tool outputs often arrive in the graph carrying implementation noise: provider-specific field names, nullable branches, nested status objects, opaque codes, and inconsistent formatting. If you pass that directly back to the model, you force the model to understand integration details it should not own. You also make prompt behavior sensitive to unrelated API changes.

Normalize tool results before wider consumption. A validator or normalization node can map provider-specific payloads into stable internal types. That gives you three benefits at once:

- the model sees cleaner evidence;
- downstream routing uses stable fields instead of brittle parsing;
- provider swaps or version changes stay localized.

This design is especially important when several tools feed the same later decision. If each tool emits a different shape and the model must reconcile them ad hoc, the workflow becomes harder to test and less predictable. A normalized state model gives the graph a common language.

Observability should attach to control boundaries

The graph already gives you natural observation points: model request, tool execution, validation outcome, approval decision, and side-effecting commit. Use them. You do not need to log every internal to-

ken or every raw payload to get useful traces. You need the boundaries that explain progression.

For each important node, capture enough information to answer four questions:

- what input contract was received;
- what output contract was produced;
- what state changed;
- which edge fired next and why.

That is the minimum set that makes multi-step execution debuggable. It also helps governance. If a regulator or incident reviewer asks why a workflow executed a privileged tool, you want an explicit path through proposal, validation, approval, and execution, not a transcript reconstructed from model chatter.

A practical control pattern for high-stakes tool use

For workflows with meaningful side effects, a robust default pattern is:

- a model node interprets the request and produces a structured proposal;
- a validation node checks proposal completeness and policy-neutral correctness;
- one or more read-only tools gather evidence;
- a normalization node converts tool outputs into stable internal state;
- a verification node or subgraph checks policy and consistency;

- an optional human gate approves or rejects;
- a side-effecting tool executes the action;
- a final validation or reconciliation node confirms the outcome.

You can compress this for simpler paths, but it is a strong default because each boundary carries a distinct form of risk. Interpretation risk sits with the model. Data-shape risk sits with validators. Integration risk sits with tools. Accountability risk sits with approval. The graph makes those risks explicit instead of letting them blur together.

The workflow should remain in charge

A well-orchestrated agent does not surrender control to the model, to the tools, or to the approval UI. The graph remains in charge of what may happen next. Model nodes contribute interpretation. Tool nodes contribute external capability. Validators contribute correctness constraints. Human gates contribute discretionary authority where automation should stop.

That division is what makes the system inspectable and safe under change. You can replace a model, refine a validator, restrict a tool surface, or move an approval threshold without dissolving the workflow into hidden behavior. Every meaningful control boundary stays named, typed, and observable. Once you build with that discipline, multi-step execution stops feeling like an intelligent black box and starts behaving like software you can reason about.

5.5. Choosing Between Explicit Workflows and Autonomous Planning

By the time you have modeled typed nodes, explicit state, guarded tool execution, validation, retries, compensation, and approval boundaries,

a natural question appears: when should you stop encoding control flow directly and let an autonomous planner take over? The question is not academic. Teams often reach it after building a few successful workflow graphs and then meeting tasks that feel less like a regulated process and more like open-ended problem solving. If you answer too aggressively in favor of planning, you can reintroduce the very unpredictability the graph was meant to remove. If you answer too conservatively in favor of explicit graphs, you can freeze a dynamic problem into brittle orchestration that is expensive to maintain.

Koog makes the distinction explicit. Graph-based agents emphasize control, state management, and visualization. Planner agents target iterative planning and execution toward a target state. Those are different execution philosophies. A graph assumes that the legal movement through the problem space can be designed ahead of time with enough fidelity to matter. A planner assumes that the route is part of the problem and must be discovered during runtime.

A simple contrast helps frame the decision.

```
import ai.koog.agents.core.agent.strategy.edge.edge
import ai.koog.agents.core.agent.strategy.node.node
import ai.koog.agents.core.agent.strategy.node.nodeFinish
import ai.koog.agents.core.agent.strategy.node.nodeStart
import ai.koog.agents.core.agent.strategy.strategy

data class AccessRequest(
 val employeeId: String,
 val systemName: String,
 val riskLevel: String
)

data class ApprovalResult(
 val approved: Boolean,
 val requiresHumanReview: Boolean
)

val validate =
 node<AccessRequest, AccessRequest> { input ->
 require(input.employeeId.isNotBlank())
 require(input.systemName.isNotBlank())
```

```
 input
 }

val evaluatePolicy =
 node<AccessRequest, ApprovalResult> { input ->
 if (input.riskLevel == "high") {
 ApprovalResult(
 approved = false,
 requiresHumanReview = true
 )
 } else {
 ApprovalResult(
 approved = true,
 requiresHumanReview = false
 )
 }
 }

val autoApprove =
 node<ApprovalResult, String> { "access-approved" }

val sendToReview =
 node<ApprovalResult, String> { "manual-review-required" }

val workflow = strategy<AccessRequest, String>("access-control") {
 edge(nodeStart forwardTo validate)
 edge(validate forwardTo evaluatePolicy)
 edge(
 evaluatePolicy forwardTo autoApprove
 onCondition {!it.requiresHumanReview && it.approved }
 )
 edge(
 evaluatePolicy forwardTo sendToReview
 onCondition { it.requiresHumanReview }
 )
 edge(autoApprove forwardTo nodeFinish)
 edge(sendToReview forwardTo nodeFinish)
}
```

This workflow is an obvious graph candidate. The path structure is known, the policy boundary is explicit, and auditability matters. You would gain little by replacing it with a planner that can freely invent steps. The runtime problem is not “find any route to a goal.” It is “move through a controlled, reviewable sequence under business constraints.”

Now contrast that with a task such as investigating an unfamiliar production issue across logs, metrics, deployment history, service ownership data, and change tickets. The useful path may depend on what evidence is discovered along the way. The decomposition is exploratory. A planner can add value because route discovery is part of the work.

The hard part is that most real systems sit between those poles.

The first decision is about the problem shape, not the framework feature

Do not start by asking whether you want to use a graph API or a planner API. Start by asking what kind of uncertainty the problem contains.

A workflow graph is strongest when uncertainty exists *inside* steps but not in the legal sequencing of those steps. You may still use an LLM to classify, summarize, extract, or draft inside nodes, but the outer control structure remains stable. The workflow knows which approvals are required, which tools are permitted, what validation gates must pass, and what happens when a failure occurs.

Autonomous planning becomes attractive when uncertainty exists in the *sequence itself*. The system may need to discover which subgoals matter, which tools are relevant, which evidence is sufficient, and what order of action is most promising. In that setting, forcing an explicit graph for every branch may create artificial rigidity and high maintenance cost.

This distinction sounds clean in prose and messy in practice. That is normal. The useful move is to ask where you are willing to tolerate model discretion. If the answer is “inside a bounded step, but not over the business boundary,” use an explicit graph around that step. If the answer is “the route itself must adapt in ways we cannot reasonably pre-encode,” a planner may be justified.

Prefer explicit graphs when the business boundary is fixed

The strongest signal in favor of an explicit graph is that the workflow crosses a business boundary with lasting consequences. Examples include approvals, entitlement changes, refunds, ticket escalation, compliance review, contract generation, customer communications with policy impact, and any tool call that mutates an external system.

These tasks usually share four properties:

- the permissible actions can be enumerated;
- the order of critical checks matters;
- failures must be explained after the fact;
- wrong actions have a cost larger than mere inconvenience.

In these cases, planner freedom is often a liability. A planner can still produce good results on many runs, but the engineering question is not whether it can usually succeed. The question is whether you can constrain, review, test, and defend its behavior when it does not. If a step, approval, tool boundary, or compensation rule must be predetermined and auditable, the outer workflow should remain explicit.

This is especially true when multiple teams share ownership. An operations team, compliance group, and application team can all reason about a graph with named nodes, visible edges, and typed state. They have a much harder time reasoning about a planner whose runtime decomposition changes across similar cases.

Planning earns its keep when decomposition is part of the task

Planner-style autonomy is more appropriate when the system must solve problems whose structure cannot be economically specified in advance. The value comes not from autonomy as a marketing idea, but from runtime search in a variable environment.

Examples include:

- exploratory research over heterogeneous sources;
- troubleshooting where the root cause is unknown and hypotheses must be generated and tested;
- adaptive information gathering across partially known systems;
- long-form synthesis tasks where the best sequence of retrieval, comparison, and verification depends on what is discovered.

In these settings, forcing a fully explicit graph can become busywork. You end up encoding many speculative branches, only to watch the real task vary in ways the graph did not anticipate. A planner can be more productive because it treats decomposition as a runtime decision.

That does not mean planners are inherently better at these problems. It means the engineering trade-off shifts. You accept more execution variance because the cost of pre-encoding every route is too high relative to the risk of guided autonomy.

The key variable is acceptable surface area of model discretion

A useful design question is: *over which decisions may the model exercise discretion without creating unacceptable risk?* This reframes the graph-versus-planner choice as an allocation of authority.

There are several layers where discretion might appear:

- **Interpretive discretion:** classify ambiguous input, summarize evidence, draft text.
- **Tool-selection discretion:** choose which read-only tools to call among an allowed set.

- **Sequencing discretion:** decide the order of intermediate steps toward a goal.
- **Policy discretion:** decide whether a regulated or consequential action is permitted.
- **Commitment discretion:** decide when to execute an irreversible or side-effecting operation.

Most business systems can tolerate the first and sometimes the second. Fewer can tolerate the third without strong boundaries. Very few should tolerate the fourth or fifth without explicit constraints.

This is why a practical heuristic works well: prefer explicit graphs at the business boundary and reserve autonomous planning for contained subproblems with low blast radius. That gives you a stable outer control surface while still using autonomy where it is genuinely useful.

Auditability is often the decisive constraint

If you need to answer "why did the system take this path?" to a reviewer, customer, regulator, or incident team, explicit workflows have a strong advantage. The graph carries the rationale structure in its topology: which node ran, which state changed, which condition fired, which approval was required, and which terminal path was reached.

A planner can still log its reasoning and actions, but the explanation burden is heavier because the route is partly emergent. Similar inputs may produce different intermediate decompositions. That variance is not necessarily wrong, but it makes post hoc reasoning more complex. The more accountability you need, the more expensive that complexity becomes.

This is also where debugging cost matters. A graph failure often reduces to one of a few classes: invalid state, wrong predicate, node bug, tool failure, or missing compensation path. A planner failure may in-

volve poor decomposition, useless tool choice, excessive looping, misplaced confidence, brittle recovery from broken tools, or failure to recognize an unsolvable task. Those are harder to localize because the system is exploring its own route.

If your service model depends on predictable support and incident handling, auditability and debugging cost should count heavily in the decision.

Long horizons and broken environments punish planners first

Autonomous planning looks most impressive on short tasks where the solution space is open but forgiving. It looks less attractive when tasks become long horizon, tool reliability drops, irrelevant tools are available, or the goal is actually unsatisfiable. These are common production conditions. Real systems expose clutter, stale integrations, ambiguous failures, and partial observability.

That matters because planners are sensitive to search quality. A few early bad choices can expand the wrong branch of the state space. Broken tools distort the planner's internal picture of what is possible. Extraneous tools increase distraction. Unsolvable tasks can trigger wasteful loops unless the planner is explicitly designed to recognize stopping conditions.

An explicit graph suffers from some of the same environmental issues, but in a different way. The graph may be less adaptive, yet its failure surface is easier to reason about. When a tool is broken, the workflow usually fails along a named path. When the task is unsatisfiable, the workflow reaches a defined terminal or review branch. You lose exploration power, but you keep tractability.

This is why planners often perform best when bounded by a graph or by a well-curated tool environment. Autonomy in a noisy, high-blast-

radius space can degrade quickly from flexibility into unpredictability.

Change rate cuts both ways

A common argument for planning is that explicit workflows are expensive to maintain when requirements change often. There is truth in that. If the domain changes weekly, hard-coded path structures can become tedious to evolve. A planner may absorb some change more naturally because it can adapt its route instead of waiting for the graph to be revised.

The counterpoint is that change in business-critical systems is rarely unconstrained. New rules arrive, but they still need to be expressed, tested, and governed. If the path changes frequently *and* those changes are policy-significant, hiding them inside planner behavior is usually a bad bargain. You have traded graph maintenance for behavior drift that is harder to verify.

A better interpretation of change rate is this: use planners when the problem structure changes faster than the cost of encoding it is justified, *and* the consequences of route variance are acceptable. Use graphs when the problem changes, but each change still requires deliberate control, approval, or audit.

This is another way to say that not all volatility argues for autonomy. Some volatility argues for better workflow modularity instead.

Implementation effort is not the same as operational cost

Teams sometimes compare a planner and a graph only by initial development effort. The planner appears cheaper because you can describe the goal and supply tools instead of modeling detailed flow. That is a partial calculation.

Operational cost often moves in the opposite direction. An explicit graph typically requires more up-front design, but its runtime behavior

is cheaper to inspect, constrain, and support. A planner may accelerate the first version while increasing the long-term cost of debugging, governance, and trust calibration.

This trade-off is not universal. In a research assistant, open-ended analyst helper, or low-risk exploratory tool, the planner's reduced upfront modeling burden may be the correct choice. In a customer-facing or policy-sensitive workflow, the savings can evaporate quickly once incidents and edge cases arrive.

When comparing effort, separate at least three costs:

- **Modeling cost:** how much structure must you encode before the system can run?
- **Control cost:** how much effort is needed to constrain undesirable behavior?
- **Support cost:** how much work is required to debug, explain, and evolve production runs?

Graphs usually have higher modeling cost and lower control and support cost. Planners often invert that balance.

Hybrid designs are often best, but only with a hard seam

The most practical architecture in many systems is not pure graph or pure planner. It is a graph that contains bounded planner regions. This works well when the business boundary is stable but some internal subproblem benefits from adaptive search.

For example, a graph may control customer request intake, identity checks, approval policy, and side-effecting actions, while delegating one node to a planner that gathers relevant evidence from a bounded set of read-only sources. The planner is allowed to decide *how* to collect

and synthesize the information. The graph remains in charge of *what* can happen with that information afterward.

This hybrid style works only if the seam is hard and explicit. The planner should have:

- a narrow goal;
- bounded tools;
- limited authority over side effects;
- typed input and output contracts;
- clear failure and timeout semantics.

Without that seam, "hybrid" can become a euphemism for unclear ownership. The planner starts influencing business routing, or the graph starts carrying ad hoc planner assumptions. The result is neither adaptive nor predictable.

A good test is whether you can replace the planner node with a deterministic implementation without changing the outer workflow contracts. If you cannot, the boundary is probably too soft.

Decision criteria that work in practice

A compact decision framework helps more than abstract principle. The questions below are practical because each one points toward either explicit control or autonomy.

Are the legal paths knowable? If yes, prefer a graph. If the useful path must be discovered dynamically, planning becomes more plausible.

Are side effects consequential? If yes, keep the business boundary explicit. Let planners operate, if at all, before commitment.

Do approvals, audits, or compensation rules exist? If yes, that is strong evidence for explicit workflows.

Is task decomposition itself the hard part? If yes, planning may add meaningful value.

How expensive is debugging route variance? If expensive, prefer graphs or tightly bounded planner islands.

How much model discretion can you tolerate? If only limited discretion is acceptable, use the graph as the outer authority.

Does the environment contain broken or extraneous tools? If yes, autonomy needs tighter containment because planners degrade under noisy tool surfaces.

What happens when the task is unsolvable? If the system must fail predictably, use explicit stopping conditions or an outer graph that can terminate cleanly.

These are not philosophical questions. They are operational signals.

Anti-patterns that push teams in the wrong direction

Several recurring mistakes distort the graph-versus-planner decision.

One is **using a planner to avoid domain modeling**. If the business process is stable but the team does not want to encode it, a planner may look attractive. The hidden cost appears later when the system behaves variably around steps that should have been explicit.

Another is **encoding exploratory work as a rigid graph**. If you find yourself drawing dozens of speculative branches for tasks whose route genuinely depends on discovered evidence, you are paying graph complexity tax without receiving corresponding control value.

A third is **unclear ownership in hybrids**. The graph thinks the planner is just an internal helper, while the planner is effectively decid-

ing customer-visible or compliance-relevant outcomes. That ambiguity is dangerous because nobody knows which layer owns correctness.

A fourth is **giving planners high-blast-radius tools too early**. If the planner can mutate critical systems while still exploring the problem, you have granted discretion at the wrong layer.

A fifth is **treating all autonomy as innovation and all explicit control as bureaucracy**. That framing obscures the real engineering question, which is how much variance your system can safely absorb.

A bounded-planner pattern is often the highest-leverage compromise

When you do need autonomy, constrain it aggressively. A bounded planner node inside an explicit graph often gives you the best trade-off. The graph handles intake, validation, policy, approval, and side-effecting execution. The planner handles one contained search or synthesis problem with a clear goal and limited tool set.

That pattern works because it respects the asymmetry between exploration and commitment. Exploration benefits from flexibility. Commitment benefits from control. By separating those phases, you avoid forcing one execution style to solve both problems.

For example, you might allow a planner to investigate which internal documents, support tickets, and logs are most relevant to a customer dispute. Once the planner returns a typed summary and evidence bundle, the outer graph validates it, routes for review if needed, and controls any final account action. The planner is useful, but it never owns the business boundary.

Choose for the failure mode, not the demo

Both explicit graphs and planners can look impressive when they suc-

ceed. The more revealing question is how each style fails under your real operating conditions. If the wrong action is expensive, if route explanations matter, if side effects need compensation, or if human gates must remain authoritative, explicit workflows are usually the right answer. If the task is exploratory, the environment is only partly known, and route discovery is where the value lies, bounded autonomy becomes worthwhile.

That is why the decision should be made from the failure mode outward. Graphs fail visibly and often conservatively. Planners fail more adaptively, but sometimes more opaquely. One is not universally superior. They optimize for different risk profiles. Once you choose based on the acceptable surface area of uncertainty rather than on implementation fashion, the architecture becomes easier to justify and much harder to regret.

Chapter 6

Planning, Memory, and Long-Running Context

Experienced teams usually discover that the hard part of agentic systems is not generating one good answer, but sustaining coherent behavior across hours, sessions, and partial failures. This chapter gives the reader the design vocabulary and implementation criteria for planner-driven execution, layered memory, retrieval grounding, token-efficient continuity, and checkpointed recovery in Koog, while making explicit which concerns belong here and which are treated elsewhere in Chapters 4 and 6.

6.1. Choosing Planner-Driven Execution with GOAP

A fixed workflow graph works well when you can name the important states ahead of time, bound the branches, and attach explicit policies

to each edge. That breaks down when the agent must discover the path while it is already executing. A research assistant may need to search, compare, refine the question, fetch one more source, and then restart the synthesis step because a tool result changed the scope. A remediation agent may inspect logs, try one recovery action, observe that the symptom persists, and choose a different branch that nobody could justify hard-coding as the single default. Support automation has the same shape: the right next step depends on account state, policy exceptions, tool availability, and whether a previous attempt partially succeeded.

Those are not cases where determinism is unimportant. They are cases where fully enumerating the intermediate control flow becomes more expensive, more fragile, and less auditable than exposing a constrained decision surface and letting the runtime choose from it. GOAP, or goal-oriented action planning, gives you that surface. Instead of writing every branch directly, you define a goal, a representation of world state, a bounded set of actions, and the conditions under which each action is valid. The planner searches for a sequence of actions whose effects can satisfy the goal with acceptable cost.

A planner is not a substitute for workflow design discipline. It is a different control primitive. Use it when the branching entropy is high enough that explicit graphs become combinatorial, but not so high that you lose the ability to inspect and govern the search space. If you cannot describe the world state clearly, cannot bound action effects, or cannot tolerate runtime variation in intermediate steps, a typed workflow graph remains the better choice.

A concrete Koog planning setup

Koog exposes planner primitives for GOAP-style execution, including types named `GOAPPlanner`, `GOAPPlan`, `Action`, and `Goal`. The exact package placement has moved across releases, so the stable architec-

tural point is more important than memorizing imports: you define typed actions with explicit preconditions, costs, effects on belief or world state, and an execution body; then you attach the planner to an agent through the planner strategy API.

The following example is intentionally compact. It models an incident assistant that wants to restore service health. The agent can inspect metrics, restart a service, or escalate to a human. The planner chooses among those actions based on the current state rather than following a fixed graph.

```
data class IncidentState(
 val metricsInspected: Boolean = false,
 val serviceHealthy: Boolean = false,
 val restartAttempted: Boolean = false,
 val escalationOpened: Boolean = false
)

val restoreHealthGoal = Goal<IncidentState>(
 name = "restore-service-health"
) { state ->
 state.serviceHealthy || state.escalationOpened
}

val inspectMetrics = Action<IncidentState>(
 name = "inspect-metrics",
 cost = 1.0,
 precondition = { state ->
 !state.metricsInspected
 },
 effect = { state ->
 state.copy(metricsInspected = true)
 },
 execute = { state, _ ->
 state.copy(metricsInspected = true)
 }
)

val restartService = Action<IncidentState>(
 name = "restart-service",
 cost = 3.0,
 precondition = { state ->
 state.metricsInspected &&
 !state.serviceHealthy &&
 !state.restartAttempted
```

```
 },
 effect = { state ->
 state.copy(
 restartAttempted = true,
 serviceHealthy = true
 )
 },
 execute = { state, _ ->
 state.copy(
 restartAttempted = true,
 serviceHealthy = true
 )
 }
)

val escalate = Action<IncidentState>(
 name = "open-escalation",
 cost = 10.0,
 precondition = { state ->
 state.metricsInspected && !state.serviceHealthy
 },
 effect = { state ->
 state.copy(escalationOpened = true)
 },
 execute = { state, _ ->
 state.copy(escalationOpened = true)
 }
)
```

In production you would not hard-code the execution body to set `serviceHealthy = true`. You would call real tools, re-read the environment, and update state from observed outcomes. The example still captures the engineering contract that matters: the planner searches over action metadata, not over unconstrained model output.

When you integrate planning into the agent layer, use Koog's planner strategy mechanism rather than older builder patterns that were removed or reorganized. The current documented direction is to create an `AIAgentPlannerStrategy` and attach a planner instance to the agent configuration. The exact wiring is version-sensitive, so keep code examples conservative when package names are not verified from the active API docs.

What GOAP changes in your control model

A typed workflow graph encodes control flow as explicit edges. The runtime chooses among a small number of known branches, often with deterministic conditions. GOAP replaces those edges with a search problem:

- the goal defines what counts as success;
- the world state records facts relevant to planning;
- each action advertises when it is legal;
- each action also declares how it changes state;
- cost lets you prefer cheaper or safer plans;
- execution can replan when reality diverges from the expected effects.

That last point is the real operational advantage. In a graph, you normally branch after a step finishes and exposes a result. In a planner, the whole sequence is provisional. The system can commit only to the next action, observe what actually happened, mutate the state, and search again. That is controlled autonomy: the system decides intermediate steps, but only from the finite action alphabet and state transitions you provided.

This structure gives you leverage only if the state model is honest. Hidden mutable state destroys planner quality. If an action reads an external flag, an untracked cache, or a tool side effect that never enters the planning state, the planner is optimizing against a false world. That creates the worst possible failure mode: a plan that appears rational in traces but was never grounded in the real execution environment.

When a planner is a better fit than a graph

Planner-driven control starts winning when the number of meaningful intermediate paths is large, unstable, or only knowable after runtime discovery.

Research and investigation agents. You usually know the destination but not the route. The agent may need to collect evidence, detect contradictions, revise search terms, and decide when enough evidence exists to synthesize. A graph that enumerates these branches quickly turns into a brittle tree of special cases.

Operational remediation. Failures often unfold as diagnostic loops. You inspect symptoms, try a low-risk fix, validate, then escalate to a different action family if the result is ambiguous or negative. Preconditions and effects are often clearer than a single canonical sequence.

Support resolution with policy overlays. The same user request may require different next steps depending on entitlements, region, prior case history, account status, and recent tool results. The branching factor is high, but the permissible actions remain small and policy-bounded.

Tool ecosystems with partial failure. If several tools can satisfy the same subgoal with different costs, latency, or confidence, a planner can choose among them. A graph can model this, but only by expanding alternatives into explicit branches that become difficult to maintain.

You should stay with typed workflow graphs when any of the following is true:

- You need exact, repeatable step order for compliance or audit.
- Human approval points must occur at fixed locations.
- The branch space is small and stable.
- The cost of a wrong intermediate action is high.

- Latency must be tightly bounded and search overhead is unacceptable.
- You cannot define valid state transitions without hidden context.

A graph is also superior when the work is primarily transactional. If the process is "validate request, write record, send confirmation," planning adds uncertainty without adding useful flexibility.

The key decision signals

You can usually decide between graph, planner, and hybrid control by checking six signals.

Branching entropy. Count how many distinct intermediate decisions you can reasonably foresee. If each node in the process opens several plausible branches and those branches depend on observations you do not have until runtime, planning becomes attractive.

Environmental volatility. If tools, data, and external conditions change frequently, hard-coded graphs decay quickly. Planners absorb some of that change by choosing among available actions at runtime, provided the action library stays well-governed.

Auditability requirements. If an auditor must ask, "Why did the system choose this path?" you need state snapshots, visible preconditions, and decision traces. Planners can still work here, but only if you instrument them at the same level of detail as you would a workflow engine.

Failure containment. If one wrong action can cause irreversible damage, do not expose that action directly to planner search. Put a deterministic gate around it, require approval, or move the dangerous step into a graph-managed enclave.

Cost sensitivity. Planning is a search problem. Even bounded search adds latency and compute. If your workload is high-volume and repet-

itive, a graph often wins on both predictability and cost.

Modeling effort. A planner reduces explicit edge modeling, but it increases action semantics work. You must write credible preconditions, costs, and effects. If your team treats planning as a shortcut around that discipline, the planner will underperform a boring but explicit graph.

A practical hybrid pattern

The most reliable production design is often hybrid rather than pure. Use a graph for the irreversible skeleton and a planner inside bounded subproblems.

For example, a support workflow may keep the outer control points fixed:

1. authenticate the request;
2. classify intent;
3. gather evidence;
4. require approval if a sensitive action is needed;
5. execute the approved action;
6. record the outcome.

Within "gather evidence," the planner can choose which diagnostic actions to run and in what order. Within "execute the approved action," the planner can choose among low-risk variants if those variants are operationally equivalent. The graph provides stable governance boundaries; the planner handles local combinatorics.

This pattern matters because it separates *autonomy scope* from *business accountability*. You do not need to choose between full determin-

ism and free-form agent behavior. You can decide exactly where search is allowed.

Operational flow for planner-driven runs

A production planner loop should be legible enough that another engineer can reconstruct the run from logs and persisted state. A useful sequence is:

1. declare the goal in machine-checkable form;
2. capture an explicit state snapshot;
3. enumerate candidate actions;
4. filter actions by preconditions and policy;
5. score or search for a plan under bounded depth or cost;
6. execute the next action behind guards;
7. observe the real outcome and mutate state;
8. replan if the observed state diverges from the predicted effect;
9. terminate on goal satisfaction, iteration bound, or governed failure.

Koog's planner execution loop is iterative rather than magical. The planner can fail to converge and may stop when it reaches a maximum number of iterations. The documented exception name for this condition is `AIAgentMaxNumberOfIterationsReachedException`. That single fact should shape your operational design. A planner run always needs explicit convergence control, including one or more of:

- a maximum iteration count;

- a maximum cumulative action cost;
- a wall-clock budget;
- a ban on revisiting equivalent states;
- a rule that escalates to a human or deterministic fallback.

Without those limits, a planner that is merely confused becomes a planner that is expensive, slow, and difficult to interrupt cleanly.

Action design: typed surfaces and honest effects

Most planner failures are action-design failures. A good action surface has four properties.

Typed inputs and outputs. Do not let an action accept a vague map of unstructured fields if the action decides policy-sensitive work. Typed state makes preconditions inspectable and testable.

Conservative preconditions. Preconditions should answer "when is this action valid?" not "when do I hope it works?" If an action requires fresh metrics, an authenticated principal, and a writable target, encode those constraints explicitly.

Predictive effects separate from observed results. The effect used for planning is a hypothesis about how the world should change if the action succeeds. The observed post-execution state is the source of truth. Never assume the effect happened just because the planner expected it.

Bounded side effects. If an action can create tickets, delete resources, or change account state, expose it only after you add idempotency keys, compensating logic, or approval gates. Planner search should not be allowed to discover destructive behavior by accident.

A useful discipline is to classify actions into three categories:

- read-only actions, which inspect state and are safest to expose;
- reversible actions, which have a clear compensation path;
- irreversible actions, which require deterministic control or approval.

A planner-heavy system should mostly search over the first two categories.

Debugging a planner is not like debugging a graph

When a graph fails, you usually inspect the node, the branch predicate, and the tool result that triggered the branch. Planner failures have more dimensions. The defect may be in the search itself, but it is often in one of these layers:

- the goal was underspecified;
- the state omitted a critical fact;
- a precondition was too permissive or too strict;
- an action cost encouraged the wrong plan;
- the expected effect did not match reality;
- replanning occurred too late or too often;
- the planner hit its iteration bound before reaching a viable state.

You need traces that make each layer visible. At minimum, persist or emit:

- the initial state snapshot;
- the goal definition or goal identifier;

- candidate actions considered at each step;
- the subset pruned by failed preconditions;
- the selected action and its score or cost contribution;
- the predicted state transition;
- the observed post-action state;
- the reason for replanning or termination.

Do not evaluate planner quality only by asking whether the run eventually succeeded. A planner that succeeds through unnecessary actions, expensive detours, or repeated unsafe near-misses is still poor. Measure action efficiency, number of replans, rate of human escalation, average path cost, and plan stability under similar starting states.

Observability and governance consequences

Allowing the runtime to decide intermediate steps changes your governance problem. A graph gives you a predefined set of paths to review. A planner gives you a predefined action vocabulary but an open set of possible paths through that vocabulary. Governance has to move one level down, from edge review to action and state review.

That means four concrete controls.

Policy-constrained action registration. Only register actions that are permitted in the current environment. Do not rely on prompts to tell the planner what it should avoid if you can prevent those actions from being available at all.

Inspectable state mutation. Every change that influences future planning must be visible in the recorded state. Hidden mutable state is both an observability defect and a governance defect.

Approval boundaries around high-impact actions. If an action changes money, identity, access, or production infrastructure, place that action behind explicit approval or wrap it in a deterministic graph segment.

Replayability for review. You should be able to reconstruct why the planner chose an action sequence from the saved state transitions and action metadata, even if the exact external environment has since changed.

This is also where planner-driven execution intersects with checkpointing and restoration. If you persist a planner run, you must decide whether restoration continues the old plan or restores the last state and replans. The latter is often safer because the world may have drifted since the original plan was computed.

Anti-patterns that look attractive at first

Using a planner to avoid modeling effort. Planning reduces edge enumeration, not semantic rigor. If you cannot define state, costs, and effects, the planner will simply hide weak modeling behind runtime variability.

Exposing unsafe tools directly to search. If a destructive tool sits in the planner action set with a plausible cost and a permissive precondition, the planner will eventually consider it. Governance belongs in the action surface, not only in post hoc review.

Allowing hidden state to steer planning. If a tool's private cache, an unstored prompt variable, or a side-channel flag influences action success, your planner traces become misleading and irreproducible.

Letting the language model define the whole search space. GOAP works because actions are constrained and typed. If you ask the model to invent new intermediate actions freely, you have stepped outside planner-driven control and into unbounded agent improvisation.

Judging success only by final outcome. A planner that reaches the goal after five wasteful steps may still be operationally worse than a graph that reaches it in two deterministic steps.

A short checklist for the design choice

Choose planner-driven execution when you can answer yes to most of these questions:

- Does the task require runtime discovery of intermediate steps?
- Can you model the relevant world state explicitly?
- Can you define a bounded, policy-safe set of actions?
- Are action preconditions and effects inspectable?
- Can you tolerate some variation in path while requiring control over outcomes?
- Do you have observability strong enough to reconstruct planning decisions?

Stay with a graph when these dominate instead:

- fixed order matters more than adaptive path selection;
- review and approval points must be static;
- latency and cost variance must be minimal;
- path explainability must be immediate and simple;
- the environment is stable enough that explicit branching stays manageable.

The planner earns its place when runtime flexibility is a real operational requirement rather than an aesthetic preference. In that setting, GOAP gives you a disciplined way to let the system choose *how* to get there without surrendering control over *what* it is allowed to do.

6.2. Designing Short-Term Memory, Long-Term Memory, and Retrieval Boundaries

Agents fail in a distinctive way when every persisted artifact is treated as "memory." A transcript fragment, a user preference, a task checkpoint, and a retrieved policy passage all end up in one undifferentiated store. The result is predictable: stale facts survive too long, low-value chat residue pollutes prompts, retrieval results get mistaken for durable truth, and the agent starts carrying yesterday's context into today's decision. Long-running systems need continuity, but continuity only helps when each layer of state has a single, explicit job.

For Koog agents, the cleanest boundary starts with three categories. Short-term memory preserves conversational working state across turns. Long-term memory preserves curated facts that should outlive the current interaction. Retrieval reintroduces external knowledge at query time and should not be confused with either of the first two. These are not naming preferences. They determine what you persist, when you evict, which writes require review, and how much you trust the material when the agent plans or answers.

A concrete Koog memory setup Koog provides separate features for short-term conversational continuity and long-term retrievable memory. `ChatMemory` persists and restores conversation history through a `ChatHistoryProvider`. `LongTermMemory` is a separate feature with its own retrieval and ingestion flow. That separation is the

architectural signal you want to preserve in your own design.

The first example shows the actual contract for short-term memory persistence. The `ChatHistoryProvider` interface is deliberately narrow: load messages for a conversation identifier, then store the updated list at the end of a run.

```
import ai.koog.agents.memory.ChatHistoryProvider
import ai.koog.prompt.message.Message
import java.util.concurrent.ConcurrentHashMap

class TicketChatHistoryProvider: ChatHistoryProvider {
 private val store =
 ConcurrentHashMap<String, List<Message>>()

 override suspend fun load(
 conversationId: String
 ): List<Message> {
 return store[conversationId].orEmpty()
 }

 override suspend fun store(
 conversationId: String,
 messages: List<Message>
 ) {
 store[conversationId] = messages
 }
}
```

That provider is suitable for tests and short-lived process memory. In production you would back it with a database or other durable store keyed by a stable conversation identifier. The important design point is that `ChatMemory` deals in *conversation history*, not arbitrary execution state and not a generic facts database.

Koog also supports preprocessing of chat history before it is reintroduced. The memory module includes a sliding-window preprocessor, which is the first built-in control for bounded short-term continuity. The exact constructor wiring can vary with the active API surface, so keep the mental model precise even when you defer the final configuration details: load the transcript, transform it through a preprocessor,

and then feed only the retained window into the run.

For long-term storage, Koog uses a different abstraction boundary. The long-term feature can retrieve records before model calls and ingest new records after completion. Underneath, retrieval is built on `SearchStorage`; ingestion writes through `WriteStorage`. That distinction matters because it prevents you from treating a raw chat transcript as if it were the same thing as a curated knowledge record.

Three layers, three control objectives You can reason about memory design by assigning each layer one control objective.

Short-term memory exists to preserve local continuity. It carries recent turns, active user intent, unresolved references, pending tool outputs, and other working material needed to make the next step coherent. Its horizon is limited. You keep it because the agent still needs it now, not because it deserves permanent retention.

Long-term memory exists to preserve durable, reusable facts with some semantic stability. Typical examples include user preferences, standing task constraints, identity-linked metadata, approved summaries, and recurring operational facts that remain relevant beyond the current session. Long-term memory should be sparse, structured, and intentionally written.

Retrieval exists to reintroduce external knowledge on demand. A retrieved policy document, code snippet, runbook paragraph, or knowledge-base article is not the agent's durable belief. It is evidence fetched for the current need. Retrieval is query-time access to an external corpus, not a statement that the agent "remembers" the material in the same way it remembers a durable user preference.

These control objectives imply different correctness rules. Short-term memory can be incomplete as long as it preserves what the current

interaction needs. Long-term memory must be selective because wrong durable recall is more damaging than temporary forgetfulness. Retrieval can be transient and probabilistic, but it must remain attributable to a source corpus rather than silently becoming agent-owned fact.

Short-term memory as working state Short-term memory is closest to what people informally mean when they say an agent should "remember the conversation." In Koog, that means conversation history loaded at run start and stored at completion. This is intentionally narrower than general application state.

That narrowness is useful. It prevents a common architectural mistake: pushing every bit of execution context into the transcript because the model can consume messages. Tool receipts, planner internals, authorization artifacts, and retry counters may all be important, but they are not automatically part of conversational continuity. If you write them into chat history, you inflate token costs and blur the line between language context and control state.

A well-designed short-term layer usually contains:

- recent user and assistant turns that still affect the current task;
- tool outputs that the model must reference directly;
- unresolved slots, commitments, or questions;
- active goals that have not yet been discharged;
- limited narrative context needed for coherent follow-up.

It usually should not contain:

- raw retrieval results that can be fetched again;

- durable user profile facts better stored separately;
- low-level workflow state or persistence checkpoints;
- verbose tool logs or traces that the model will not use;
- historical turns retained only because they once existed.

The sliding-window preprocessor reflects this design. If the goal is immediate coherence under a bounded token budget, recent turns often dominate older ones. A windowing policy is not enough by itself for every workload, but it enforces the right instinct: short-term memory is an operational buffer, not an archive.

Choosing the boundary for short-term persistence Your first decision is the identity of the conversation. If you key short-term memory too broadly, unrelated tasks leak into each other. If you key it too narrowly, the agent loses continuity every time the user changes channel, device, or subtask.

A useful pattern is to scope conversation history to the *interaction thread* rather than the person globally. A support ticket, a chat session, or a task instance usually makes a better `conversationId` than a user identifier. You can still link the thread to the user elsewhere. Keeping those identities separate reduces cross-task contamination and makes eviction policy much easier to reason about.

The second decision is how much chat history to retain before preprocessing. Retaining every turn indefinitely is the easiest implementation and usually the worst policy. You pay for stale residue in three ways:

1. token budget grows even when relevance does not;
2. old instructions and obsolete facts compete with current intent;

3. the agent becomes harder to debug because any response may be conditioned on forgotten earlier turns.

A bounded short-term layer gives you a clean default. If older content still matters, you either compress it into a summary, promote the durable part into long-term memory, or recover it through retrieval from an external source. Those are different operations for a reason.

Long-term memory as curated durable state Long-term memory should be difficult to write accidentally. Its job is to preserve facts whose value survives the current exchange and whose correctness matters across future runs. Because it is durable, errors compound. A mistaken preference stored once can bias many later interactions. A stale domain fact can persist long after the world has changed.

That is why long-term memory needs stricter admission rules than chat history. A fact belongs in long-term memory when it is:

- likely to matter in future interactions;
- stable enough that reuse is reasonable;
- attributable to a reliable source or explicit user statement;
- representable in a structured, reviewable form;
- distinct from ephemeral dialogue residue.

Examples that often qualify:

- “The user prefers CSV exports over PDF reports.”
- “This account must route approvals through team X.”
- “The task may not perform destructive actions without approval.”

- "The service uses region-specific maintenance windows."

Examples that usually do not qualify:

- "The user sounded frustrated in the last message."
- "The model suspects the root cause is cache drift."
- "A retrieved article mentioned an older operating limit."
- "The current conversation included a long troubleshooting log."

Koog's `LongTermMemory` feature reinforces this separation. Retrieval and ingestion are configured as independent flows. That means you can decide what gets searched without automatically deciding what gets written back. This is an important design freedom. Many teams wire retrieval first and then casually dump conversation artifacts into the same store. The result is a hybrid corpus of user-specific facts, temporary summaries, and external documents, none of which has a clear trust model.

Write criteria for long-term memory A durable memory system needs explicit write criteria. Without them, you are not building memory; you are building contamination.

A strong write policy usually asks four questions before a record is accepted:

Is the fact durable? If the answer may change tomorrow or after the next tool call, keep it out of long-term memory.

Is the fact specific? Durable memory should be structured enough that you can inspect, update, or delete it. Free-form narrative blobs are difficult to govern.

Is the provenance acceptable? A direct user statement, an approved workflow result, or a verified system record is stronger than an inferred model guess.

Is the retention justified? If the system cannot explain why this fact should persist across sessions, do not store it durably.

That policy leads naturally to schema discipline. Even if the underlying store can hold unstructured text, define fields such as entity identifier, fact type, value, source, confidence, created-at time, and expiration or review horizon. Free text can still exist, but it should not be the only representation.

Retrieval is not memory Retrieval gets mislabeled as memory because it makes the agent appear informed. The mechanism is different. The agent is not recalling a fact from durable internal state; it is fetching relevant material from an external knowledge substrate when needed.

That distinction matters for three reasons.

First, retrieved knowledge remains external and revisable. If a policy changes in the source corpus, the next retrieval should surface the update. Long-term memory, by contrast, can go stale unless you explicitly rewrite or expire it.

Second, retrieval is query-dependent. What appears in context depends on the search request, indexing quality, metadata filters, and ranking policy. That means absence from retrieval is not proof that no relevant fact exists.

Third, retrieval has a different trust model. A retrieved record may be authoritative, ambiguous, outdated, tenant-inappropriate, or only partially relevant. The agent should treat it as evidence to ground the current step, not as automatically absorbed durable belief.

Koog's storage abstractions make this separation explicit. `SearchStorage` supports retrieval of relevant records. `WriteStorage` supports ingestion. These are retrieval-system contracts, not chat-history contracts. If you use them to store everything the agent has ever seen, you erase the distinction between external knowledge access and durable memory formation.

A useful boundary test When you are unsure where a datum belongs, apply three tests.

Replay test. If you start a new session tomorrow, should the datum still influence behavior even if the current conversation never mentions it? If yes, it may belong in long-term memory.

Recompute test. Can you fetch or derive the datum cheaply from an external source when needed? If yes, retrieval may be sufficient and durable storage may be unnecessary.

Context test. Does the datum matter only because of the immediate exchange or active task state? If yes, keep it in short-term memory or task state, not long-term memory.

For example, a user's standing export preference passes the replay test and belongs in durable memory. A paragraph from the vendor runbook usually fails the replay test and passes the recompute test, so it belongs in retrieval. A tool result showing the most recent failed deployment may matter right now but not next week; that is short-term context unless you explicitly elevate a durable lesson from it.

Trust levels and revalidation The layers also differ in how much revalidation you need before reuse.

Short-term memory often carries the lowest validation burden because it is fresh by construction. If the user stated a constraint two turns ago,

replaying it is usually reasonable. The main risk is not staleness but overload or ambiguity.

Long-term memory needs explicit freshness policy. Some records are nearly static, such as preferred output format. Others decay, such as organizational roles, routing rules, or operating thresholds. If a durable fact can drift, attach a review horizon or require validation before operational use.

Retrieval always needs relevance and source checks. Even perfect search can return material that is outdated, superseded, or applicable to a different tenant or environment. That is why metadata and corpus governance matter as much as similarity quality. A retrieval record should arrive with enough context for the agent or surrounding system to decide whether it is safe to use.

If you collapse these trust levels into one undifferentiated "memory score," the agent loses the ability to distinguish between what was just said, what has been approved for long-term reuse, and what was merely fetched for situational grounding.

Interaction with planner-driven agents Planner-driven execution raises the stakes of memory boundaries. A planner chooses actions based on state. If the state is polluted with stale transcript residue or unverified retrieved facts, the planner can make locally rational but globally bad decisions.

Use short-term memory to carry active commitments and immediate observations that affect the next planning step. Use long-term memory for durable constraints that should shape planning repeatedly, such as approval requirements or persistent user preferences. Use retrieval to bring in runbooks, policies, and external facts only when the current plan needs them.

Do not let the planner silently treat retrieved passages as if they were durable state transitions. A runbook excerpt saying that a restart *usually* fixes a symptom is evidence, not proof that the service is restart-safe in the current environment. Likewise, do not let every intermediate planner hypothesis become long-term memory. Candidate explanations are working beliefs, not durable facts.

Anti-patterns that damage memory quality **Persisting every conversation turn forever.** This creates transcript bloat, token waste, and context drift. Retain what the active interaction needs, then compress, evict, or promote selectively.

Storing retrieved documents as durable user memory. Retrieval results are not identity-linked facts. If you need durable knowledge records, ingest them intentionally with schema and provenance.

Letting the model self-write long-term memory without policy. Durable writes need criteria, provenance, and usually review rules. An unconstrained model will over-store speculation and incidental detail.

Mixing execution state with durable facts. Retry counters, pending tool calls, and workflow flags belong to execution control or checkpointing, not to user or task memory.

Using one store for everything because it is convenient. Convenience at the storage layer produces ambiguity at the reasoning layer. Separate interfaces are worth the extra plumbing.

A compact operational policy A practical policy for most systems is simple:

1. Load short-term chat history for the active conversation.

2. Apply a bounded preprocessor before prompting.
3. Retrieve external knowledge only for the current need.
4. Write to long-term memory only when a durable fact passes explicit admission rules.
5. Keep execution checkpoints and workflow state out of both chat history and long-term memory stores.

That policy is not conservative for its own sake. It keeps each store legible. When the agent behaves incorrectly, you can ask the right question: did it inherit bad local context, recall a bad durable fact, or retrieve the wrong external evidence? If all three are merged, diagnosis becomes guesswork.

Data ownership, deletion, and lifecycle Boundary design also determines who owns deletion and retention policy. Chat history often follows session or task lifecycle. Long-term memory usually follows account or domain lifecycle and needs targeted update and delete operations. Retrieval corpora follow document governance: source refresh, versioning, access scope, and tenancy controls.

You should not have to delete a user's historical preference by rewriting a whole transcript archive. You should not have to purge obsolete policy text from a long-term memory table when the real problem is stale retrieval indexing. A separate lifecycle per layer is operationally cleaner and legally safer.

Expiration policy belongs here as well. Short-term memory expires aggressively unless preserved through summary or promotion. Long-term memory expires selectively based on fact type. Retrieval records inherit expiration and supersession rules from their source corpus. Once you assign each layer a lifecycle, stale context stops accumulating silently.

What to optimize for The correct target is not maximum remembrance. It is reliable reuse of the right information at the right time. Short-term memory optimizes continuity. Long-term memory optimizes durable personalization and task carryover. Retrieval optimizes grounded access to external knowledge. Those objectives are compatible only when you keep the boundaries sharp enough that each layer can be governed, tested, and debugged on its own terms.

6.3. Building Embeddings and RAG Pipelines for Grounded Agent Behavior

A vector store attached to an agent does not automatically produce grounded behavior. You can retrieve text that looks relevant, place it into the prompt, and still get a poor decision because the wrong chunk was indexed, the right chunk was buried among ten weak matches, metadata filters were missing, or the retrieved material was never aligned with the action the agent needed to take. Retrieval becomes valuable only when you treat it as an engineered pipeline whose output is judged by downstream behavior: better diagnoses, safer tool choices, fewer hallucinated facts, and more stable plans.

That design pressure is stronger in long-running agents than in one-shot question answering. A planner may need a runbook step, a support agent may need the current entitlement policy, and a remediation agent may need the latest maintenance window or dependency map. If retrieval is noisy, stale, or tenant-unsafe, the agent does not merely answer poorly; it can choose the wrong next action. That is why the retrieval subsystem belongs beside memory and control design rather than as an afterthought.

A minimal retrieval substrate in Koog

Koog separates retrievable storage from chat history and from durable

execution state. At the retrieval layer, the documented `rag-base` abstractions include `LookupStorage`, `WriteStorage`, `DeletionStorage`, `SearchStorage`, `SearchRequest`, `SimilaritySearchRequest`, and `DocumentWithPayload`. Those names matter because they define the stable boundary you should target even when the embedding provider or vector backend changes.

The following example stays close to the verified API surface and avoids inventing provider-specific helpers. It shows the shape of the storage interface you design against: write curated records into a search-capable store, then issue a similarity-oriented request at query time.

```
import ai.koog.rag.base.DocumentWithPayload
import ai.koog.rag.base.SearchStorage
import ai.koog.rag.base.WriteStorage
import ai.koog.rag.base.SimilaritySearchRequest

suspend fun ingestPolicyRecord(
 writer: WriteStorage,
 id: String,
 text: String,
 tenantId: String,
 source: String
) {
 val doc = DocumentWithPayload(
 id = id,
 text = text,
 payload = mapOf(
 "tenantId" to tenantId,
 "source" to source,
 "kind" to "policy"
 )
 )
 writer.write(doc)
}

suspend fun searchPolicies(
 storage: SearchStorage,
 query: String
) = storage.search(
 SimilaritySearchRequest(
 query = query
 )
)
```

The exact retrieval result types and backend wiring depend on the concrete implementation, but the engineering contract is already clear. Your application owns document preparation, metadata quality, and the decision of what to write. The search layer owns ranked lookup over that corpus. That is enough to reason about pipeline design without pretending that a vector database is the system.

Koog's `LongTermMemory` feature builds on these retrieval abstractions. Retrieval before a model call and ingestion after completion are independent flows. The default query source is the last user message, which is a reasonable baseline but rarely the last word for production systems. Many real tasks need query rewriting, metadata-constrained search, or multiple retrieval passes targeted at different corpora.

What embeddings are actually buying you

An embedding is not a knowledge store. It is a learned projection that maps text, code, or other content into a vector space where semantically related items are closer together than unrelated ones. That makes approximate nearest-neighbor search practical at scale. The retrieval system then uses that geometry to propose candidates that are worth ranking, filtering, and grounding into the prompt.

This distinction matters because teams often over-attribute intelligence to embeddings themselves. Embeddings do not solve provenance, freshness, tenancy, or policy scope. They do not know whether a retrieved paragraph is the canonical operating procedure or an obsolete note from a decommissioned environment. They solve one subproblem: semantic candidate generation under a particular embedding model's notion of similarity.

That is still a powerful subproblem to solve. Many operational corpora are not keyword-friendly. Users ask for "why is billing export blocked for this account" while the documentation stores the relevant rule under "entitlement gating for scheduled report delivery." Embeddings

can bridge that lexical gap. They can also help with semantically similar ticket histories, code comments, troubleshooting notes, and policy fragments whose vocabulary differs from the user's phrasing.

The catch is domain fit. Code, legal policy, multilingual support content, and incident logs do not all behave the same under one embedding model. If your corpus contains mixed modalities, evaluate retrieval against the domain that actually matters for the agent's decisions rather than against a generic semantic benchmark.

Corpus design determines retrieval quality

Most retrieval failures originate before any search call is issued. If the corpus is low quality, no ranking strategy will rescue it consistently. Four design choices dominate.

Chunk boundaries. A chunk should preserve the semantic unit that the agent needs to act on. If you split a procedure from its prerequisites, the model may retrieve the action without the warning. If you keep whole documents intact, you increase noise and token cost. For operational prose, medium-grained chunks with local overlap often work better than either sentence-level fragmentation or full-document stuffing. Structured sources such as playbooks, tickets, and policy tables may need schema-aware chunking rather than plain text slicing.

Metadata richness. Retrieval without metadata is often unsafe. Tenant, environment, document type, source system, freshness timestamp, access scope, and supersession markers should travel with each indexed unit. Missing metadata is how you retrieve the right instruction for the wrong customer.

Freshness and versioning. A support policy that changed last week is worse than useless if the store still returns the old text as a top hit. Treat reindexing, replacement, and deletion as first-class maintenance operations rather than occasional batch jobs.

Deduplication and canonicality. If the same content appears in six lightly edited forms, vector retrieval may surface redundant variants and crowd out other evidence. Mark a canonical source when one exists, and either deduplicate duplicates or down-rank them.

These corpus decisions are more influential than many teams expect. Retrieval systems fail quietly: they return something, so the pipeline appears healthy. The visible failure happens later when the agent chooses an action grounded on a weak or stale chunk.

Pipeline stages and their failure modes

A useful retrieval pipeline has at least seven stages, each with its own failure class.

Ingestion. You normalize source records, split them into retrieval units, attach metadata, and write them to storage. A common failure is storing raw material that was never intended for agent consumption, such as partial logs, duplicate exports, or tenant-mixed operational dumps.

Embedding generation. You compute vectors for the queryable units. Failure shows up when the embedding model underrepresents your domain semantics or when ingestion and query embedding are inconsistent across environments.

Index maintenance. You update or rebuild the search index as sources change. Failure here creates staleness, orphaned records, or inconsistent search behavior between replicas.

Query construction. You decide what text to search for and which filters to apply. The default of using the last user message can miss critical context. A good query may need a rewritten intent, extracted entities, or planner state such as environment, product, or symptom class.

Candidate retrieval. The search layer returns likely matches. The failure mode is either low recall, where the needed evidence is absent, or low precision, where too many weak matches survive.

Reranking and filtering. You refine the candidate set based on metadata, lexical overlap, or more expensive relevance scoring. Failure here often appears as correct records ranked below superficially similar but less useful ones.

Grounding assembly. You choose how many records to place into the prompt and in what order. Even relevant chunks can become ineffective if you overfill the prompt or bury the key evidence in the middle of a long context window.

That final stage deserves special attention. Long context is not a guarantee of reliable use. Models often underuse relevant material when it is surrounded by too much other text, especially when the crucial passage lands in the middle. Retrieval quality therefore depends not only on finding good evidence but also on presenting it economically.

Query construction is where task awareness enters

The default retrieval query source in Koog is the last user message. That is a sensible baseline because it requires no extra machinery and often matches the current need. It also fails in predictable ways. The last user turn may be elliptical, dependent on earlier context, or missing the entities that matter for retrieval. “Can you check that policy again?” is not a useful query by itself.

A robust query construction step often extracts or adds:

- the active task or planner goal;
- entity identifiers such as product, tenant, region, or ticket type;
- symptoms or error classes from recent tool outputs;

- a narrowed document family such as runbook, policy, or FAQ;
- negative constraints such as excluding archived content.

You can implement this without making the pipeline opaque. Keep the original user text, but enrich it with explicit fields and metadata filters derived from the task state. That preserves observability. When a retrieval result is wrong, you can inspect whether the failure came from search quality or from a poor query frame.

Metadata filters are safety controls, not performance tricks

Filters often get discussed as optimization. In production they are governance controls. Tenant boundaries, region boundaries, environment class, document type, language, and supersession status should usually constrain the search space *before* similarity scoring, not after the fact in prompt logic.

For example, if a remediation agent supports both staging and production, environment should be a filter rather than a late textual hint. Otherwise the search layer can surface a staging-only runbook that happens to be semantically similar, and the model may not notice the mismatch. The same logic applies to account tier, customer tenancy, internal versus external documentation, and retired versus active policies.

Well-designed metadata also improves ranking efficiency. The search engine can compare relevant candidates within the right slice of the corpus rather than across everything ever indexed. That reduces both latency and semantic confusion.

Reranking is often where useful retrieval begins

Approximate nearest-neighbor search is good at surfacing plausible candidates quickly. It is not usually good enough by itself for action-critical decisions. Similarity alone cannot distinguish between “how to restart service X” and “when not to restart service X during active mi-

gration," even though the latter may be more important to the current task.

Reranking gives you a second-stage decision. You can score candidates with additional signals:

- lexical overlap with entities or error codes;
- metadata quality and freshness;
- document type preference for the current task;
- source authority, such as canonical runbook over ticket chatter;
- query-document interaction scored by a stronger model.

The point is not to maximize abstract relevance. It is to maximize *action reliability*. A lower-similarity chunk from the canonical policy store may be more useful than a higher-similarity chunk from an anecdotal incident note. Reranking is where you encode that operational preference.

Grounding assembly must be economical

After retrieval and reranking, you still have not solved the grounding problem. You must decide what to place into the model context and how to arrange it. Three practical rules help.

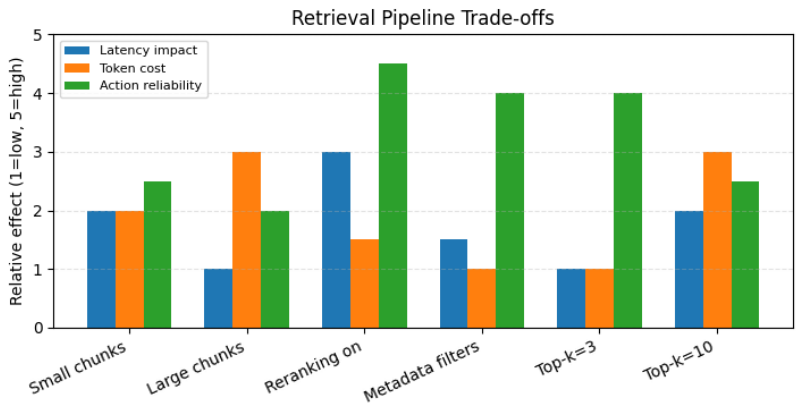
Prefer fewer, stronger chunks. A top-3 set of high-quality records usually beats a top-12 pile of medium-quality ones. More context raises token cost and can dilute the model's attention.

Lead with the most actionable evidence. Put the chunk that most directly constrains the answer or action early in the grounded context. If a policy prohibition exists, surface it before explanatory background.

Separate evidence types. Distinguish canonical policy, procedural instructions, and contextual examples. Mixing them into one blob makes it harder for the model to reason about authority.

This is also the point where summary records can help. A concise synthesized overview paired with canonical source passages often performs better than many raw passages alone. The summary provides orientation; the canonical chunk provides authority. That pairing works only when the summary is maintained and attributable rather than treated as a replacement for primary material.

Trade-offs across pipeline choices



The chart encodes a practical pattern rather than a benchmark claim. Small chunks can improve targeting but often increase retrieval count and assembly overhead. Large chunks reduce candidate count but inflate prompt tokens and can blur the relevant span. Reranking adds latency yet often improves action reliability enough to justify the cost in operational agents. Metadata filtering gives a strong reliability boost with comparatively low latency overhead because it removes unsafe or irrelevant candidates early. A small top-k generally keeps prompts

sharper, while a large top-k increases both token cost and the risk that the decisive evidence gets buried.

These trade-offs should be measured against your actual workload. A support assistant answering low-risk policy questions may tolerate broader top-k and no reranker. A remediation agent choosing among side-effectful actions usually should not.

Hybrid retrieval patterns

Pure dense retrieval is not your only option. Several hybrid patterns work well in practice.

Lexical plus vector retrieval. Dense search handles semantic mismatch; lexical search preserves exact identifiers such as error codes, product names, or configuration keys. Combining them helps when operational language mixes prose with symbolic tokens.

Metadata filter before similarity search. This is often the single highest-value improvement in enterprise corpora. Search the correct slice first, then rank within it.

Summary plus source passage. Keep concise overview documents for orientation, but pair them with canonical passages used for the final grounded answer or action.

Multiple corpora with task-specific selection. Policy, troubleshooting notes, and customer-specific records do not always belong in one index. A planner or retrieval policy can select which corpus to query based on the task phase.

These patterns matter because retrieval is not one operation. It is a policy about how evidence should be assembled for a specific decision.

Evaluating retrieval by downstream effect

The easiest retrieval metric to collect is similarity quality on held-out

queries. It is rarely sufficient. You need evaluation tied to what the agent actually does with the retrieved material.

Useful questions include:

- Did retrieval surface the evidence needed for the correct action?
- Did the grounded answer cite or reflect the authoritative source?
- Did metadata constraints prevent cross-tenant or stale results?
- How often did retrieval misses trigger wrong tool choice, escalation, or hallucinated policy?
- What was the token and latency cost per successful grounded run?

For planner-driven agents, retrieval quality should also be tested at action boundaries. A chunk is useful only if it changes or constrains behavior in the right way. That standard is stricter than asking whether the retrieved text looks relevant to a human reviewer.

Anti-patterns that weaken grounding

Dumping raw chat history into a vector store. Chat transcripts are not automatically good retrieval units. They mix speculation, partial context, and low-authority phrasing. Use them only after curation and metadata discipline.

Indexing uncurated logs. Operational logs can help when transformed into structured evidence, but raw logs are noisy, redundant, and often carry access or retention risks.

Treating RAG as a substitute for state. Retrieval can reintroduce external knowledge; it does not replace explicit workflow state, planner state, or durable user memory.

Using wide top-k as a bandage for poor corpus quality. If the system only works when you stuff ten or twenty chunks into the prompt, the underlying indexing or ranking is probably weak.

Ignoring deletion and supersession. A retrieval store without active maintenance becomes an archive of contradictions. The model cannot reliably infer which version is current unless you encode that policy in metadata and ranking.

Where Koog fits and where you still own the design

Koog gives you a clean architectural split. Chat history stays in memory features designed for conversational continuity. Long-term retrieval-backed memory uses dedicated retrieval abstractions. Search and write paths are explicit. That is exactly the right amount of framework support. It keeps the boundaries clear without pretending that corpus engineering, embedding selection, ranking strategy, and prompt assembly are solved by a single API call.

You still own the parts that determine whether grounded behavior is trustworthy: what enters the corpus, how it is chunked, which metadata fields govern retrieval, how often the index refreshes, how you rewrite queries, and how you decide that the retrieved evidence is strong enough to inform an answer or action. When those decisions are disciplined, retrieval stops being a cosmetic add-on and becomes a reliable external memory substrate that the agent can use without confusing evidence with belief.

6.4. Compressing History for Token-Efficient Continuity

A long-running agent often fails gradually rather than catastrophically. The first few turns are sharp, grounded, and efficient. After enough in-

teraction, the prompt starts carrying old hypotheses, settled subproblems, redundant confirmations, and verbose tool output that no longer matters. Latency rises, token cost rises, and quality drops even though the agent technically has *more* context than before. The problem is not lack of memory. The problem is that raw history is a poor storage format for active reasoning once the session becomes large.

Compression fixes that only if you treat it as a controlled state transformation rather than an after-the-fact summary. You are not merely shrinking text. You are deciding which parts of prior interaction remain operationally relevant, which parts can be rewritten in compact form, which facts must be preserved exactly, and which material should disappear entirely. That makes compression a reliability mechanism, not just a cost optimization.

A concrete bounded-history setup in Koog

Koog already gives you the first necessary control point: chat history preprocessing. `ChatMemory` restores the conversation history for a run, and the memory module includes a sliding-window preprocessor that bounds how much of that history is retained for immediate use. Even when you later add richer summarization logic, this basic windowing step is valuable because it enforces a fixed upper bound on raw replay.

The exact constructor names for every preprocessor variant were not fully verified in the collected API notes, so the stable, documented design point is the one to use here: configure `ChatMemory` with a `ChatHistoryProvider`, then apply a `ChatMemoryPreProcessor` so that only the recent window is carried forward as raw transcript. That gives you a safe baseline even before you add custom summarization.

A minimal provider remains straightforward:

```
import ai.koog.agents.memory.ChatHistoryProvider
import ai.koog.prompt.message.Message

class SessionHistoryProvider(
```

```
 private val data: MutableMap<String, List<Message>>
): ChatHistoryProvider {

 override suspend fun load(
 conversationId: String
 ): List<Message> {
 return data[conversationId].orEmpty()
 }

 override suspend fun store(
 conversationId: String,
 messages: List<Message>
 ) {
 data[conversationId] = messages
 }
}
```

With that boundary in place, you can treat compression as a second layer above basic retention control: keep the most recent turns raw, compress older interaction into a smaller artifact, and retrieve older external facts separately when needed rather than replaying the entire transcript.

What must survive compression

Compression fails when you summarize what was said instead of preserving what still matters. Those are not the same thing. Many conversational turns have little long-term value. A short exchange confirming a file name or acknowledging a completed tool call can disappear with almost no cost. Other details are easy to erase accidentally and expensive to recover: user constraints, unresolved decisions, tool outputs still needed for subsequent actions, safety restrictions, references to external evidence, and facts that should be promoted to durable memory rather than left in the transcript.

A useful retention test is to ask whether a future response or action would become wrong if the detail disappeared. If yes, the information must survive in some form. That form does not have to be the original wording. It may be better represented as a structured field, a compact

event record, or a validated summary sentence. The operational objective is continuity of *relevant state*, not fidelity to every turn of dialogue.

For most long-running agents, the material that must survive falls into five classes:

- *open commitments*: promises, pending tasks, approvals, and unresolved questions;
- *stateful facts*: identifiers, confirmed preferences, resource names, environment details, and validated tool findings;
- *safety constraints*: forbidden actions, approval requirements, policy restrictions, and risk boundaries;
- *decision provenance*: why an important choice was made, especially if it depends on retrieved or user-supplied evidence;
- *local discourse anchors*: enough immediate context to keep references like "that deployment" or "the second option" unambiguous.

Everything else is a candidate for eviction, reduction, or regeneration.

Compression is different from memory and retrieval

The distinction matters because these mechanisms can look similar in code while serving different control objectives.

Short-term memory keeps working context active. Compression changes the representation of that context so it still fits within a bounded prompt budget. Long-term memory preserves durable facts that should outlive the current session. Retrieval reintroduces external knowledge from a source corpus when needed. If you compress conversation history and then treat the result as durable memory, you risk storing accidental speculation as fact. If you use

retrieval to compensate for poor compression, you may re-fetch external documents while still losing the user commitments that were never part of the external corpus.

A healthy long-running agent usually carries four distinct context bands into a model call:

1. a stable system and task prefix;
2. a recent raw interaction window;
3. a compact rolling summary or extracted state record;
4. retrieved external evidence relevant to the current step.

That layered arrangement is much more robust than either full transcript replay or aggressive summarization alone.

Compression strategies and when to use them

Not all compression strategies fail in the same way. Choose the representation that matches the kind of continuity you need.

Rolling narrative summary. This is the common default: periodically condense older turns into a prose summary. It preserves broad conversational continuity and is easy to read in debugging traces. It is also lossy. Narrative summaries flatten uncertainty, drop edge cases, and often omit exact values unless you force the summarizer to retain them.

Structured state extraction. Instead of summarizing a conversation as prose, extract fields such as active goal, confirmed constraints, pending tasks, entities, and last authoritative tool results. This is usually safer for planner-driven or operational agents because the important parts are explicit. The downside is that discourse texture is reduced; the agent may answer correctly but sound less conversationally aware unless you pair structure with a small recent raw window.

Event log compression. Preserve a compact sequence of normalized events such as "ticket escalated," "restart failed," or "user approved rollback." This works well when chronology matters and the domain has a natural event vocabulary.

Hierarchical or recursive summarization. Summarize segments, then summarize summaries. This extends continuity across very long sessions, but drift accumulates quickly if you do not validate each layer. A summary of a summary is not equivalent to a shorter view of the original interaction; it is a second transformation with its own omissions and distortions.

Selective replay. Keep exact snippets for specific classes of turns while compressing the rest. This is effective for preserving legally sensitive wording, exact user instructions, or canonical tool outputs. It requires a classifier or rule system to decide what deserves verbatim retention.

Each strategy trades readability, precision, token cost, and governance effort differently. Narrative compression is fast to adopt but easy to over-trust. Structured compression takes more design work but gives you stronger guarantees around commitments and control state.

A practical operational flow

A useful compression pipeline follows a strict order. If you compress opportunistically whenever the conversation feels long, you will eventually summarize unstable material and canonize partial facts. A better flow is:

1. detect budget pressure or a compression threshold;
2. classify historical material by salience and authority;
3. identify unresolved commitments and stateful facts;
4. wait until in-flight tool actions settle;

5. produce a bounded summary or extracted state artifact;
6. validate the artifact against critical fields;
7. store the compressed artifact and trim the raw window;
8. feed the next run with recent raw turns plus the artifact.

The threshold can be token-based, turn-count-based, latency-based, or phase-based. A token threshold is usually the most reliable because it aligns directly with the failure mode you are managing. If the active prompt budget is 12,000 tokens and the recent raw history plus system prompt already consumes 9,000, you need compression before retrieval or additional tool output arrives. Turn count alone is too coarse because ten short turns and ten verbose tool-heavy turns behave very differently.

Salience classification is the hard part

The central design problem is not summarization quality. It is deciding what deserves retention before you summarize. If that classification is weak, every downstream representation will be weak.

You can classify dialogue material along three axes:

Operational relevance. Will the next response, tool choice, or plan step depend on this item?

Durability. Is this detail likely to remain relevant beyond the immediate exchange?

Authority. Did this detail come from the user, a validated tool result, a retrieved external source, or a model inference?

These axes lead to concrete behavior. High relevance and high authority suggest retention. High durability suggests promotion into long-term memory rather than leaving the detail buried in a summary. Low

authority suggests the detail should remain provisional or disappear entirely.

For example, “the user approved deleting test artifacts” is high authority and high operational relevance, so it should survive compression explicitly. “The model speculated that the issue might be DNS” is low authority and should not become canonical state unless validated by later evidence. “The deployment artifact hash was abc123” may be high authority but low durability; it belongs only as long as the current task depends on it.

Narrative summary versus extracted state

If your agent mainly answers questions and does not drive complex tools, a compact narrative summary often works well enough. It preserves topic flow, user intent, and the local story of the interaction. That makes responses feel coherent across many turns.

Planner-driven and tool-heavy agents usually need more than narrative continuity. They need exact state. A planner deciding whether to continue remediation cannot rely on a prose sentence like “the system tried a few fixes and the user wanted a careful approach.” It needs explicit facts such as restart attempted, approval required for rollback, latest health check failed, and escalation not yet opened.

A strong pattern is to split the compressed artifact into two parts:

- a *state capsule* holding structured commitments, constraints, identifiers, and authoritative outcomes;
- a *conversation capsule* holding a brief narrative summary of goals, rationale, and discourse context.

That split keeps the model grounded in exact state without losing the conversational thread entirely.

Validation is mandatory

Summaries are attractive because they feel clean. They are dangerous because they are hard to challenge after the original turns are evicted. If the summary drops a constraint or invents confidence that the original conversation never justified, future responses can remain consistently wrong for many turns.

Validate compressed artifacts before you treat them as canonical short-term state. Validation can be lightweight and still useful:

- compare extracted entities against the original window;
- verify that unresolved tasks in the summary still exist in the task tracker or planner state;
- check that exact values such as dates, IDs, and limits were copied accurately;
- ensure safety constraints and approval requirements are present;
- reject summaries that convert uncertainty into certainty.

For high-stakes workloads, keep a small audit tail of raw turns even after compression so you can reconstruct how the summary was formed. You do not need to replay that tail into every model call, but you do want it available for debugging or dispute resolution.

When to compress in the lifecycle

Timing matters. Compress too late and you pay excess latency and token cost. Compress too early and you summarize unstable interaction before tool outcomes settle. The worst moment to compress is while actions are still in flight and the conversation contains provisional assumptions that may be overturned by a pending result.

A safe rule is to compress at *stable boundaries*: after a tool call completes, after a user approval or rejection is recorded, after a planner iteration ends, or after a response turn closes without unresolved stream fragments. If your system streams partial assistant output, do not treat partial text as durable history until the final message is complete. Partial generations are especially dangerous because they often contain exploratory reasoning that should never become canonical context.

This is also where checkpointing and compression must stay separate. A checkpoint persists execution state so a run can resume. Compression transforms conversational context so future prompts remain efficient. They may occur near the same boundaries, but they should not write to the same logical store.

The limits of larger context windows

A natural response to compression complexity is to avoid it and rely on larger model context windows. That helps, but only up to a point. Large windows reduce the need for immediate truncation; they do not remove the need to rank, organize, and condense information. Models often underuse relevant content when it is buried in the middle of a long prompt, and long windows amplify the cost of carrying redundant or stale material.

That has two practical consequences. First, prompt ordering matters: important retained state and retrieved evidence should be placed where the model will reliably attend to them. Second, compression remains useful even when budget is abundant, because smaller and better-curated context often performs better than larger undifferentiated context.

A layered continuity pattern

A robust long-session prompt usually looks like this in logical order:

1. stable instructions and task framing;
2. compressed state capsule;
3. compact narrative summary if needed for discourse continuity;
4. recent raw user and assistant turns;
5. current tool outputs;
6. retrieved external evidence for the active step.

This order is not arbitrary. Stable instructions should remain fixed so the model and any provider-side caching can benefit from a consistent prefix. Compressed state should appear early because it carries the facts that govern the next step. Recent raw turns preserve local language and reference resolution. Retrieval comes late enough to be task-specific but not so late that it gets ignored after a large noise block.

If you reverse this order and bury the compressed state behind pages of raw history or retrieved text, you lose much of the value of compression.

Failure modes and anti-patterns

Recursive summarization without validation. Each layer of summarization compounds omissions and distortions. Without checks, the agent drifts into a polished but inaccurate account of the session.

Compressing before tool outcomes settle. This captures speculative state as though it were final. Wait for authoritative outcomes or explicitly mark pending items as unresolved.

Treating model inferences as canonical facts. A summary can note hypotheses, but it must label them as hypotheses. Never silently upgrade inference to fact.

Preserving conversational fluff while dropping constraints. Many generic summarizers retain the visible topic and remove the hidden operational details that actually matter, such as approvals, deadlines, or environment restrictions.

Using one summary for every purpose. The representation needed for planner state is not identical to the representation needed for user-facing conversational tone. Distinguish them.

Letting prompts implicitly decide retention policy. If the summarizer prompt says "keep important information" but you never define important, you have outsourced a core systems decision to vague language.

Choosing the right compression policy

You can usually select a reasonable policy by looking at five signals.

Session length. Longer sessions justify more structured compression and stronger validation because summary drift has more time to accumulate.

Tool density. The more tool outputs and planner state changes the conversation contains, the less safe pure narrative summarization becomes.

Auditability. If you must explain later why the agent acted as it did, retain explicit commitments, approvals, and authoritative outputs in structured form.

Risk of wrong recall. If a missed constraint can trigger the wrong action, compress conservatively and validate aggressively.

User experience goals. If conversational tone matters, pair structure with a small recent raw window or a compact narrative capsule so the agent does not sound mechanically amnesiac.

A support chatbot answering low-risk informational questions can tol-

erate a lightweight rolling summary. A remediation assistant with approval gates and tool side effects usually needs structured extracted state plus selective replay of critical turns.

What Koog gives you and what you still design

Koog already gives you a meaningful starting point through chat history preprocessing and sliding-window retention. That is enough to keep short-term memory bounded and stop the most obvious transcript growth problem. The more advanced parts of compression remain your design responsibility: what to summarize, how to represent retained state, when compression occurs, and how you validate the result before reusing it.

That is the right division of labor. Framework support should expose hooks and enforce memory boundaries; it should not hide the fact that compression is application policy. The agent stays coherent over long interactions when you deliberately preserve active commitments, exact constraints, and authoritative outcomes while compressing everything else into a representation sized for the next decision rather than for historical completeness.

6.5. Implementing Checkpointing and State Restoration

A long-running agent becomes operationally serious the moment you admit that it will be interrupted. Processes crash. Hosts are evicted. deployments roll forward while a task is mid-flight. Tool providers time out after a side effect may already have happened. A human approver may pause the run for hours and expect it to continue later without repeating work. At that point, prompt history and durable memory are no longer enough. You need execution durability: a way to preserve where the run was, what state it had already committed, and how to

resume without duplicating or corrupting side effects.

That is the purpose of checkpoints. A checkpoint is not a chat transcript, not a vector record, and not a user profile. It is a durable snapshot of execution state at a recovery boundary. If you blur those categories, restoration becomes dangerous. The agent can rehydrate stale conversation without knowing which tool action already fired, or it can resume a planner loop from an invalid state because the checkpoint captured text but not control position.

A concrete Koog persistence setup

Koog exposes checkpointing through the `Persistence` feature. The older `Persistency` name is deprecated and removed, so use the documented install name exactly. The official sample also shows the most important configuration choices: a storage provider, automatic persistence, and optional rollback behavior.

```
import ai.koog.agents.snapshot.feature.Persistence
import ai.koog.agents.snapshot.feature.RollbackStrategy
import ai.koog.agents.snapshot.storage.
     InMemoryPersistenceStorageProvider

install(Persistence) {
    storage = InMemoryPersistenceStorageProvider()
    enableAutomaticPersistence = true
    rollbackStrategy = RollbackStrategy.Default
}
```

That example is intentionally small. `InMemoryPersistenceStorageProvider` is useful for tests and process-local experiments. Production systems should back persistence with storage that survives process death and can be shared across instances if resumption may happen on a different host.

Koog's persistence feature is designed for resumable execution. The documented methods include `createCheckpoint`, `getCheckpointById`, `getLatestCheckpoint`, `rollbackToCheckpoint`,

`rollbackToLatestCheckpoint`, `saveCheckpoint`, and `setExecutionPoint`. Those names reveal the intended model: checkpoint creation is explicit, recovery targets can be addressed directly, and restoration can reset both state and execution position.

What a checkpoint is and is not

A checkpoint captures execution-relevant state such as message history, current node, and input data. That is already enough to distinguish it from the other state layers used by an agent.

Short-term memory exists to make the next interaction coherent. It answers, "What was recently said and what still matters conversationally?"

Long-term memory exists to preserve durable facts that should survive across sessions. It answers, "What should the agent remember later about this user, task, or domain?"

Retrieval exists to fetch external knowledge on demand. It answers, "What evidence should be brought into context right now?"

Checkpointing exists to resume an interrupted run safely. It answers, "Where was execution, what had become authoritative, and what must not be repeated blindly?"

This is more than a conceptual tidy-up. The storage contract, retention policy, and correctness tests differ sharply. A chat summary can be approximate and still useful. A checkpoint that misstates whether a tool call completed can cause duplicate side effects. Retrieval can be probabilistic. Restoration cannot.

Recovery boundaries are a design decision

You do not get reliable restoration by turning on automatic persistence and hoping the right state appears. You need explicit recovery boundaries. A recovery boundary is a point in execution where the system

can stop, persist, and later resume or replan without ambiguity.

Good boundaries share three properties:

- the state up to that point is internally consistent;
- any external side effects before that point are known or attributable;
- the next step after restoration is clear.

That usually places checkpoints in four common locations:

Before side-effecting tool calls. Persist the intent and inputs before a risky action begins. If the process dies during the call, you have a durable record of what was attempted.

After validated state transitions. Once a tool result has been confirmed and incorporated into authoritative state, persist again so restoration does not need to recompute the same work.

At approval gates. If a human must authorize the next step, checkpoint before waiting and after receiving the decision.

At planner re-evaluation points. A planner run can move through multiple inferred steps. Persist at points where the plan can be safely recomputed from current state if necessary.

These boundaries matter more than checkpoint frequency alone. Frequent checkpoints written at unstable or ambiguous moments simply preserve confusion more often.

A practical checkpointing flow

A production-safe checkpoint flow is usually more disciplined than the naive pattern of "run step, then save state." A better sequence is:

1. capture the current authoritative execution state;

2. write a checkpoint before initiating a risky or expensive step;
3. execute the guarded step with idempotency or correlation data;
4. record the observed outcome and any external receipt;
5. mutate state only after outcome validation;
6. write the next checkpoint with the committed post-step state;
7. on restoration, inspect whether the guarded step completed, partially completed, or remains unknown;
8. either resume, compensate, or replan.

That pattern separates *intent*, *attempt*, and *commit*. Without that separation, you cannot tell after a crash whether the agent merely planned to call a tool, actually called it, or already integrated the result.

What must be persisted

The exact payload depends on your workflow, but several categories recur.

Execution position. For graph-based runs, persist the current node or step identity. For planner-driven runs, persist enough state to decide whether to continue the old plan or derive a new one. Koog's persistence feature exposes `currentNodeId` and `setExecutionPoint`, which makes the control-position aspect explicit.

Authoritative message history. If the interaction history affects future behavior, persist the committed message history that belonged to the run at that boundary. This is not a replacement for chat memory design; it is the history relevant to recovery.

Input data and resolved parameters. Persist the request, extracted entities, resolved targets, and other inputs that define what the run

was doing. Restoration should not depend on re-parsing ambiguous upstream input.

Tool receipts and correlation IDs. If a tool call can create an external side effect, persist request identifiers, idempotency keys, external operation IDs, or tickets created by the step. These are often the only way to determine after a restart whether the action already happened.

Retry and attempt state. Persist counters or status markers that prevent the system from looping indefinitely on a failing step.

Planner or approval context when relevant. If the next decision depends on an approved action, a selected strategy, or a plan hypothesis, persist that context in a form you can validate later.

A useful rule is to persist what would be expensive, unsafe, or ambiguous to reconstruct. Do not persist environment-specific handles, live network clients, or objects whose semantics exist only in process memory. Those should be rebuilt on restoration from durable identifiers, not serialized as if they were state.

Rollback strategy is a policy choice

Koog exposes two documented rollback modes: `RollbackStrategy.Default` and `RollbackStrategy.MessageHistoryOnly`. They represent two distinct operational philosophies.

`Default` restores the full saved state, including message history and other stateful data captured by persistence. Use it when you need exact continuation and when the saved state remains compatible with the running code and tool environment.

`MessageHistoryOnly` restores only saved message history and restarts from the first node. Use it when exact workflow continuation is risky because code, tools, or state contracts may have changed across deployments. This is safer but less precise. You preserve conversational con-

tinuity while accepting recomputation of execution logic.

That trade-off becomes especially important across releases. If the agent code changed between checkpoint creation and restoration, full rollback may reconstruct a state that newer nodes no longer understand. In those cases, restarting from the beginning with preserved message history can be the safer choice, especially when the workflow is designed to be idempotent.

Restoration and side-effect safety

Most difficult recovery bugs are not about reading the checkpoint. They are about deciding what to do next without repeating a side effect. Consider a payment reversal, an infrastructure change, or an external ticket creation. If the process crashes after sending the request but before writing the success checkpoint, restoration faces uncertainty: did the side effect happen?

There are only a few safe strategies.

Idempotent external actions. The best option is to make the tool operation idempotent by supplying a stable idempotency key or external correlation ID. Then replaying the call is safe because the target system will treat it as the same logical operation.

External receipt lookup. If the tool provides a query API, persist the operation identifier and check status on restoration before deciding whether to retry.

Compensation. For reversible workflows, restore and execute a compensating action instead of blind replay. This is safer when side effects are not inherently idempotent.

Human escalation. If the side effect cannot be verified automatically and repeating it is dangerous, stop and require human review.

Koog's persistence feature can be paired with a rollback tool registry

for compensation-aware recovery. Even when you do not use explicit compensating tools, the design lesson stands: restoration logic must be side-effect-aware, not just state-aware.

Planner-driven restoration

Planner-driven agents add one more decision after restoration: do you continue the prior plan or replan from the restored state? There is no universal answer. Use the old plan only when the assumptions behind it are still valid and the execution environment has not drifted in a meaningful way.

Three questions help decide:

Did the world change while the run was paused? If external conditions may have drifted, replanning is safer.

Did the code or tool surface change? If actions, schemas, or policies changed between checkpoint and restore, exact continuation may no longer be valid.

Is the plan itself authoritative or merely advisory? In many systems the plan is a transient optimization over current state. In that case, restoring the state and recomputing the plan is often preferable to restoring the plan object itself.

For planner-heavy systems, a strong pattern is to checkpoint the *world state*, latest authoritative observations, and current execution point, then re-enter the planner loop from there. That keeps restoration aligned with the actual environment rather than with a stale search result.

Granularity: how often should you checkpoint?

Checkpoint too rarely and recovery becomes expensive or unsafe because too much ambiguous work lies between boundaries. Checkpoint too often and you pay storage overhead, I/O latency, and operational

complexity for little gain. The right granularity depends on five concrete signals:

Side-effect risk. The more dangerous or irreversible the next action, the closer the checkpoint should be to that action boundary.

Cost of recomputation. Expensive retrieval, long tool chains, and human approval waits justify tighter checkpointing.

Recovery time objective. If the system must resume quickly after interruption, finer-grained checkpoints reduce the amount of repeated work.

State volatility. If state changes rapidly during a run, checkpoint at stable transitions rather than at arbitrary clock intervals.

Storage and throughput budget. High-volume systems need checkpointing discipline that matches their persistence infrastructure.

A practical compromise is event-driven checkpointing rather than timer-driven checkpointing. Persist at meaningful control events: before external mutation, after authoritative completion, and at approval or planner boundaries. Those points align durability cost with business risk.

Serialization boundaries and schema hygiene

Checkpoint data must survive both process restart and software evolution. That means you should treat checkpoint schemas as durable contracts, not incidental implementation details. Persist simple, explicit data structures: identifiers, scalar fields, structured state records, message history, and version tags. Avoid serializing live object graphs or provider-specific handles that will not deserialize cleanly across environments.

Version the checkpoint payload deliberately. Even if the persistence API abstracts storage, your application state may evolve. A version

field, migration path, or compatibility shim can turn a risky restore into a controlled downgrade or replan. Without that discipline, old checkpoints become operational landmines during deployments.

This is where rollback mode interacts with schema evolution. Full-state restoration is powerful when schemas are stable. Message-history-only rollback gives you a fallback when they are not.

Automatic persistence is helpful, not sufficient

Enabling `enableAutomaticPersistence = true` is useful because it reduces the risk that a run moves through critical steps without ever writing recovery state. It should not replace explicit checkpoint design. Automatic persistence cannot know which external side effects need idempotency treatment, which planner state is still valid after a pause, or which approval gates define meaningful restoration boundaries in your domain.

Treat automatic persistence as the baseline safety net. Then add explicit checkpoint placement around the steps that matter most. If your workflow mutates external systems, waits on human input, or mixes planner reasoning with tools, explicit boundaries remain essential.

Common failure modes

- Duplicate side effects after restart. The system restores and replays a tool action without checking whether the original request already succeeded. Prevent this with idempotency keys, external receipts, or human review.
- Persisting non-serializable state. A checkpoint tries to capture process-local handles, client objects, or streaming state that cannot be restored meaningfully.
- Restoring stale planner assumptions. The checkpoint rehydrates a plan whose predicates no longer match the current environ-

ment.

- Checkpointing too late. A crash occurs after a side effect but before the state that proves it was done is persisted.
- Checkpointing unstable partial output. Streaming fragments, provisional summaries, or incomplete tool responses are stored as authoritative state.
- Confusing checkpoint data with memory data. Durable user facts or retrieval evidence get mixed into execution checkpoints, making both lifecycle and debugging harder.

Each failure mode has a predictable root cause: the system persisted data without first deciding what role that data played in recovery.

A compact restoration checklist

When a run resumes, make the restore path explicit rather than implicit. A simple sequence works well:

1. load the latest checkpoint for the agent and task;
2. validate schema and compatibility;
3. inspect execution point and last known authoritative outcome;
4. verify external side effects using receipts or idempotency keys;
5. decide whether to continue, compensate, restart, or replan;
6. restore state using the chosen rollback strategy;
7. resume only after the replay decision is justified.

That checklist may feel heavier than naive restart logic, but it transforms restoration from a best-effort convenience into a governable operational path.

Where checkpointing fits in the larger state model

Checkpointing becomes much easier once each state layer has one responsibility. Short-term memory preserves conversational continuity. Long-term memory preserves durable reusable facts. Retrieval brings in external evidence at query time. Compression keeps prompt history small enough to remain useful. Checkpoints preserve executable state at boundaries where interruption would otherwise create ambiguity.

That separation lets you debug failures cleanly. If the restored run chose the wrong tool because it forgot a user constraint, the fault may lie in memory or compression. If it repeated a side effect after restart, the fault lies in checkpoint placement or idempotency design. If it resumed with stale policy context, the fault may lie in retrieval freshness rather than persistence. Once those boundaries are explicit, a resumable Koog agent stops being a fragile conversation with tools and starts behaving like a recoverable distributed system that happens to reason in natural language.

Chapter 7

Observability, Reliability, and Production Operations

Production agent systems rarely fail in the clean, obvious ways that sample code suggests; they degrade through partial traces, ambiguous tool behavior, hidden retries, budget drift, and state that can no longer be trusted. This chapter gives the reader an operator's mental model for Koog in the wild: instrument everything that matters, distinguish transient faults from structural ones, recover only from safe boundaries, and treat performance and cost as observable engineering constraints rather than after-the-fact surprises.

7.1. Tracing as a First-Class Development Practice

A user tells you the agent "sometimes gets stuck." The HTTP access log shows a request arrived, the process stayed alive, and the request eventually timed out. That is enough to prove the system was busy, but not enough to explain what it was doing. You cannot tell whether the model kept revising a plan, whether a tool call blocked on a downstream service, whether a workflow branch looped, or whether execution resumed from an incomplete checkpoint and drifted into a dead path. Ordinary request logging collapses a structured run into a start line and an end line. Agent systems fail inside the space that logging flattens.

Treat tracing as the primary execution record for any nontrivial Koog agent. The unit you need to observe is not a single RPC or servlet request. It is a causally connected run composed of workflow steps, model calls, tool invocations, retries, state transitions, and outcomes. A trace gives you that structure. Each span records one bounded operation. Parent-child relationships show how the run decomposes. Attributes describe the operation in queryable form. Events mark notable moments inside a span without turning the span into a dump of unstructured text. Once you adopt that model, observability stops being a debugging afterthought and starts influencing how you shape workflow boundaries, how you name operations, where you persist state, and what information you allow to cross trust boundaries.

A minimal Koog setup should emit spans as early as possible so you can validate that runs are being captured before you design richer telemetry. The following example installs Koog's OpenTelemetry feature and sends spans both to a local logging exporter and to an OTLP endpoint:

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.agents.features.opentelemetry.feature.OpenTelemetry
import io.opentelemetry.exporter.logging.LoggingSpanExporter
```

```
import io.opentelemetry.exporter.otlp.trace.OtlpGrpcSpanExporter

val agent = AIAgent(
    executor = simpleOpenAIExecutor(
        System.getenv("OPENAI_API_KEY")
    )
) {
    install(OpenTelemetry) {
        addSpanExporter(LoggingSpanExporter.create())
        addSpanExporter(
            OtlpGrpcSpanExporter.builder()
                .setEndpoint("http://localhost:4317")
                .build()
        )
    }
}
```

That snippet is intentionally small. It does not solve naming, redaction, sampling, or workflow semantics. It proves a more basic point: if you cannot see the run shape while the system is still small, you will discover your blind spots under incident pressure instead of during development.

Core vocabulary A *trace* is the full execution record for one logical run. A *span* is one timed operation within that trace. A parent span encloses broader work; a child span captures a subordinate step. A *correlation identifier* links the trace to domain objects such as a user request, job ID, ticket number, or document batch. *Context propagation* carries trace identity across function calls, coroutines, queues, remote providers, and tool boundaries. *Causal context* means you can answer not just what happened, but what caused a later step to exist at all.

Those definitions matter because agent behavior is branchy and nested. If a model span appears with no visible parent workflow span, you know little about its purpose. If a tool span lacks a link to the model output that generated its arguments, you cannot reconstruct whether the tool call was correct or merely successful. If a retry produces another provider call but carries no relation to the failed attempt, your

backend shows two isolated slow calls instead of one operation under recovery. Good tracing preserves execution boundaries and causality at every control edge.

For agent systems, a few boundary types deserve explicit attention.

Run boundaries Create a top-level run span at ingress. This span should correspond to the business-visible unit of work: answer one user request, process one document, complete one workflow instance, or reconcile one external record. Do not let the top-level span default to whichever framework callback happens to fire first. You want a run boundary that remains stable even if transport or deployment mechanics change.

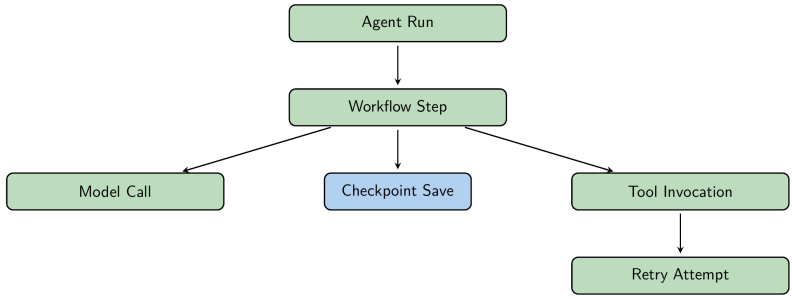
Workflow boundaries Every meaningful workflow node, planner transition, branch decision, or checkpoint restore should appear as a child span or a structured event on a workflow span. If execution can fork or loop, those edges must become visible in telemetry. Otherwise a timeout only tells you that the run consumed wall-clock time, not which control path consumed it.

Model boundaries Model interaction deserves its own spans because it is a high-latency, high-variance boundary with provider-specific failure modes. Treat prompt assembly, provider round-trip, streaming delivery, and post-processing as distinct operations when they differ operationally. A single coarse span called “llm” is not enough once you need to isolate prompt inflation, slow first token latency, or structured-output repair attempts.

Tool boundaries A tool invocation is not just another function call. It is a contract boundary between generated intent and executable side

effect. You need to see the tool name, schema or operation shape, argument validity, execution duration, result class, and any side-effect identifier that makes replay auditable.

State boundaries Checkpoint writes, state hydration, and state mutation points deserve tracing because they determine whether a run can be resumed safely. If the run resumed from durable state, record that fact. If a branch decision depended on restored memory or prior workflow state, expose that dependency. Hidden state is where many "random" agent failures actually live.



What must be visible in a run You need enough information to reconstruct intent, control flow, timing, and outcome without turning traces into raw transcripts. That balance is the central discipline.

At minimum, capture a stable run identifier, tenant or environment identity when relevant, the workflow or agent name, the model provider and model identifier, the operation name for each span, start and end times, success or error status, and enough attributes to explain why a span existed. For a model call, that usually includes

the request model, operation type, prompt template or prompt assembly path identifier, token or message counts when available, and output type. For a tool call, it includes the tool name, argument validation status, side-effect class, and result category. For a workflow transition, it includes the source node, destination node, decision reason category, and whether the step consumed restored state.

Do not confuse visibility with indiscriminate payload capture. If you attach full prompts, full model outputs, full retrieved documents, and full tool arguments to every span, you make three things worse at once: privacy exposure, backend cost, and query usability. Large opaque blobs are difficult to aggregate and dangerous to retain. The better pattern is to separate *identity*, *summary*, and *content*. Put stable, low-cardinality identifiers and summaries in span attributes. Put short, time-local markers in events. Keep sensitive or bulky content either redacted, sampled, hashed, or stored outside the trace backend behind stricter controls.

Prompt and response visibility Prompt behavior still needs visibility because prompt assembly errors are common and expensive. The safe approach is to trace the prompt assembly process rather than defaulting to permanent raw capture. Record which template or system instruction variant was used, which retrieval set or memory segment contributed, the input size, and whether the content passed a redaction policy. If you need debug-grade content inspection, make it selective: enable it for controlled environments, sampled runs, or short-lived incident windows. Treat response capture similarly. Preserve enough structure to explain the operation outcome, but avoid turning every trace into an archive of user data.

Events versus attributes Use attributes for data you want to filter, group, and aggregate. Use events for moments inside the opera-

tion: “stream_started”, “tool_args_validated”, “checkpoint_written”, “retry_scheduled”, “resume_loaded”. This distinction sounds minor until you operate at scale. If you store every internal moment as a new child span, the trace becomes noisy and expensive. If you store every detail as one giant text attribute, the trace becomes unreadable and difficult to query. A disciplined split between spans, events, and attributes keeps traces structural.

Naming and semantic discipline Span names should describe the operation, not the implementation detail of the current code path. Prefer names such as `agent.run`, `workflow.step`, `model.generate`, `tool.invoke`, and `state.checkpoint`. Stable names let you compare runs over time even if the internal code changes. Attach provider-specific or model-specific variation as attributes, not as a combinatorial explosion of span names.

Keep high-cardinality values out of the fields you expect to aggregate. A user ID or raw prompt hash may be useful for forensic lookup, but it is a poor primary dimension for dashboards and alerting. Design attributes the way you design a schema for operational data: stable keys, predictable value classes, and enough consistency to survive backend migration.

Tracing changes design decisions early Once you commit to reconstructable runs, several design pressures appear immediately.

You stop hiding workflow decisions inside prompt prose. If the model decides between “search”, “summarize”, and “escalate”, that decision needs a named boundary in telemetry. Otherwise you cannot explain branch frequency, branch latency, or branch-specific failures. This often leads you to externalize routing and planner output into explicit workflow nodes rather than burying them in one large model exchange.

You become stricter about tool contracts. If a tool call must be visible with a name, schema version, argument validation result, and outcome, then the tool interface itself tends to become cleaner. Ambiguous "misc" tools and loosely typed payloads are hard to trace well because they are hard to reason about.

You place persistence boundaries more deliberately. A checkpoint span forces a question: what exact state became durable here, and what would replay mean after this point? If you cannot answer that operationally, the checkpoint is not yet a safe recovery boundary.

You also discover where correlation breaks. Asynchronous callbacks, background jobs, and remote tool services often sever trace context unless you propagate it deliberately. That is not just a telemetry defect. It is usually evidence that your execution boundary is underspecified.

A practical tracing flow A workable operating pattern is simple enough to apply on every project.

First, define the run at ingress. Assign or adopt a correlation identifier that matches the domain unit of work. Create the top-level span there, not after several internal layers have already run.

Second, propagate context through every boundary that can hide causality: coroutines, dispatchers, queues, tool executors, and remote service calls. If a step can outlive the original request thread, it can also lose trace identity.

Third, add spans at every control boundary: workflow node entry and exit, model request, tool invocation, checkpoint write, checkpoint restore, and any explicit retry attempt. Keep these spans coarse enough to reflect meaningful operations, but not so coarse that one span contains several independent failure modes.

Fourth, attach domain-relevant attributes at span creation time. If

you rely on sampling, some attributes need to exist before the operation completes. Model provider, model ID, workflow node name, tool name, and operation class are usually worth setting early.

Fifth, record notable events inside long-lived spans. For example, mark prompt assembly completed, stream first chunk received, tool arguments validated, checkpoint committed, or retry backoff scheduled.

Sixth, review failed traces by asking causal questions in order: where did the run spend time, which branch fired, what state did that branch depend on, did a tool contract mismatch appear before the visible failure, and did any retry alter inputs or execution path.

A concrete failure-reading pattern Suppose a run times out after ninety seconds. A good trace lets you see a top span for the agent run, a child workflow span for the planning step, a model span that completed quickly, a tool span for an external search, and then a retry span under that tool call because the first attempt hit a transient upstream error. You can then inspect whether the retry reused the same arguments, whether the backoff was bounded, and whether the workflow transitioned after the retry or stalled waiting for a result. If execution later restored from a checkpoint, that restore should appear explicitly so you can distinguish a genuine continuation from an improvised replay.

Without those boundaries, you would only know that the request started and later timed out. With them, you can answer whether the problem was provider latency, tool instability, planner indecision, or a state-management bug.

Anti-patterns that make traces operationally useless The first anti-pattern is tracing only provider calls. That yields latency data, but not run structure. You may know the model took eight seconds

without knowing which workflow node asked for the call or what tool action it was trying to prepare.

The second is treating logs as a substitute for spans. Logs can complement traces, especially for local detail, but they rarely preserve hierarchy and causality by themselves. A page full of timestamped messages is not a run model.

The third is dumping raw prompts and outputs without structure. This creates privacy risk and still fails to answer aggregate questions such as which prompt variant produced a branch error or whether a retry changed the tool arguments.

The fourth is omitting state transitions. If the agent depends on retrieved memory, cached planner state, or durable checkpoints, then hidden state produces phantom behavior. Trace the load, use, and persistence of state in a bounded way.

The fifth is allowing trace schemas to drift between teams or services. If one tool span records a field called `tool.name` and another records `actionName`, cross-run analysis becomes brittle. Stable semantics matter more than a polished dashboard.

Privacy-aware telemetry boundaries The more capable the agent, the more likely your traces touch sensitive material. That includes user inputs, retrieved enterprise content, tool arguments that contain identifiers, and model outputs that may reflect private context. Build redaction into the tracing design rather than relying on ad hoc post-processing.

A practical approach is to classify fields into three groups. *Safe operational metadata* includes names, durations, statuses, bounded identifiers, and schema versions. These belong in traces by default. *Sensitive but sometimes necessary content* includes prompts, outputs, tool argu-

ments, and retrieved snippets. Capture these only under explicit policy: sampled, masked, development-only, or routed to restricted storage. *Forbidden content* includes secrets, raw credentials, regulated identifiers, and fields with no operational diagnostic value. These should never enter telemetry.

That policy should shape instrumentation decisions early. If a tool argument schema contains both a customer identifier and a free-text note, structure the telemetry so you can record the schema path, validation status, and note length without automatically exporting the note itself. Tracing quality improves when data contracts are designed for observability, not retrofitted after incidents.

Compatibility and schema drift Trace payload shape is not completely static. Koog instrumentation hooks, provider SDK behavior, and generative AI semantic conventions evolve. Design your operational dashboards and alert rules around stable concepts: run, workflow step, model operation, tool invocation, checkpoint, status, duration, and error class. Keep provider-specific or evolving attributes additive rather than foundational. If you later opt into richer GenAI conventions or export to a specialized backend, the core run model should remain intact.

Validation during development Do not wait for production traffic to assess tracing quality. During development, inspect traces for three properties.

First, *completeness*: every meaningful control boundary appears exactly once in the expected parent-child location.

Second, *causality*: if a tool call exists, you can identify which workflow step and which model output produced it.

Third, *boundedness*: traces remain readable and affordable. Payload size stays under control, high-cardinality attributes are deliberate, and sensitive content follows policy.

When those properties hold, the trace is useful for both engineering and operations. You can diagnose incidents, compare workflow variants, detect loop paths, reason about latency budgets, and audit recovery behavior with one execution record instead of reconstructing the story from disconnected logs.

Serious Koog development starts there. If a run cannot be reconstructed with confidence, every later discussion about retries, failure classification, export pipelines, and performance tuning rests on guesswork rather than evidence.

7.2. Instrumenting Koog with OpenTelemetry and External Observability Backends

A team usually discovers its observability architecture at the worst possible time: the agent works well enough in development, then a production incident forces everyone to answer simple questions that the telemetry cannot support. Which workflow branch ran? Did the tool call fail locally or in a downstream service? Did the agent retry with changed inputs? Was the slow path inside the model provider, the retrieval stage, or a queue between services? If your trace data is trapped in one vendor-specific format, fragmented across ad hoc logs, or missing context at async boundaries, you cannot answer those questions without manual reconstruction.

Use OpenTelemetry as the portability layer for Koog instrumentation. That choice does two things at once. It gives you a standard vocabulary for traces, resources, propagation, and export, and it keeps the execution evidence usable even if you change model providers, collec-

tors, or observability backends. Specialized AI observability platforms can still add value, especially for prompt-centric diagnostics and experiment review, but they work best as downstream consumers of a stable telemetry model rather than as the only place your run semantics exist.

A minimal Koog setup should export spans in a portable form before you add any backend-specific enrichment. Start with one local exporter for immediate inspection and one OTLP exporter for the real pipeline:

```
import ai.koog.agents.core.agent.AIAgent
import ai.koog.agents.features.opentelemetry.feature.OpenTelemetry
import io.opentelemetry.exporter.logging.LoggingSpanExporter
import io.opentelemetry.exporter.otlp.trace.OtlpGrpcSpanExporter

val agent = AIAgent(
    executor = simpleOpenAIExecutor(
        System.getenv("OPENAI_API_KEY")
    )
) {
    install(OpenTelemetry) {
        addSpanExporter(LoggingSpanExporter.create())
        addSpanExporter(
            OtlpGrpcSpanExporter.builder()
                .setEndpoint("http://localhost:4317")
                .build()
        )
    }
}
```

That example is small on purpose. It proves the architecture you want to keep: Koog emits trace data once, in a standard model, and exporters forward it to one or more destinations. The logging exporter gives you immediate local feedback during development. The OTLP exporter sends spans to a collector or backend that speaks the OpenTelemetry protocol. You can add backend-specific consumers later without rewriting the instrumentation model around them.

Why the portability layer matters

Agent systems change underneath you. You may swap model

providers, split one monolith into several services, move a tool out of process, or introduce a queue between planning and execution. If your telemetry semantics are tightly coupled to one backend's naming scheme or one SDK's event model, those changes become observability migrations as well as application migrations.

A portable instrumentation model separates four concerns:

- application code emits spans, attributes, and events,
- context propagation preserves causal identity across boundaries,
- exporters and collectors transport telemetry,
- backends store, query, visualize, and alert on that telemetry.

Keep those layers distinct. The application should know what a workflow step or tool invocation means. It should not need to know whether the spans ultimately land in a self-hosted tracing system, a cloud observability stack, or a specialized AI platform. Once you preserve that separation, backend choice becomes an operational concern instead of an architectural dependency.

Koog's OpenTelemetry hook

Koog exposes OpenTelemetry as an installable feature on the agent. That feature is the correct place to connect tracing export rather than stitching ad hoc export logic into business code. The exporter hook accepts OpenTelemetry span exporters, including Java SDK exporters and compatible Kotlin exporters. That matters for interoperability: you can reuse the broader OpenTelemetry ecosystem instead of waiting for framework-specific transport adapters.

Koog also exposes built-in integration points for some specialized destinations. If you choose to send data to Langfuse or Weave, use those helpers as exporters attached to the same tracing model rather than

as a separate, competing instrumentation path. The key design rule is simple: one semantic source of truth, many outputs.

A practical baseline pipeline

For most production deployments, use a three-step path:

- Koog emits spans through its OpenTelemetry feature.
- An OTLP exporter sends those spans to an OpenTelemetry collector.
- The collector forwards data to one or more storage or analysis backends.

That collector layer is not optional plumbing. It is where you gain batching, retry, routing, filtering, and policy enforcement without hard-coding those concerns into the agent process. If you export directly from every process to a vendor endpoint, you make backend migration harder and lose a useful control point for production operations.

A compact collector configuration for local development or an early deployment often looks like this:

```
receivers:
  otlp:
    protocols:
      grpc:

processors:
  batch:

exporters:
  debug:

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
```

```
    exporters: [debug]
```

That configuration is intentionally minimal. It accepts OTLP over gRPC, batches spans, and prints them through the collector's debug exporter. You can replace the `debug` exporter with a vendor-specific or storage-specific exporter later, but the receiving side and the Koog instrumentation remain unchanged.

What to normalize before export

Portable transport alone is not enough. You also need normalized semantics. Without them, two backends can ingest the same spans and still produce different stories because the attribute model is inconsistent.

Normalize three categories of metadata early.

Resource attributes

Resource attributes identify the process or service emitting telemetry. Koog provides default resource attributes such as `service.name` and `service.version`. Preserve and set those consistently. They become the primary dimensions for separating environments, versions, and service roles in multi-service deployments. If your agent runtime invokes tools through another service or background worker, resource attributes let you distinguish where a span was emitted without changing the application-level span naming.

Operation names

Keep span names stable and implementation-neutral. Names such as `agent.run`, `workflow.step`, `model.generate`, and `tool.invoke` survive backend changes and code refactors. Do not encode specific prompt variants, customer IDs, or provider-specific details into the span name. Put those in attributes instead.

Operational attributes

Attach attributes that support filtering and aggregation across runs: environment, workflow name, tool name, model identifier, provider name, outcome class, retry count, checkpoint indicator, and error type when present. Avoid high-cardinality fields in the dimensions you expect to graph or alert on. If you need to preserve a request or tenant identifier for forensic lookup, keep it bounded and deliberate.

Propagation is where traces usually break

Teams often think of export as the hard part and propagation as an SDK detail. The opposite is closer to reality. Export problems are usually obvious because no data arrives. Propagation problems are subtler: data arrives, but the trace is fragmented into unrelated pieces.

Any boundary that changes execution context can sever causality. In Koog deployments, the usual suspects are coroutine context switches, thread pools, background job executors, remote tool services, message queues, and callback bridges into provider SDKs. If the trace context does not cross those edges, you end up with orphan spans that look valid in isolation but do not reconstruct the run.

Treat propagation as a contract. The component that starts a run must establish trace identity, and every downstream component that continues the work must preserve or deliberately link that identity. When a tool invocation crosses a process boundary, pass the trace context with the request just as you would pass authentication or idempotency metadata. If a workflow step enqueues work for later resumption, store enough correlation metadata to reconnect the resumed work. That design principle matters more than any single library call because it keeps the run coherent when the system architecture evolves.

JVM and multiplatform considerations

The easiest OpenTelemetry integrations are typically on the JVM be-

cause the Java SDK ecosystem is mature and many exporters assume JVM runtime facilities. In a Kotlin Multiplatform codebase, keep the semantic model in shared code and allow platform-specific export or transport details where necessary. That usually means the names of spans, the attribute keys, and the run correlation policy should remain common across targets, while exporter wiring, transport security, or collector connectivity may differ by platform.

Do not force identical mechanics where the runtime does not support them cleanly. Force identical semantics instead. A trace emitted from shared JVM code and one emitted from another target should still describe the same operational concepts. When a backend query asks for all `tool.invoke` spans that produced validation failures, the query should not need to know which platform emitted them.

General observability stacks versus specialized AI backends

You usually need both classes of tools, but they serve different jobs.

A general observability stack is strongest when you need correlation across the entire distributed system. It can connect agent traces to HTTP services, databases, caches, queues, host metrics, and service-level objectives. If your incident starts as "the assistant is slow," a general stack helps you discover that the real cause is downstream queue saturation, collector distress, or a dependency outage unrelated to the model provider.

A specialized AI observability backend is strongest when you need prompt-centric inspection, evaluation metadata, run comparison across prompt or model variants, or interface features designed specifically for agent and LLM workflows. Those systems are often more comfortable for inspecting prompt assembly, tool argument payloads, and qualitative output differences. They can accelerate experimentation and debugging, especially for teams iterating rapidly on prompt or model behavior.

The mistake is treating those roles as interchangeable. A specialized backend is not a substitute for system-wide observability. A general backend is not always the best interface for prompt inspection. Keep the general stack as the durable, portable source of operational truth. Add specialized destinations when they meaningfully improve developer workflow or model analysis.

Where specialized exporters fit

Koog supports backend-specific exporter helpers, including integrations for Langfuse and Weave. Use them as additional sinks attached to the same agent telemetry pipeline. That gives you one run model and multiple views of it.

For example, you might send all traces to an OpenTelemetry collector for retention, routing, and cross-service analysis, while mirroring a sampled subset of development or incident traces to a specialized platform for prompt review. That arrangement preserves portability. If you later change vendors, your core instrumentation and collector pipeline remain intact.

Backend-specific export is most useful under three conditions:

- you need prompt and output inspection features beyond a general stack,
- you need experiment or evaluation views tied to traces,
- you can enforce the same privacy and retention rules you apply elsewhere.

If none of those conditions hold, adding another backend usually creates more operational burden than value.

Dual export without dual semantics

A common anti-pattern is dual instrumentation: one set of spans goes to a general backend, another set of backend-specific events goes to an AI platform, and the two systems use different IDs, names, and timing boundaries. That creates two truths for the same run. Engineers then waste time reconciling timestamps and guessing which system reflects reality.

Avoid that. Emit once. Export many times. If a specialized backend needs extra metadata, add that metadata in a way that does not destroy the portability of the core trace. Keep the run ID, span hierarchy, and stable operation names consistent everywhere. If a backend cannot preserve those basics, it is a poor primary observability store.

Collector-first architecture

Once you have more than one service, environment, or backend, a collector-first topology becomes the operational default. It gives you a central place to apply four kinds of control.

First, *routing*. You can send production traces to your long-term storage backend and route selected development or sampled traces to a prompt-analysis system.

Second, *policy enforcement*. Redaction, attribute filtering, or retention selection can happen before data leaves your controlled environment.

Third, *reliability*. Batching and retry behavior in the collector reduce direct coupling between transient backend instability and the application process.

Fourth, *change management*. You can change exporters or add a new destination without redeploying the agent fleet.

Keep one constraint in mind: the collector should shape transport and policy, not invent semantics that the application failed to emit. If the

application does not mark workflow transitions or tool boundaries, the collector cannot recover that meaning later.

Backend selection criteria

Choose a backend mix by looking at decision signals rather than feature lists.

Operational correlation

If you need to correlate agent runs with broader service telemetry, incidents, and SLOs, a general observability platform is mandatory. This is non-negotiable for production systems where the agent is one component in a larger service graph.

Prompt and evaluation workflow

If your team spends substantial time comparing prompts, outputs, and evaluation artifacts, a specialized backend can reduce debugging friction. That benefit is real, but it is secondary to preserving standard telemetry semantics.

Governance and compliance

Prompt and output data often contains sensitive content. Choose destinations that support the retention, masking, isolation, and access controls your environment requires. A platform with excellent prompt UX but weak governance posture is a poor fit for regulated or high-sensitivity workloads.

Lock-in tolerance

A backend becomes risky when operational dashboards, IDs, alert logic, and trace interpretation depend on proprietary concepts you cannot reproduce elsewhere. The stronger your OpenTelemetry discipline, the lower this risk becomes.

Cost and cardinality behavior

Backend pricing often changes sharply with telemetry volume and high-cardinality dimensions. Agent traces are especially vulnerable because prompts, model names, tool arguments, and user-linked identifiers can explode cardinality and payload size. Evaluate not just storage cost, but also the query behavior and retention pressure those fields create.

A workable production pattern

A solid default for serious deployments looks like this:

- instrument Koog with `install(OpenTelemetry)`,
- export over OTLP to a collector,
- enforce resource naming and attribute normalization centrally,
- send traces to a general observability backend for durable operational analysis,
- optionally mirror selected traces to a specialized AI backend for prompt inspection and experiment review.

That pattern scales because each layer has a clear job. Koog emits the run semantics. The collector handles transport, routing, and policy. The backends provide interfaces optimized for different kinds of analysis.

Validation and troubleshooting

When you first wire the pipeline, validate it in increasing order of complexity.

Start with the logging exporter inside the process. Confirm that spans are being created at the boundaries you expect. If a run does not produce a coherent local trace, export will not fix it.

Next, send the same spans to a local collector with a debug exporter. Confirm that the collector receives them and that resource attributes such as `service.name` are present and stable.

Then add your real backend. Confirm that span hierarchy, timing, and key attributes survive the trip unchanged. Backend-specific remapping sometimes alters field names or drops large attributes. Detect that early.

A simple collector run for local verification might produce output like this:

```
Resource attributes:
 service.name: koog-agent
 service.version: 1.0.0

Span:
 Name: agent.run
Span:
 Name: workflow.step
Span:
 Name: model.generate
```

The exact formatting varies by collector version and exporter, but you should see the resource metadata and the expected operation names. If the spans arrive as flat, unrelated entries, investigate propagation before tuning exporters.

Anti-patterns that create long-term pain

Do not couple the application directly to one vendor's trace schema. That saves minutes early and costs months later.

Do not emit different identifiers to different backends. One run must remain one run everywhere.

Do not export raw prompt and response content by default just because a backend UI makes it convenient. Sensitive data and telemetry cost both accumulate silently.

Do not use logs as a fallback for missing spans. If a control boundary matters, instrument it as a span or event.

Do not let collector-side transformations become the only place semantics are defined. The application owns the meaning of the run.

Do not confuse "data visible in one dashboard" with "portable observability." If your run model cannot survive a backend replacement, the architecture is still brittle.

Designing for backend change

The cleanest test of your observability design is whether you could replace the storage and visualization layer without rewriting the agent. If the answer is yes, your instrumentation model is doing its job. If the answer is no, look for the coupling point: proprietary span names, backend-only IDs, prompt-specific metadata with no standard representation, or business logic that depends on a vendor SDK call.

Portable semantics first, backend convenience second. That ordering keeps Koog telemetry useful across providers, services, and deployment targets, and it preserves the one thing operations cannot recover after the fact: a coherent, standardized account of what the agent actually did.

7.3. Classifying Failure Modes Across Providers, Tools, and Workflows

A production incident rarely announces its true category. You see a timeout, but the cause may be a provider rate limit, a malformed tool argument, a workflow loop, or a resumed run operating on stale state. You see a successful HTTP 200 from a downstream service, but the agent still returns the wrong answer because it called the wrong tool with syntactically valid inputs. You see a trace with a gap in the middle

and spend an hour debugging business logic when the real failure was lost propagation across an async boundary. If you treat all of those cases as generic "errors," you will patch symptoms, overuse retries, and miss the containment action that preserves correctness.

Use a fault taxonomy that classifies failures by *behavioral type* and by *execution domain*. Behavioral type tells you whether a problem is likely transient, persistent, semantic, state-related, or side-effect ambiguous. Execution domain tells you where it manifests: provider, tool, workflow control, or observability path. You need both axes. A timeout from a provider may be transient. A timeout caused by an unreachable workflow branch is not. A tool call may complete at the transport level and still fail semantically because the arguments expressed the wrong intent. The category drives both diagnosis and containment.

A useful starting point is to encode failure classes directly in the telemetry you already emit. For example, when a tool call fails argument validation or a workflow step hits a missing-state precondition, record that as a structured error type rather than a free-form log line. The exact code that sets these attributes depends on your tracing layer, but the shape should stay stable. A simple event payload model for failure classification looks like this:

```
{
 "error.type": "tool.schema_mismatch",
 "error.domain": "tool",
 "error.class": "structural",
 "tool.name": "create_invoice",
 "workflow.node": "issue_billing_action",
 "retry.eligible": false
}
```

That example is deliberately plain. The point is not the container format. The point is that an operator or query engine should be able to distinguish a structural tool-schema failure from a transient upstream timeout without reading paragraphs of logs. Once you can classify failure state in structured telemetry, the incident response path gets much

shorter.

Two axes of classification

Start with behavioral type.

Transient faults

Transient faults are temporary failures in communication or availability: network interruptions, backend timeouts, short-lived rate limits, or brief provider unavailability. These may disappear if you retry later or route around a localized outage. Their defining property is not that they are harmless. It is that the underlying operation could plausibly succeed unchanged once the transient condition clears.

Persistent or deterministic faults

Persistent faults do not improve with repetition. Unsupported provider features, invalid tool schemas, deterministic parser failures, and missing required workflow inputs all belong here. If the same request against the same code and same state will fail the same way again, do not treat it as transient simply because it happened over the network.

Semantic faults

Semantic faults are the most dangerous because the system may appear healthy at the transport level. The request succeeded, the tool returned a result, and no exception was thrown, but the operation violated intent or business meaning. The agent selected the wrong tool, updated the wrong record, summarized stale data, or entered a workflow branch whose preconditions were not semantically valid. These faults evade simplistic success-rate metrics.

State-related faults

State-related faults arise from missing, stale, contradictory, or cor-

rupted workflow state. The model may behave inconsistently because the run resumed with an incomplete checkpoint, because retrieved context and durable state disagree, or because a branch decision depended on state that was never made durable. State faults often masquerade as model unreliability because the visible failure occurs at a later reasoning step.

Ambiguous-side-effect faults

Ambiguous-side-effect faults occur when an external action may have partially succeeded but you cannot prove it. A tool request to create a ticket times out after the downstream system commits the write but before your agent receives the acknowledgment. If you replay blindly, you may create a duplicate ticket. If you avoid replay, you may silently lose the intended action. These cases demand idempotency keys, read-after-write verification, or compensating logic.

Now add execution domain.

Provider domain

Failures tied to the model provider, embedding service, or remote model API: rate limits, transport errors, unavailable models, latency spikes, malformed structured output, or context-window overflow. Provider faults are common, but they are not the only explanation for model-facing errors.

Tool domain

Failures in tool invocation, argument validation, permission checks, downstream API semantics, stale data, and non-idempotent side effects. Tool faults are often misdiagnosed as model failures because the model produced the request that triggered them.

Workflow domain

Failures in control flow: invalid branch conditions, unreachable states,

retry loops, unmet preconditions, planner divergence, compensation gaps, and node transitions that are locally valid but globally inconsistent. Workflow faults usually surface as timeouts, repetition, or inexplicable branch choices.

Observability domain

A trace can fail even when business logic succeeds. Lost propagation, missing span links, dropped exporter data, or inconsistent run identifiers create partial evidence. Trace fragmentation is not just a telemetry nuisance. It is a failure mode because it produces false narratives during incident response.

Provider-side failures

Provider faults are the first place many teams look, often because the provider is the least controllable dependency. That instinct is partly right, but you need finer distinctions.

Rate limits and temporary throttling

These are classic transient failures. Signals include bursts of provider errors, increased latency before failure, and multiple affected runs across otherwise healthy workflows. A trace often shows the same operation succeeding for some requests and failing for others under higher concurrency. Containment usually means backoff, concurrency reduction, or graceful degradation rather than prompt surgery.

Transport and timeout failures

A network timeout between your process and the provider, or between the provider client and its endpoint, is also usually transient. The trace signature is a model span that consumes wall-clock time without a successful response event. If other provider requests in the same period show similar symptoms, treat this as environmental or upstream instability.

Capability mismatch

A provider may not support the mode you requested: structured output shape, specific tool semantics, a model family, or a request option your client tried to use. This is not transient. If the feature is unsupported, repetition only amplifies load and hides the real defect. The trace signature is strong repeatability: the same request shape fails immediately with minimal variance. Containment is fail-fast, feature negotiation, or provider routing at a policy layer.

Output conformance failures

The provider returns a syntactically valid response that fails your downstream parser, schema validator, or tool-call contract. Depending on context, this can be transient or structural. If the schema prompt is unstable and the model usually conforms, a bounded retry may be justified. If the output contract is too strict for the model or the prompt is systematically underspecified, the fault is structural. Diagnosis depends on comparing failure frequency across the same prompt and model pair.

Tool-layer failures

Tool failures occupy the boundary where generated intent meets executable operation. They deserve their own taxonomy because they mix validation concerns, business semantics, and external side effects.

Schema mismatch and invalid arguments

When the model produces arguments that do not satisfy the tool contract, the dominant failure is structural. The system already knows the schema it expected. A retry that regenerates the same shape with the same prompt context often adds latency without increasing correctness. The trace should show the tool name, the validation result, and the exact argument fields or categories that failed, without dumping sensitive values unnecessarily.

Permission and policy failures

A tool call may be well-formed but unauthorized, disallowed by policy, or blocked by missing credentials. This is usually persistent until configuration or authorization changes. Operators often misread these as transient because they arrive as remote API errors. The distinguishing evidence is stability: the same principal, tool, and action fail repeatably regardless of timing.

Stale or contradictory downstream data

Some tools succeed technically while returning outdated or inconsistent data. This creates a semantic failure rather than a transport failure. The downstream service is up, but the run still makes a wrong decision because the input domain is stale. The trace signature often includes normal latency and no explicit error. You need domain checks or comparison events to detect that the tool result violated freshness or invariant expectations.

Ambiguous external effects

A payment initiation, ticket creation, email send, or record update may have committed even though your tool span ended in timeout or connection loss. That places the incident in the ambiguous-side-effect class. The correct response is not "retry because timeouts are transient." The correct response is to verify whether the effect already exists, typically through an idempotency key or read-after-write lookup, before replaying anything.

Workflow-control failures

Workflow failures often produce the most confusing symptoms because the local operations may all succeed. The model returns text. The tool returns JSON. The database stays up. Yet the run goes nowhere useful.

Unreachable branches and unmet preconditions

A workflow node may assume state that was never established. The branch condition fires because of incomplete validation, and the run enters a path that cannot complete. The trace pattern is usually a sequence of locally valid spans followed by a downstream failure that appears unrelated. To diagnose it, inspect the state and decision attributes on the workflow transition span rather than the final failing call.

Control loops and runaway retries

A planner or branch node may repeatedly produce the same next step without making progress. This is distinct from a simple provider retry. The observable signal is recurring workflow-step spans with similar attributes and stable or growing run duration. If you only monitor end-to-end success rate, these loops look like generic slowness. If you inspect span repetition per run, the control path becomes obvious.

Compensation gaps

A workflow may correctly detect a failed action but lack the logic to compensate or quarantine state. The result is a technically terminated run that leaves the system in an invalid business state. This is a workflow fault with state implications. Look for traces where an error span is followed by an apparently successful termination without any compensating step or operator escalation.

Planner divergence

In multi-step agent workflows, the planner can drift from the intended task while remaining internally coherent. The run keeps making moves, but each move moves farther from the user goal. This is a semantic workflow failure. You detect it through branch choice frequency, repeated low-value tool calls, and mismatch between the declared workflow objective and the actual operations chosen.

State-related failures

State faults deserve separate treatment because they corrupt interpretation itself. A system operating on the wrong state may still look syntactically healthy.

Missing required state

A resumed or branched run may lack data that the next node assumes is present. If the node checks preconditions, you get a clear structural failure. If it does not, the missing state becomes latent and reappears as a later semantic defect. Instrument state dependencies so you can see which node expected which durable fields or retrieval artifacts.

Stale checkpoints

A checkpoint may deserialize successfully while representing obsolete business facts. For example, a workflow resumes with a cached approval status that was revoked externally. This is not corruption in the binary sense; it is semantic staleness. The trace should mark the resume event, the checkpoint age or version if available, and any validation step that compares persisted state to live sources.

Corrupted or contradictory state

Different parts of the system may disagree about the same run: a memory store says one thing, a checkpoint another, and a downstream record a third. The agent then behaves "randomly" because different nodes consult different truth sources. The strongest evidence is contradiction across spans that reference the same logical entity. Quarantine the run rather than trying to heal it with more model calls.

Lost trust in state

Sometimes you cannot prove correctness after partial failure. A tool may have mutated one system, the checkpoint write may have failed, and the process may have crashed before recording the outcome. At

that point the dominant fault is not transport but trust. Automatic continuation is dangerous because the system no longer knows what is true.

Semantic failures across domains

Semantic failure cuts across providers, tools, and workflows. It deserves special emphasis because it is the class most likely to slip past technical availability metrics.

A provider response that parses cleanly can still be wrong for the task. A tool call that returns 200 can still update the wrong record. A workflow branch that matches its declared condition can still violate business policy. The common property is mismatch between observed behavior and intended effect. Diagnose these failures by aligning traces with domain invariants. Ask which operation the system claimed to perform, which object it actually touched, and whether the result advanced or violated the business objective. Traces help because they preserve the sequence of decisions and actions. They do not remove the need for domain validation.

Trace fragmentation as a failure class

Treat trace fragmentation as a first-class incident category. If propagation is lost across a queue, coroutine boundary, or remote tool service, your trace may show a clean provider call and a separate clean tool call with no link between them. An operator may infer the provider chose not to call the tool when the real issue is missing context propagation. This class is especially costly because it poisons diagnosis.

Look for orphan spans, sudden shifts in run identifiers, missing parent-child relationships, or repeated isolated spans that should belong to larger traces. When trace fragmentation occurs, fix the observability path before drawing strong conclusions about business logic.

Mapping class to containment

Classification matters because each class narrows the safe response.

Transient provider and transport faults often justify bounded retry, routing, or temporary degradation.

Structural faults usually justify fail-fast behavior, schema correction, or configuration repair.

Semantic faults justify validation, escalation, tighter contracts, or workflow redesign rather than transport-level mitigation.

State-related faults justify quarantine, checkpoint verification, or operator intervention.

Ambiguous-side-effect faults justify idempotency checks, read-after-write confirmation, or compensating actions before any replay.

If you collapse all five into one bucket, the system tends to over-retry and under-diagnose.

A simple incident-reading procedure

When a Koog run fails, classify it in this order:

- Determine whether the visible failure is transport, structural, semantic, state-related, or ambiguous-side-effect.
- Identify the execution domain: provider, tool, workflow, or observability path.
- Check whether the same request shape would fail again unchanged.
- Check whether any external side effect may already have occurred.
- Decide whether containment should preserve availability, correctness, or forensic evidence first.

That sequence prevents a common mistake: jumping to retry before deciding what kind of failure actually happened.

Anti-patterns

Do not treat every exception as retryable. Deterministic schema failures and missing workflow state often become slower, noisier incidents when retried.

Do not hide tool faults behind generic fallback text. A user-facing message may need to soften the error, but the telemetry must preserve the precise domain and class.

Do not assume success from HTTP status alone. Many semantic and state failures produce technically successful remote calls.

Do not resume from corrupted or untrusted state simply because a checkpoint exists. Persistence does not imply correctness.

Do not ignore missing traces or broken span links during incident review. When the evidence path is fractured, your root-cause analysis is already on unstable ground.

What a mature taxonomy gives you

Once the taxonomy is stable, incident handling changes. Dashboards can separate provider throttling from tool-schema defects. Alerts can trigger on repeated workflow loops instead of generic latency. Operators can distinguish a failed write from an unverified write. Engineering discussions become sharper because teams stop arguing over symptoms and start naming fault classes with specific evidence.

That is when the telemetry starts serving operations instead of merely recording activity. A run that fails inside Koog is no longer just “broken.” It becomes classifiable, queryable, and containable in a way that protects both system correctness and operator time.

7.4. Designing Retries, Recovery Boundaries, and Durable Resumption

A model call times out after requesting a tool action that may already have created a ticket in an external system. If you retry immediately, you may create a duplicate ticket. If you fail the run immediately, you may strand a user request that could have completed safely. If you resume later from a checkpoint that was written before the tool call, you may replay an operation whose side effects are no longer unknown but merely unobserved. That is the operational pressure point where fault classification has to become recovery policy.

The central discipline is to separate *retry* from *recovery*. Retries are one narrow mechanism for transient faults. Recovery is the broader decision process that determines whether execution may continue, where it may continue from, what evidence is required first, and which parts of prior work remain trustworthy. If you collapse those ideas into one generic “try again” button, the system becomes less reliable under stress, not more.

A practical recovery policy should be visible in traces and encoded around bounded units of work. The details of persistent state storage depend on your workflow engine and infrastructure, but the operational shape is stable. You need a clear record of the failed operation, whether the operation was side-effect free or idempotent, whether any durable checkpoint predates or postdates the action, and whether continuation is safe. A compact policy model looks like this:

```
{
 "operation.name": "tool.invoke",
 "tool.name": "create_ticket",
 "failure.class": "ambiguous_side_effect",
 "retry.eligible": false,
 "checkpoint.resume_safe": false,
 "recovery.action": "verify_or_compensate"
}
```

That payload is intentionally generic. The point is not a specific API. The point is that recovery decisions should be structured, auditable, and aligned to the semantics of the operation rather than hidden inside scattered exception handlers.

Retries are not a general reliability strategy

Retries feel safe because they are simple to add and often work in development. A transient timeout disappears on the second attempt, so the mechanism looks validated. Under production load, the same habit can amplify exactly the condition you are trying to survive. If an upstream provider is overloaded and every failing request is retried immediately or on the same schedule, your clients add more traffic to a degraded dependency. That can turn a local slowdown into a cascading incident.

Use retries only for failures you have reason to believe are transient: transport interruptions, temporary provider unavailability, or rate-limit conditions that are likely to clear. Even then, bound them. A retry policy needs a budget, a deadline, backoff, and jitter. The budget caps how much extra work a single user operation may trigger. The deadline prevents the run from silently stretching beyond its service objective. Backoff reduces pressure on the failing dependency. Jitter prevents synchronized clients from hammering the same recovery window.

A retry that ignores those constraints is not resilience. It is traffic amplification.

Observability retries are different from business retries

Do not confuse telemetry durability with business-operation safety. Exporter queues, collector retries, and OTLP backoff help preserve traces when an observability backend is slow or temporarily unavailable. Those mechanisms are useful, but they protect telemetry trans-

port. They do not justify replaying a tool call, a provider request with side effects, or a workflow node that may have already committed state elsewhere.

This distinction matters because engineers often internalize one healthy pattern and misapply it to another domain. It is good that the collector retries span delivery. That tells you nothing about whether the application should retry a payment request, replay a notification send, or rerun a branch that mutated workflow state.

Choose the retry scope before the retry policy

A recovery design starts by choosing the unit of retry. There are at least four useful scopes.

Provider-request retry

Retry only the model or remote provider request. This is the narrowest scope and the least dangerous when the request is pure and side-effect free. It is appropriate for timeouts, transport errors, or brief rate limits where reissuing the request does not duplicate an external action.

Tool-invocation retry

Retry a tool call only if the tool operation is known to be idempotent or provably side-effect free. A read-only search call can usually be retried. A “create” or “update” call often cannot be retried safely without an idempotency key or verification step.

Workflow-node retry

Retry the entire node when the node itself is a durable, idempotent unit of work. This is often the right abstraction for graph workflows because the node can own its preconditions, side-effect contract, and checkpoint boundary. Retriable behavior belongs to the workflow semantics, not to buried helper functions that the operator cannot see.

Run-level resumption

Resume the run from a persisted checkpoint or trusted boundary after a crash, timeout, or operator intervention. This is broader than retry. You are not necessarily repeating the failed call. You are continuing the run from the last state that remains trustworthy.

If you choose the wrong scope, a reasonable retry algorithm still produces bad outcomes. Retrying a pure provider request at the call level is often safe. Retrying an entire workflow node that contains a non-idempotent tool call may not be.

Recovery boundaries must match durable semantics

A recovery boundary is the place where you can stop, inspect state, and continue without ambiguity about what has already happened. Those boundaries should align to durable and idempotent units of work, not arbitrary internal function calls.

A poor boundary is "inside the middle of a helper that formats a prompt, calls a tool, and writes two records." That unit mixes pure computation, external side effects, and persistence. If it fails halfway through, replay semantics are unclear.

A better boundary isolates phases with different trust properties:

- gather and validate inputs,
- make a side-effect-free provider or reasoning call,
- execute one externally visible action with an idempotency strategy,
- persist resulting workflow state,
- advance to the next workflow node.

When those phases are explicit, you can reason about replay. Inputs can often be recomputed. Pure model calls can often be retried with a budget. External actions require idempotency or verification. Persisted state establishes whether resumption can continue from after the action or must stop for operator review.

Idempotency is the first recovery primitive

If a unit of work can be applied multiple times with the same effective result, recovery becomes simpler. That is what idempotency buys you. For tool-backed operations, design for idempotency whenever the downstream system allows it. Pass a stable operation identifier to the external system, or maintain your own mapping between a workflow action and the external effect it created.

Even when the external system lacks first-class idempotency support, you can often approximate it with a read-before-write or read-after-write verification step. For example, before replaying a "create_ticket" action, check whether a ticket with the same workflow correlation ID already exists. If yes, bind the run to the existing ticket instead of creating another one.

Without idempotency, ambiguous-side-effect faults force conservative handling. The safe answer becomes "stop and verify" rather than "retry and hope."

What bounded retry looks like operationally

A bounded retry policy has at least five decision elements.

Eligibility

Only retry faults classified as plausibly transient. A deterministic schema error, invalid prompt contract, missing required state, or unauthorized action is not eligible.

Budget

Cap the number of retries or the cumulative extra time a run may spend on retries. This protects both the dependency and your own latency objective.

Backoff

Increase the wait between attempts. Immediate retry is reasonable only for very short local races. For remote dependencies, backoff reduces synchronized load.

Jitter

Randomize the delay. If every agent instance retries after the same fixed delay, you turn failure recovery into a coordinated burst.

Deadline

Stop retrying when the run can no longer meet its user-visible or workflow-level deadline. A successful operation that arrives after the user abandoned the request may still create side effects you no longer want.

The exact numbers depend on your workload, but the shape does not. If a policy omits budget, backoff, jitter, or deadline, it is incomplete.

Provider retries versus tool retries

Provider retries are usually the safer class because model requests are often read-like operations. Even there, be careful with workflows that use provider responses to drive subsequent actions. If a retry changes the output enough to choose a different branch or tool path, you need trace evidence that the input or decision context changed across attempts.

Tool retries are more dangerous because the tool boundary is where side effects live. Use a stricter gate:

- Is the tool read-only or explicitly idempotent?
- Can you prove whether the previous attempt committed?
- Does the downstream system expose a stable operation identifier?
- Will replay violate business invariants even if the transport succeeds?

If any of those answers are unfavorable, do not hide the problem inside an automatic retry loop.

Workflow-level retries should be explicit

A workflow node may call several helpers and external systems before it either advances or fails. That makes the node the right place to express recovery policy, because it is the first boundary where the operator can reason about semantic completion. If a node is retryable, define why it is retryable: idempotent action, pure provider call, or verified compensation path. If it is not retryable, make that visible in telemetry and fail explicitly.

Avoid invisible retries buried inside prompt templates, callback chains, or utility wrappers. Hidden retries are hard to count, hard to budget, and easy to misdiagnose. They also distort traces by making a single workflow step appear slower without revealing how many recovery attempts happened inside it.

Durable resumption is broader than retry

A resumed run is not just a retried request after a pause. Durable resumption means you can restore enough trustworthy state to continue execution from a known-good boundary after process loss, machine restart, or deliberate operator intervention. That requires persisted state and replay-safe boundaries.

No framework-specific resume API is assumed here. The operational model is what matters. A durable resume point should answer three questions clearly:

- What state was made durable before interruption?
- Which external effects, if any, were already committed?
- What work remains eligible for replay without duplication or drift?

If you cannot answer those questions from persisted state and trace evidence, the boundary is not yet trustworthy enough for automatic resumption.

Checkpoint design for resumption

Checkpoint too frequently and you pay in storage, latency, and operational complexity. Checkpoint too rarely and you lengthen the replay window until ambiguity dominates. The right granularity follows semantic milestones, not instruction counts.

Good checkpoint candidates include:

- after validated input normalization,
- after a planner decision that determines the next durable branch,
- after a verified external side effect commits,
- after state reconciliation completes,
- before entering a long-running or asynchronous subworkflow.

Poor checkpoint candidates include arbitrary mid-function snapshots, unverified speculative model output, or pre-side-effect snapshots with

no record of whether the subsequent action actually committed. Those boundaries make resumption look possible while preserving the worst kind of uncertainty.

Trustworthy versus lossy resume points

Not all persisted state is equally useful. A trustworthy resume point contains enough information to continue without guessing. A lossy resume point forces you to reconstruct missing facts from logs, heuristics, or external system queries.

For example, persisting "workflow node = send_invoice" is usually not enough. You also need to know whether the invoice tool call was attempted, whether an idempotency key was used, whether a downstream confirmation was received, and what local state was updated afterward. Without that, the next process can only guess whether it should send again or skip the action.

When you design checkpoints, think like an operator reading an incident trace. What would you need to decide whether continuation is safe? Persist that.

Compensation is not the same as retry

Some failures should not be retried at all. They should be compensated. If a run created a downstream record and then failed before updating local workflow state, the correct continuation may be to record the external effect locally or to issue a compensating delete, not to replay the creation. Compensation requires business semantics. It is the recovery path for cases where the system did something real but ended in an inconsistent state.

That is why recovery design belongs at workflow boundaries. Compensation rarely makes sense at the level of a raw HTTP client wrapper or model executor. It makes sense at the level where you can interpret what the action meant.

A practical recovery procedure

When a run fails, apply a disciplined sequence.

- Classify the fault: transient, structural, semantic, state-related, or ambiguous-side-effect.
- Identify the smallest meaningful failed unit: provider request, tool action, workflow node, or whole run.
- Determine whether that unit is side-effect free, idempotent, or neither.
- Check for a trustworthy checkpoint after the last confirmed effect.
- If the effect is ambiguous, verify externally before replay.
- Retry only if the unit is eligible and the retry budget remains.
- Otherwise compensate, escalate, or stop with preserved evidence.

This procedure works because it forces you to answer safety questions before attempting liveness improvements.

Tracing the recovery path

Recovery decisions should be visible. Emit retry attempt counts, backoff intervals, idempotency keys or correlation identifiers where safe, checkpoint load and save events, and explicit reason codes for stop-versus-retry decisions. A failed run with clean recovery telemetry is often easier to operate than a superficially successful run whose continuation path is opaque.

The trace should let you answer: was this the first attempt or a resumed one, did the retry reuse the same inputs, did the tool action require

verification, and did resumption continue from durable state or from heuristic reconstruction? If those answers are absent, the system is asking operators to trust recovery logic they cannot audit.

Trade-offs you need to choose deliberately

Every recovery design balances at least five pressures.

Latency versus safety

Aggressive retry can improve short-term completion rates but worsen tail latency and overload. Conservative verification protects correctness but may require user waiting or asynchronous completion.

Storage overhead versus trust

More frequent checkpoints reduce replay ambiguity but increase write load, storage cost, and state management complexity.

Availability versus correctness

A system can appear highly available by retrying and replaying liberally while quietly duplicating side effects or corrupting state. For most externally visible actions, correctness wins.

Automation versus operator control

Automatic resumption is valuable for clearly transient or idempotent cases. It is dangerous when side effects are ambiguous or state trust is lost. Design for handoff points where an operator can inspect evidence.

Simplicity versus semantic precision

A single retry library is simpler to deploy than workflow-aware recovery logic. It is also less accurate because it cannot distinguish a safe pure call from a business-critical side effect. Complexity belongs where semantics demand it, not everywhere.

Anti-patterns

Do not put infinite retries around provider calls. They turn partial outages into queue growth and deadline violations.

Do not hide provider rerouting inside retries if the alternate provider changes behavior materially. A run that silently switches model semantics midstream may stop failing technically while still failing semantically.

Do not replay non-idempotent tool calls without verification. Timeouts are not proof of non-execution.

Do not checkpoint every trivial state mutation. That produces storage churn without improving trust.

Do not trust a checkpoint merely because it exists. Checkpoints can be stale, incomplete, or positioned before an ambiguous action.

Do not let telemetry pipeline retries justify business-operation retries. Those are separate safety domains.

What reliable continuation actually looks like

A resilient Koog workflow is not one that retries the most aggressively. It is one that can stop at a meaningful boundary, preserve what is known, identify what is uncertain, and continue only when replay is safe. That usually means bounded retries for truly transient faults, explicit idempotency for external actions, semantic checkpoint boundaries, and traces that expose every recovery decision in a form an operator can trust.

7.5. Engineering Latency, Cost, and Throughput Under Production SLOs

An agent can be technically correct and still fail its job. A support workflow returns the right answer, but users wait twenty seconds for the

first token. A document-processing pipeline finishes eventually, but queue depth doubles every afternoon because tool concurrency outruns a downstream service. A routing agent stays within error budgets, yet monthly spend climbs because prompt history and debug metadata quietly expand every request. These are not secondary optimizations to postpone until after reliability is solved. They are production constraints that shape whether the system is usable, affordable, and operationally sustainable.

Treat latency, cost, and throughput as first-class observable properties of a Koog system. You already have the execution structure from traces, the fault taxonomy to classify failure, and recovery rules to continue safely. Now you need to convert that visibility into performance decisions. The practical goal is not to minimize every number. It is to establish explicit service-level objectives, attribute resource consumption to concrete workflow stages, and make trade-offs deliberately rather than by accident.

A useful operating baseline is to compute stage timings and cost estimates at run completion, then expose them as structured telemetry. The exact storage path depends on your metrics and tracing stack, but the shape should be stable. A compact end-of-run record might look like this:

```
{
 "run.id": "r_92af",
 "latency.ms.total": 4820,
 "latency.ms.model": 2310,
 "latency.ms.tools": 1680,
 "latency.ms.checkpoint": 210,
 "tokens.input": 5230,
 "tokens.output": 690,
 "cost.usd.estimated": 0.084,
 "tool.calls": 3,
 "workflow.branch": "research_then_answer"
}
```

That record is not a replacement for traces. It is a summary view de-

rived from them. The trace tells you how the run unfolded. The summary tells you how the run consumed latency and money. You need both because production tuning depends on aggregated patterns, not just individual incident analysis.

Start with explicit SLOs

A system without explicit service objectives cannot optimize coherently. Teams then react to whichever graph moved last: average latency one week, token spend the next, throughput collapse after that. Set objectives that reflect the actual service promise. For interactive systems, end-to-end latency and first-token latency usually matter more than absolute completion throughput. For batch workflows, throughput, completion deadline, and cost per completed item may matter more than immediate response.

At minimum, define objectives for the following:

- end-to-end latency, usually at a percentile rather than an average,
- success rate adjusted for semantic correctness where possible,
- throughput under expected concurrency,
- cost per run or per business transaction,
- observability-pipeline health so you trust the measurements.

Percentiles matter because agent systems often have wide latency spread. A run that fans out to tools, waits on a provider, and persists checkpoints will not behave like a simple CRUD endpoint. Average latency hides exactly the slow tail that users notice.

Decompose latency by stage

The most common optimization mistake is treating the run as one opaque duration. That leads to low-quality actions such as changing models when the real bottleneck is a slow tool or shrinking prompt context when the real issue is queueing.

Break run latency into at least these components:

Ingress and request setup

Time spent normalizing input, loading state, assigning a run context, and preparing the workflow. This is usually small, but it becomes visible in systems with heavy authentication, large request payloads, or aggressive state hydration.

Prompt assembly and retrieval

Time spent gathering memory, retrieval results, examples, or tool schemas and assembling them into model input. This is where prompt inflation often begins. The model may not be slow; your prompt construction may be.

Provider round-trip

Time spent waiting on model inference or embedding generation. For streaming responses, distinguish time-to-first-token from total generation time. A run can feel responsive while still being expensive if first token is fast but long generation follows.

Tool wait time

Time spent in tool execution, including network calls, downstream queueing, and result normalization. This is often the dominant contributor in agentic systems that coordinate external services.

Workflow control overhead

Time spent in planner logic, branch evaluation, state transitions, and local serialization. This is easy to overlook because each step is small,

but repeated branching or looping can make control overhead significant.

Checkpoint and persistence time

Time spent writing durable state, loading resume data, or reconciling local and external state. This cost is operationally justified, but it is still latency you must account for.

Once you have this decomposition, optimization becomes diagnostic instead of speculative.

Read traces as performance evidence

Traces are more than debugging artifacts. They are the execution graph you need for latency attribution. A p95 run time spike becomes actionable only when you can locate which span families expanded. If `model.generate` spans remain stable but `tool.invoke` spans fan out or stretch, the provider is not your first target. If checkpoint spans suddenly grow after adding richer state capture, you have introduced durability overhead that may require batching or coarser boundaries.

Use traces to answer three questions for every slow path:

- Which span class dominates total duration?
- Is the delay compute-bound, queue-bound, or dependency-bound?
- Does the slow path correlate with a specific workflow branch, model choice, or concurrency level?

That third question is where many optimizations emerge. A run may be slow not because the system is globally slow, but because one branch activates a heavier model, invokes an extra tool, or carries a much larger prompt context. Branch- specific cost is often more informative than whole-system averages.

Model allocation is an optimization surface

Not every reasoning step deserves the same model. Classification, extraction, routing, or format repair can often tolerate a smaller or faster model than deep synthesis or high-risk decision support. The performance question is not "which model is best," but "which model is sufficient for this stage under the quality target and SLO."

Use traces and outcome metrics together. If a lightweight router model reduces stage latency substantially without increasing branch misclassification, that is a good exchange. If a smaller model saves cost but causes more invalid tool arguments, the apparent win disappears once retries, operator review, or user correction are counted.

A disciplined model-allocation strategy usually separates workflow roles:

- cheap and fast for routing or extraction,
- moderate for tool argument generation and constrained transforms,
- strong for long-context synthesis or high-value reasoning.

The exact mapping varies by workload, but the structure matters. A single expensive model used for every step is easy to configure and often operationally wasteful.

Token economics are a systems problem

Cost drift in agent systems rarely comes from a single dramatic mistake. It emerges from accumulation: longer chat history, extra retrieval passages, verbose tool schemas, larger system instructions, repeated self-correction loops, and debugging fields that were meant to be temporary. Because each change looks small, teams miss the compound effect until the invoice or quota alarm arrives.

Track at least three token dimensions:

- input tokens by workflow stage,
- output tokens by workflow stage,
- total tokens per successful business outcome.

The third metric is the most revealing. A system that uses more tokens but solves more cases correctly may still be efficient. A system that uses more tokens for the same outcome has token-spend drift.

Prompt growth deserves explicit control. Keep templates versioned, track retrieval-set size, and monitor history length per run. If a branch regularly crosses a prompt-size threshold, the resulting provider latency and cost jump are not surprising side effects. They are a direct result of input design.

Streaming changes perceived latency, not always total cost

Streaming improves responsiveness by delivering partial output earlier. For interactive systems, first-token latency may matter more than final completion time because users anchor on visible progress. That makes streaming a valuable experience optimization even when the underlying compute cost remains unchanged.

Do not overstate the benefit. Streaming can reduce perceived wait without reducing provider usage or backend cost. In some systems it adds complexity by requiring incremental rendering, cancellation handling, and trace events for partial delivery. Measure both first-token latency and total run duration. If streaming helps the user experience enough to justify that complexity, keep it. If your workload is batch-oriented or tool-dominated, streaming may deliver little operational value.

Concurrency can improve throughput or create collapse

Parallel tool calls and concurrent workflow execution often look like free speed. Sometimes they are. Independent read-only tool calls can collapse wall-clock time dramatically when downstream services have capacity. The same design can also overwhelm a rate-limited dependency, increase provider contention, or inflate queue depth until p95 latency worsens.

Measure concurrency effects where they matter:

- run-level throughput under sustained load,
- per-dependency concurrency and queue depth,
- p95 tool latency as concurrency rises,
- downstream error rate and timeout rate,
- cost of duplicate or abandoned work.

If throughput increases while tail latency and downstream errors stay stable, the concurrency change is healthy. If average latency improves but p95 tool timeouts spike, you have shifted pressure rather than removed it. Protect each dependency with explicit concurrency limits, especially tools that call non-elastic services.

Use throughput as a capacity metric, not a vanity metric

Throughput alone is easy to game. You can increase completed runs per minute by lowering quality, shrinking context too aggressively, or queuing more work than users find acceptable. Treat throughput as the rate of *useful completed work* under the latency and correctness objectives.

For interactive agents, throughput is often constrained by acceptable tail latency. For batch workloads, throughput may be the primary objective, but it still has to be normalized by cost and failure rate. If you

double throughput and triple downstream retries, the system did not truly improve.

Observability overhead is part of the budget

The telemetry pipeline itself consumes CPU, memory, network bandwidth, and backend budget. Rich traces help diagnosis, but they are not free. Capturing full prompt and response bodies increases payload volume, raises privacy risk, slows export, and increases storage cost. Even if the application remains healthy, an undersized collector can drop spans under load and blind you during the exact periods when visibility matters most.

Manage observability overhead explicitly.

Sampling strategy

Head sampling is cheaper because the decision happens early and fewer spans flow through the system, but it may miss rare slow or failing runs. Tail sampling retains better evidence for slow and erroneous traces, but it pushes memory and state requirements into the collector because decisions happen after more trace data has accumulated. Choose based on the question you need to answer. If rare failures are the main risk, tail sampling often pays for itself operationally. If traffic is extremely high and the failure rate is low, head sampling may be the only affordable baseline.

Batching

Batching exporters or collectors reduces overhead and improves throughput of the telemetry pipeline. It also increases flush latency and can make near-real- time debugging less responsive. For production storage backends, batching is usually worth it. For incident debugging or local development, you may want less aggressive batching so traces appear quickly.

Collector sizing and memory protection

Collectors are part of the production path. If they are under-provisioned, telemetry becomes lossy even when the application is healthy. Monitor collector queue growth, memory pressure, and dropped-span indicators just as you monitor application latency. A healthy agent with an unhealthy collector is operationally opaque.

Cost is not just model spend

When teams say "AI cost," they often mean provider invoice cost. That is only part of the picture. A production Koog system pays for provider inference, retrieval and storage infrastructure, tool-side API usage, checkpoint writes, collector resources, observability storage, and engineering time spent operating the system.

This broader cost view matters because optimizations can shift spend rather than reduce it. A cheaper model that increases retries may save provider dollars but consume more tool invocations and operator review time. A richer trace payload may reduce debugging labor while increasing telemetry storage cost. Make these trades visible rather than assuming the cheapest model call is the cheapest system design.

Decision signals for optimization

When you consider a change, evaluate it against concrete signals.

Model downgrade or rerouting

Use when a stage is latency-sensitive and quality margin is wide. Reject it when semantic failures, invalid tool arguments, or operator escalations rise.

Parallel tool execution

Use when tools are independent and downstream services can absorb concurrency. Reject it when queue depth, timeout rate, or rate-limit

pressure increases.

Streaming output

Use when first-token latency strongly affects user satisfaction. Reject it when the workload is batch-oriented or when added complexity outweighs perceived responsiveness gains.

Richer prompt context

Use when it materially improves correctness or reduces follow-up actions. Reject it when token growth dominates cost without measurable quality gain.

More frequent checkpoints

Use when recovery ambiguity is expensive or side-effect boundaries are risky. Reject it when checkpoint overhead dominates latency without improving trust.

Anti-patterns

Do not optimize average latency alone. Users and operators feel tail behavior.

Do not scale concurrency without protecting tool backends. One overloaded dependency can erase the benefit of every local optimization.

Do not mix experimental and production traffic in the same performance baseline. Prompt experiments and evaluation runs distort cost and latency metrics.

Do not ignore prompt growth because the provider still accepts the request. Accepted input can still be operationally inefficient.

Do not assume observability data is free. Telemetry volume, collector memory, and backend retention all shape the true operating budget.

Do not treat cost reduction as a universal objective. A more expensive

stage can be justified if it reduces retries, support load, or business-critical errors.

A practical optimization loop

A disciplined tuning loop is short and evidence-driven.

- Define the target SLO or budget you are trying to improve.
- Use traces to identify which span families dominate the bad tail.
- Confirm whether the issue is provider latency, tool wait, queueing, prompt growth, or checkpoint overhead.
- Change one lever at a time: model allocation, prompt size, streaming, concurrency, or checkpoint frequency.
- Compare p50, p95, cost per useful run, and error-adjusted success rate before and after the change.
- Re-check collector health so observability overhead did not become the new bottleneck.

That loop keeps optimization grounded in operational evidence. It also protects you from the common failure mode where a local improvement moves pressure into a less visible part of the system.

Performance engineering as control, not cleanup

A mature Koog deployment makes latency, cost, throughput, and telemetry overhead visible enough that model mix, concurrency policy, prompt design, and checkpoint frequency become controllable parameters rather than recurring surprises. Once those parameters are observable at the level of workflow branches, tool families, and run outcomes, you can tune for responsiveness without guessing, contain spend without blinding the system, and scale throughput without quietly paying for it in correctness or operator pain.

Chapter 8

Embedding Koog into JVM and Kotlin Multiplatform Systems

By the time a team reaches this chapter, the hard problems are no longer "How do I call a model?" but "Where does the agent live, who owns its dependencies, how does state cross service boundaries, and what can truly be shared across platforms?" The chapter answers those questions by moving from backend embedding patterns, through boundary wiring for tools and state, into Kotlin Multiplatform design and finally an end-to-end production reference architecture that shows how the book's earlier concepts fit into one deployable whole.

8.1. Integrating Koog with Spring Boot and Ktor

A Koog agent that behaves well in a sample program can become unreliable the moment you place it inside a long-lived server. The failure mode is rarely the model call itself. The trouble starts at the host boundary: configuration arrives from multiple sources, requests carry identity and deadlines, secrets rotate, logs need correlation IDs, and cancellation must propagate through coroutines instead of leaving orphaned work behind. If you embed Koog as if it were a local utility object, the surrounding service quietly loses control over startup, shutdown, state ownership, and observability.

That is the central integration rule: the server framework owns process lifecycle, dependency graphs, configuration, and request context; Koog owns agent execution semantics inside those boundaries. Once you enforce that split, both Spring Boot and Ktor become straightforward hosting environments rather than sources of accidental complexity.

A minimal host shape

Treat the agent runtime as an application service dependency, not as a framework entrypoint and not as a global singleton stuffed with request data. A useful baseline looks like this:

```
class AgentExecutionService(
 private val agentFactory: AgentFactory,
 private val toolGateway: ToolGateway,
 private val telemetry: AgentTelemetry
) {
 suspend fun handle(request: AgentRequest): AgentResult {
  val agent = agentFactory.create()
  return telemetry.trace(request) {
   agent.run(request)
  }
 }
}
```

The names here are intentionally generic because Koog's higher-level

runtime types and integration helpers are version-sensitive around the 1.0.0 stable release. The architectural point is stable: inject long-lived infrastructure into a service, create an execution for each run, and pass request-scoped data at invocation time rather than burying it inside the object graph.

Dependency setup and version baseline

Koog reached its first stable release as version `1.0.0` on May 21, 2026. That release established stable and beta module streams, removed older deprecated APIs, finalized graph DSL node names, redesigned Java interop, and decoupled HTTP transport from Ktor. On the JVM, Koog requires JDK 17 or later, and existing projects are expected to use Kotlin 2.3.10 or later. The repository also exposes first-class framework integration modules, including `koog-spring-boot-starter` and `koog-ktor`. Ktor's own DI support is available through the documented `io.ktor:ktor-server-di-jvm` artifact. Spring's container continues to default beans to singleton scope, with prototype and request scopes available when you need different lifetime rules.

A realistic Gradle shape for the host service therefore starts with framework dependencies and Koog modules, while keeping provider-specific configuration externalized:

```
dependencies {
 implementation("io.ktor:ktor-server-core-jvm")
 implementation("io.ktor:ktor-server-netty-jvm")
 implementation("io.ktor:ktor-server-di-jvm")

 implementation("ai.koog:koog-ktor")
 implementation("ai.koog:koog-spring-boot-starter")
}
```

Use the exact artifact coordinates from the release you target in your build. The repository confirms the module names, but provider-specific starter variants and package-level APIs change faster than the core hosting pattern. When an exact builder method

or annotation is not confirmed in the published API reference, prefer constructor-wired composition in your own application code over framework magic.

What the host framework should own

You get a stable integration model by assigning ownership deliberately.

Application-lifetime concerns usually belong to the framework container:

- model client factories and provider configuration,
- HTTP transport clients or transport factories,
- retrieval clients and vector store adapters,
- tool registries and domain service adapters,
- persistence repositories and checkpoint stores,
- metrics, tracing, and structured logging exporters,
- secret resolution and profile-based configuration.

Execution-lifetime concerns should arrive at the boundary of each run:

- authenticated principal or service identity,
- tenant and authorization scope,
- correlation ID and trace context,
- request deadline and cancellation signal,
- locale, channel, and user-facing response constraints,
- conversation or workflow instance identifier.

That separation prevents the two most common production mistakes. The first is creating provider clients inside request handlers, which destroys connection reuse and makes configuration impossible to validate at startup. The second is constructing one mutable agent object and feeding every request through it, which turns prompt state, memory handles, or tool-side caches into shared mutable state.

Spring Boot: container-managed composition

Spring Boot is strongest when you let the container assemble infrastructure explicitly and keep request-specific concerns out of singleton beans. Constructor injection remains the safest default because the full dependency graph is visible at startup and easy to override in tests.

```
@Configuration
class AgentConfiguration {

 @Bean
 fun agentExecutionService(
  agentFactory: AgentFactory,
  toolGateway: ToolGateway,
  telemetry: AgentTelemetry
 ): AgentExecutionService {
  return AgentExecutionService(
   agentFactory = agentFactory,
   toolGateway = toolGateway,
   telemetry = telemetry
  )
 }

 @Bean
 fun agentFactory(
  modelBackend: ModelBackend,
  tools: ToolRegistry
 ): AgentFactory {
  return DefaultAgentFactory(modelBackend, tools)
 }
}
```

This arrangement matters more than it first appears. Spring singleton scope means one bean instance per container by default. That is appropriate for stateless configuration objects, client factories, repositories,

and telemetry exporters. It is not appropriate for mutable conversation state. Spring's own scope rules also warn about a subtle trap: if you inject a prototype bean into a singleton bean, that prototype is resolved once at singleton creation time unless you use a provider or lookup mechanism. Teams often mark an agent bean as prototype-scoped and then inject it into a singleton controller or service, assuming they get a fresh agent per request. They do not.

If you need a fresh execution object for each call, inject a factory or provider, not the stateful object itself. In Kotlin that often means your own `AgentFactory` interface; in Spring it can also mean a provider abstraction resolved at call time.

```
@RestController
class AgentController(
 private val service: AgentExecutionService
) {
 @PostMapping("/agent/execute")
 suspend fun execute(
  @RequestBody body: ExecuteRequest,
  request: HttpServletRequest
 ): ExecuteResponse {
  val agentRequest = AgentRequest(
   input = body.input,
   principal = request.userPrincipal?.name,
   correlationId = request.getHeader("X-Correlation-ID")
  )
  val result = service.handle(agentRequest)
  return ExecuteResponse
 }
}
```

The controller should extract HTTP-specific details, convert them into an application request model, and then stop. Do not let controllers assemble prompts, choose tools, open database transactions, or instantiate provider SDKs. Every one of those choices makes the HTTP layer harder to test and harder to reuse from scheduled jobs, message consumers, or internal service calls.

Conditional wiring and profiles

Real services often need different providers across environments: local development may use one model backend, staging may use a lower-cost hosted provider, and production may use a region-specific configuration. Spring profiles and conditional beans are a better fit for that variability than agent-internal `if` statements.

```
@Configuration
class ProviderConfiguration {

 @Bean
 @Profile("dev")
 fun devModelBackend(): ModelBackend {
  return StubModelBackend()
 }

 @Bean
 @Profile("prod")
 fun prodModelBackend(): ModelBackend {
  return RemoteModelBackend()
 }
}
```

Keep the agent code ignorant of where the backend came from. The host chooses a provider; the agent consumes an abstraction. That keeps production secrets, rate limits, and failover policy in the configuration layer where operators can reason about them.

Transactions and side effects

A mature Spring service rarely performs all work inside a single controller method. Domain state changes typically sit behind `@Transactional` service methods, and Spring allows event handlers to run in transaction phases. That matters for agent integration because tool calls and memory writes often look harmless until a business transaction rolls back. If you append memory, publish an outbound event, or trigger a downstream notification before the transaction commits, you can leave the agent's durable side effects out of sync with business truth.

A safer pattern is to let the transactional application service commit

first and then emit agent-adjacent effects after commit. For example, if the agent summarizes a newly created support case, create the case inside the transaction, commit it, and only then persist derived memory or emit enrichment events. The host application, not the agent runtime, should own that consistency boundary.

Request scope versus execution scope

Spring offers request-scoped beans in web applications, but use them sparingly. They are useful for thin context carriers, not as a hiding place for all execution state. Request scope ends with the HTTP request. Agent execution may not. Streaming responses, deferred processing, resumable workflows, and checkpointed continuation all outlive the inbound request. If you place conversation state inside a request-scoped bean, you have quietly tied the execution model to synchronous HTTP.

A better rule is simple: request-scoped beans may hold accessors for request metadata, but conversation history, run state, checkpoints, and long-term memory belong in explicit stores or execution objects that can survive beyond the request thread.

Ktor: code-first composition with explicit modules

Ktor gives you fewer container abstractions by default, which makes ownership boundaries more visible. That is an advantage when you want tight startup control. Ktor's DI plugin is now documented as a first-class option, with application-wide registration and typed resolution. The DI plugin also supports conflict-policy settings, optional resolution, and lifecycle-managed resources, which is enough for most agent-hosting needs.

A compact Ktor module can register dependencies and keep the route layer thin:

```
fun Application.module() {
 install(io.ktor.server.plugins.di.DI)
```

```
 dependencies {
  provide<ModelBackend> { RemoteModelBackend() }
  provide<ToolRegistry> { DefaultToolRegistry() }
  provide<AgentFactory> {
   DefaultAgentFactory(
    resolve<ModelBackend>(),
    resolve<ToolRegistry>()
   )
  }
  provide<AgentTelemetry> { DefaultAgentTelemetry() }
  provide<AgentExecutionService> {
   AgentExecutionService(
    agentFactory = resolve<AgentFactory>(),
    toolGateway = DefaultToolGateway(),
    telemetry = resolve<AgentTelemetry>()
   )
  }
 }

 routing {
  post("/agent/execute") {
   val body = call.receive<ExecuteRequest>()
   val service by dependencies.resolve<AgentExecutionService>()
   val request = AgentRequest(
    input = body.input,
    principal = call.principal<UserIdPrincipal>()?.name,
    correlationId = call.request.headers["X-Correlation-ID"]
   )
   val result = service.handle(request)
   call.respond(ExecuteResponse)
  }
 }
}
```

The exact registration DSL around `dependencies`, `provide`, and `resolve` should match the Ktor version you use, but the structural rule remains stable. Register long-lived dependencies at application startup. Resolve them in handlers or services. Keep route code responsible only for transport adaptation.

Ktor's lighter model makes one anti-pattern particularly tempting: defining stateful variables in the route module closure because they are easy to reach. Avoid that. A mutable object captured by a rout-

ing lambda is still effectively application-global state. It bypasses DI, is awkward to replace in tests, and tends to accumulate request data accidentally.

Coroutine context and cancellation

Both Spring and Ktor can host suspending request handlers on the JVM. That gives you a single execution model for Koog, provider calls, database drivers, and internal services, but only if you preserve coroutine context correctly.

Do not detach agent work into an unstructured background scope just to keep handlers simple. If you launch work in a scope that is not tied to the request or application lifecycle, cancellation and deadlines stop propagating. The client disconnects, the HTTP timeout expires, but the model call and tool chain continue running in the background. You then pay for tokens, hold connections open, and emit partial side effects long after the request is dead.

Use structured concurrency. Pass the current coroutine context through to the agent execution service, or wrap the execution in a bounded timeout owned by the application service. If you need work to continue after the request returns, make that explicit with a durable queue, scheduler, or checkpointed workflow handoff rather than an ad hoc `launch`.

```
suspend fun executeWithDeadline(
 service: AgentExecutionService,
 request: AgentRequest
): AgentResult = withTimeout(request.timeoutMillis) {
 service.handle(request)
}
```

This is where execution scope becomes more important than framework choice. Spring and Ktor differ in ergonomics, but both can enforce deadlines, cancellation, and trace propagation if your composition keeps those values at the boundary rather than embedding them

inside singleton components.

Application-service mediation versus direct controller calls

You can wire a controller or route directly to an agent factory and call it a day. That works for a demo. It is usually the wrong shape for a service. An application-service layer between transport and agent execution gives you four concrete benefits.

First, it centralizes policy checks such as authorization, tenant validation, and feature flags. Second, it keeps transport DTOs away from prompts and workflow state models. Third, it gives you one place to align timeouts, retries, and trace boundaries. Fourth, it lets you reuse the same execution logic from HTTP, batch jobs, and internal service triggers.

Direct controller-to-agent invocation is acceptable only when the endpoint is genuinely trivial, stateless, and unlikely to be reused. Even then, the shortcut tends to grow into an architectural seam you later regret.

Manual composition versus framework-managed graphs

Some teams avoid framework DI for startup determinism or reduced magic. That can be reasonable, especially in Ktor, where manual composition is often only a few lines longer. The trade-off is not ideology; it is operational clarity.

Framework-managed graphs help with profile selection, bean replacement in tests, lifecycle hooks, and centralized configuration binding. Manual composition helps when you want explicit startup order and no container-specific annotations in your core services. The wrong choice is mixing both without a rule. If half of the graph lives in Spring beans and the other half is built manually in static helpers, you lose the advantages of either model.

A practical compromise is to keep your agent-facing services free of framework annotations and let the framework only assemble them. That preserves portability while still benefiting from container lifecycle management.

Version-sensitive integration points

Koog 1.0.0 changed several integration assumptions that affect embedded services. HTTP transport is no longer treated as Ktor-coupled infrastructure, so older code samples that directly pass Ktor client objects into Koog constructors may no longer match current APIs. The 1.0 line also redesigned Java interop so blocking entrypoints follow an `xxxBlocking` naming pattern from Kotlin while presenting plain names to Java through renamed overloads. In a mixed Java/Kotlin server, be deliberate about which entrypoints you expose from your service layer. Keep suspending execution inside Kotlin where possible, and reserve blocking adapters for integration edges that truly require them.

Koog's stable and beta module streams also matter operationally. If a service is part of a production request path, prefer stable modules unless you are consciously accepting API churn. Beta modules can still be the right choice for experiments, but make that dependency visible in architecture review rather than discovering it during an upgrade.

Hidden instability patterns

Several integration mistakes are easy to miss because they work under light load.

Constructing provider clients inside handlers. You pay repeated initialization cost, lose connection pooling, and cannot fail fast on bad configuration.

Hiding I/O inside tool constructors. A tool should depend on already-configured services. If the constructor opens sockets or reads secrets,

tests become slow and unpredictable.

Passing HTTP DTOs directly into prompts. Transport models drift with the API surface. Domain request models change less often and protect the agent layer from controller churn.

Leaking framework classes into portable logic. If your core agent service depends on `ApplicationCall`, `HttpServletRequest`, or Spring annotations, you have made later multiplatform sharing much harder than it needs to be.

Treating conversation state as request-local. As soon as execution streams, retries, checkpoints, or resumes, request-local storage is the wrong lifetime.

Using a mutable singleton agent. Concurrency bugs surface slowly, often as cross-request prompt contamination or races around shared mutable state.

A durable request path

A server-hosted Koog execution should follow a repeatable flow regardless of framework:

- The HTTP entrypoint authenticates the caller and parses transport DTOs.
- The transport layer extracts correlation ID, tenant, deadlines, and user-visible response options.
- An application service converts those values into an execution request.
- The service obtains a fresh agent execution from a factory.
- The execution uses framework-managed infrastructure: model backends, tools, persistence, retrieval, and telemetry.

- Cancellation, timeout, and trace context propagate through the coroutine context.
- The service maps the result into a transport response and emits any post-commit side effects through application-owned boundaries.

That flow is intentionally conservative. It gives the host application clear control over lifecycle and side effects while keeping Koog responsible for orchestration. Once those boundaries are stable, adding memory, checkpointing, multiplatform sharing, or alternate deployment topologies becomes an extension of the same design rather than a rewrite forced by early shortcuts.

8.2. Wiring Memory, Tools, and Persistence Through Application Boundaries

A server-hosted agent becomes hard to govern when it reaches directly into databases, search indexes, caches, and outbound APIs as if they were just more prompt-time conveniences. That design often works in a prototype because everything lives in one process and one developer controls all the code. Under production load, the same shortcut turns authorization into an afterthought, makes tests depend on live infrastructure, and blurs the line between conversational context, durable workflow state, and business data. You avoid that collapse by treating memory, retrieval, tools, and checkpoint persistence as application-owned capabilities that are exposed to Koog through narrow, explicit seams.

The distinction matters because these concerns age differently. Tool implementations evolve with domain policy. Retrieval adapters evolve with indexing strategy and tenancy rules. Memory policies change

as context costs and relevance trade-offs shift. Checkpoint formats harden under operational recovery requirements. If you wire all of them directly into agent code, any one of those changes forces broad edits across the system. If you expose them as host-managed services behind stable boundaries, you can replace storage engines, split services, add audit trails, or change tenancy rules without rewriting orchestration logic.

A concrete composition pattern Start with a shape that keeps the boundaries visible. The host application owns infrastructure clients and domain services. The agent receives only the capabilities it should be allowed to exercise.

```
data class SupportAgentInput(
 val tenantId: String,
 val userId: String,
 val conversationId: String,
 val message: String
)

interface CaseLookupService {
 suspend fun findRecentCases(
 tenantId: String,
 userId: String,
 query: String
 ): List<CaseSummary>
}

interface PolicyScopedRetriever {
 suspend fun retrieve(
 tenantId: String,
 userId: String,
 query: String
 ): List<KnowledgeSnippet>
}

interface CheckpointStore {
 suspend fun load(runId: String): ByteArray?
 suspend fun save(runId: String, state: ByteArray)
}
```

These interfaces are intentionally small. They do not leak JDBC ses-

sions, ORM entities, HTTP client types, or framework classes. They encode the decisions the host application wants the agent to be able to make: look up cases within an authorized scope, retrieve snippets under policy constraints, and persist a resumable execution state. You can back those interfaces with databases, search services, queues, or in-memory test doubles without changing the orchestration code.

This pattern also keeps you honest about where authorization happens. If a tool receives a repository and tenant-unaware query methods, it will eventually bypass business rules. If it receives a service whose signature already requires tenant and user scope, the application has a place to enforce access rules before the agent ever sees the data.

Three layers of composition You get predictable behavior by keeping three layers distinct.

Infrastructure adapters speak to concrete systems: PostgreSQL, MySQL, H2, vector stores, object storage, message buses, and third-party HTTP APIs. They know about retryable errors, connection pools, serialization formats, and driver-specific exceptions.

Application services express business operations in terms the rest of the system understands: fetch active account context, record an approved action, search only documents visible to a tenant, or load the last successful checkpoint for a workflow instance. They own authorization, validation, transaction boundaries, and idempotency rules.

Agent-facing capabilities present the smallest useful surface to Koog: a retriever, a memory provider, a tool contract, or a checkpoint persistence abstraction. They should not expose the full richness of your internal domain model unless the agent truly needs it.

Most design mistakes happen when one of these layers disappears. Direct repository access from a tool collapses infrastructure and appli-

cation policy into the agent boundary. A prompt builder that queries a vector store directly collapses retrieval policy into orchestration. A checkpoint adapter that serializes a Spring request object collapses transport state into durable runtime state. The fix is not more interfaces for their own sake; it is keeping operational responsibilities separated enough that each part can change independently.

Memory is not just transcript storage Koog's memory features evolved significantly before the 1.0 stable line. Older names such as `AgentMemory` no longer describe the current model and should stay out of new designs. The more durable framing is that conversational and long-lived memory are policy surfaces, not implementation details. You decide what should be remembered, when storage failures are acceptable, and how retrieved context should be filtered before it reaches the model.

Koog's preview and changelog materials document chat-oriented memory support through a `ChatMemory` feature and a `ChatHistoryProvider` abstraction, with JDBC and ORM-oriented backends for stores such as PostgreSQL, MySQL, and H2. They also document the move toward `LongTermMemory` as the preferred concept and the addition of an explicit `FailurePolicy` after storage errors had previously been swallowed. That operational lesson is more important than the name itself: memory writes are side effects and must have explicit failure behavior.

If memory storage fails, you need to know whether the run should continue, fail, or degrade with a visible warning. Silent best-effort writes are almost always wrong in regulated or resumable systems because they produce an execution that appears successful while losing critical context for later runs.

Treat memory as a governed write path A robust memory write path answers four questions:

- What content is eligible for storage?
- Under what identity and tenant scope is it stored?
- At what point in the business flow is it durable?
- What happens if storage fails?

That means you should not let arbitrary tools append raw strings to long-term memory. Route candidate memory writes through an application service that can normalize content, attach metadata, redact sensitive fields, and decide whether the write belongs before or after the associated business transaction commits.

For example, imagine an agent that helps customer support staff resolve billing problems. The agent may summarize a successful resolution into long-term memory so later runs can retrieve a concise support history. That summary should be written only after the underlying billing correction commits successfully. If the transaction rolls back, the memory write must not survive as an alternate reality.

Persistence and memory are different concerns Teams often conflate three different stores:

- **business data**, such as orders, cases, invoices, and user records;
- **memory**, which stores interaction history or reusable knowledge;
- **checkpoints**, which store execution state for recovery or resumption.

They overlap operationally but not semantically. Business data is the system of record. Memory is a selective context source for future runs. Checkpoints are recovery artifacts for in-flight or suspended execution. If you use one storage model for all three, you either over-couple recovery to business schema or you under-specify business consistency.

Koog's checkpoint support through `AIAgentStorage` and `runFromCheckpoint` makes the recovery role explicit. The key architectural move is to persist agent runtime state separately from conversational memory. A resumed execution needs the state required to continue the workflow correctly, not a lossy reconstruction from transcript snippets. Conversely, long-term memory should not inherit opaque runtime internals that have meaning only to the executor.

A practical host-side flow A production request path usually follows an order like this:

1. Resolve tenant, principal, and policy scope from the incoming request.
2. Open or join a business transaction if the request performs a domain mutation.
3. Load domain records needed to decide what the run is allowed to do.
4. Narrow retrieval scope before any search or memory fetch occurs.
5. Load a checkpoint if the request resumes an interrupted run.
6. Execute the agent with injected capabilities, not raw repositories.
7. Commit business side effects under application service control.
8. Write post-commit memory entries and outbound events.

9. Persist the next checkpoint if the workflow may continue later.

That order prevents the most expensive kind of inconsistency: an agent side effect that escapes before the durable business action it was meant to accompany. It also gives you a reliable place to emit traces and audit records because each stage has a clear owner.

Design tools as policy-bound operations A tool should represent an allowed action, not an infrastructure shortcut. If the agent needs to approve a refund, do not give it a database handle and let it update tables. Expose a tool that calls an application service with the same validations, authorization checks, and audit behavior that a human-facing endpoint would use.

```
interface RefundApprovalService {
 suspend fun approveRefund(
 tenantId: String,
 actorId: String,
 caseId: String,
 amount: Long
 ): RefundApprovalResult
}
```

That service can still use repositories and transactional writes internally, but the agent boundary stays stable and testable. You can fake the service in tests, wrap it in additional audit logic, or reroute it to another subsystem later. More importantly, you keep domain invariants in one place instead of copying them into tool implementations.

When you evaluate a proposed tool, ask a blunt question: if a human used this capability through the standard application path, what validation and authorization would apply? The tool should go through that same path or a shared application service that enforces the same rules.

Direct infrastructure access versus mediated services There are cases where a tool can call infrastructure directly. Read-only

lookup against a fully policy-scoped search adapter can be acceptable, especially when the application service would add no business value beyond forwarding parameters. The criteria are strict:

- the operation is read-only,
- tenant and authorization scope are already embedded in the adapter,
- the returned data shape is stable and safe for the agent to consume,
- the access path is independently observable and testable.

Once the operation writes data, crosses service boundaries with side effects, or depends on nontrivial business rules, mediated application services are the safer default. The latency cost is usually negligible relative to model execution, while the governance benefit is substantial.

A hybrid design is common. Retrieval may go directly through a scoped adapter. Mutating tools go through application services. Memory writes pass through a memory policy service. Checkpoints use a dedicated persistence adapter. The architecture stays coherent because each path reflects a deliberate trust decision rather than a convenience shortcut.

Retrieval should be scoped before ranking Retrieval quality is often discussed in terms of embeddings, chunking, and ranking. In embedded systems, the more consequential problem is scope. If the retriever can see documents across tenants, business units, or security classes and relies on post-filtering later, you have already lost architectural control. Narrow the candidate set before ranking whenever possible.

That means the host application should inject a retriever that already knows the caller's tenant and policy context or receives those values explicitly on each invocation. Do not encode those rules in prompt text and hope the model ignores unauthorized results. A policy-scoped retriever can still perform semantic search, caching, and result fusion internally, but it must do so within application-defined visibility boundaries.

The same rule applies to memory retrieval. Memory is often more sensitive than static knowledge because it may contain user-generated text, summaries of prior decisions, or extracted operational context. Scope first, rank second, prompt third.

Checkpoint boundaries and serialization discipline Checkpointing is valuable only when the stored state is both durable and portable enough to survive process restarts, deployments, and partial failures. That requires disciplined serialization. Persist plain execution state, identifiers, and domain values. Do not serialize framework-managed objects such as request contexts, transaction handles, live HTTP clients, dependency injection containers, or coroutine jobs. Those objects are tied to a process lifetime and rarely deserialize meaningfully after a restart.

A good checkpoint payload typically contains:

- workflow or run identifier,
- current workflow node or execution marker,
- typed domain state needed for continuation,
- references to external resources by ID,
- version metadata for state evolution.

A bad checkpoint payload contains open resources, framework references, or opaque objects whose meaning depends on a specific classloader arrangement. If your checkpointing strategy requires those objects, the execution boundary is wrong. Reconstruct them from the host application when the run resumes.

Transaction ordering and post-commit effects For mutating workflows, use a strict ordering rule: business transaction first, durable commit second, post-commit memory or event side effects third. This is where agent systems often fail in subtle ways. A tool performs a domain write and immediately emits a memory summary or an outbound event. The transaction then rolls back, but the memory and event survive. Future runs now retrieve false history, and downstream systems observe actions that never committed.

You prevent that by making post-commit work explicit. In Spring, that often means using transaction-aware events or a service boundary that emits only after commit. In Ktor or manual composition, the same discipline applies even without annotation-driven transactions: structure the application service so that side effects which depend on durable business truth happen only after the commit path succeeds.

Memory ingestion, checkpoint persistence, and outbound notifications do not all belong in the same phase. Checkpoints for resumable execution may need to occur before a long external wait, while business-derived memory generally belongs after the relevant domain state is durable. The host application should own those choices. The agent should not infer them ad hoc.

Failure policies must be explicit Koog's move to add a failure policy for long-term memory is a useful model for the entire composition boundary. Every external capability should declare what happens on failure.

For each memory, retrieval, tool, and checkpoint path, choose among behaviors such as:

- **fail the run**: required for critical writes and strict consistency;
- **degrade visibly**: continue, but attach warnings and trace data;
- **retry within bounds**: appropriate for transient infrastructure faults;
- **suspend for resumption**: preferable when partial progress is valuable but not safely repeatable.

Do not leave that decision hidden inside adapters. The application service and runtime policy layer should declare it so tests can assert the intended behavior. A retrieval timeout may justify fallback to a smaller local cache. A checkpoint write failure in a resumable workflow may require immediate suspension. A memory write failure may be acceptable for ephemeral chat support but unacceptable for regulated investigation workflows.

Testability comes from narrow seams You know the composition boundary is working when tests stay small and deterministic. A unit test for orchestration should be able to inject a fake retriever, a fake memory adapter, and a fake checkpoint store without booting Spring, starting Ktor, or connecting to a database. An application-service test should be able to verify transaction ordering and post-commit behavior with infrastructure fakes or slice tests. A tool test should exercise the tool contract against a stubbed domain service, not through a full HTTP stack.

This is another reason to avoid letting agent code depend on repository entities or transport DTOs. Those dependencies make isolated tests impossible because the agent becomes entangled with persistence

mappings and controller concerns. The smaller the capability surface, the easier it is to simulate failure, reorder operations, and assert idempotency.

Multi-tenant and audit-sensitive systems Boundary design gets stricter in multi-tenant systems. Every memory lookup, retrieval query, tool invocation, and checkpoint load must carry tenant identity explicitly or derive it from a trusted execution context. Never rely on a globally configured tenant or a mutable thread-local alone, especially in coroutine-heavy servers. Pass tenant and actor identity as part of the operation contract or bind them into a request-scoped execution context that is converted into explicit parameters at the application boundary.

Auditability follows the same rule. Record who asked for a tool action, which run initiated it, what business object it targeted, and whether the action committed. If you only log provider prompts and responses, you miss the operational events that determine whether the system behaved correctly.

Common anti-patterns Several anti-patterns recur across otherwise competent implementations.

- **Repository-backed tools.** A tool calls repositories directly because it seems faster than writing an application service. Authorization and validation then diverge from the rest of the system.
- **Unbounded memory injection.** The host simply appends every message and summary into every future run. Token cost rises, relevance falls, and sensitive context leaks across tasks.
- **Checkpointing framework state.** Serialized request objects,

coroutine internals, or injected services make checkpoint recovery fragile and deployment-sensitive.

- **Retrieval mixed with business rules.** Ranking code starts making eligibility decisions that belong in application policy, which becomes hard to test and impossible to audit.
- **Hidden side effects in tools.** A tool appears to be a read operation but also mutates caches, emits events, or stores memory. That breaks reasoning about retries and idempotency.

A durable composition rule Keep infrastructure powerful and agent-facing capabilities narrow. Let the host application own policy, transactions, tenancy, and lifecycle. Expose memory, retrieval, tools, and checkpoint persistence to Koog as explicit services with clear failure behavior and scoped authority. Once those seams are in place, you can change storage backends, retrieval strategies, and recovery mechanics without letting infrastructure concerns leak into orchestration or letting the agent become an unreviewable source of side effects.

8.3. Designing Shared Agent Logic for Kotlin Multiplatform

Kotlin Multiplatform makes it easy to over-share. Once you have `commonMain`, it is tempting to move every agent-related class into the shared source set and treat the remaining platform modules as thin wrappers. That approach usually fails for the same reason server-side over-coupling fails: the closer code gets to transport, storage, credentials, native capabilities, and runtime lifecycle, the more platform-specific it becomes. What repays sharing is not the entire system but the parts whose meaning stays stable across targets: orchestration, state transitions, prompts, tool contracts, and policy.

A useful test is operational rather than philosophical. Ask whether a class still makes sense if you strip away the host environment. If the answer is yes, it belongs in shared code. If it needs a web framework, a mobile SDK, a browser API, a secure enclave, or a file path to function, keep it at the edge. Shared code should describe what the agent is trying to do, not how a specific runtime makes that possible.

A disciplined shared-core example

Start with a shared core that contains only domain models, orchestration policies, and capability interfaces. The code should compile without Spring, Ktor, Android, iOS-specific APIs, or browser globals.

```
import kotlinx.serialization.Serializable

@Serializable
data class AgentInput(
 val tenantId: String,
 val userId: String,
 val message: String
)

@Serializable
data class AgentOutput(
 val reply: String,
 val actions: List<String>
)

interface KnowledgeGateway {
 suspend fun search(
 tenantId: String,
 query: String
 ): List<String>
}

interface TicketTool {
 suspend fun createTicket(
 tenantId: String,
 userId: String,
 summary: String
 ): String
}

class SupportOrchestrator(
 private val knowledgeGateway: KnowledgeGateway,
```

```
 private val ticketTool: TicketTool
) {
 suspend fun prepareContext(input: AgentInput): List<String> {
 return knowledgeGateway.search(
 tenantId = input.tenantId,
 query = input.message
 )
 }
}
```

This code is intentionally narrow. It uses serializable data classes because state that crosses targets needs a stable representation. It defines capabilities as interfaces because the implementations differ sharply by platform. The orchestrator does not know whether knowledge comes from a server vector store, an on-device cache, or a browser-backed remote API. It only knows the contract.

That pattern scales well because it gives each target a clean job. Shared code owns policy and sequencing. Platform code owns the concrete binding to networking, persistence, secure storage, native SDKs, and telemetry.

What is worth sharing

The highest-value shared assets are the ones that encode meaning rather than mechanics.

- typed workflow and conversation state,
- prompt templates and prompt assembly rules,
- tool contracts and result types,
- validation rules and business policy abstractions,
- orchestration logic that sequences model calls and tool usage,
- serialization contracts for checkpoints and messages,
- error categories that the application can interpret consistently.

These elements benefit from one implementation because divergence across targets creates correctness problems rather than platform adaptation. If a workflow state machine behaves differently on Android and JVM for no product reason, you have created a debugging problem, not a feature.

Prompt policy is another strong candidate for sharing when the same business task exists on multiple platforms. You may still vary final wording or channel constraints at the edge, but the core intent, tool eligibility, safety constraints, and expected structured outputs usually belong in one place.

What should stay platform-specific

Platform modules should own anything tied to a runtime or operating environment.

- HTTP clients and provider SDK bindings,
- secret loading and secure credential storage,
- persistence drivers and local database integrations,
- file system access, background execution, and native capabilities,
- Spring beans, Ktor modules, Android components, and iOS bridges,
- observability exporters and runtime-specific logging hooks,
- transport entrypoints such as controllers, routes, or browser adapters.

This boundary is especially important for Koog embedding because the host environment determines lifecycle, dependency graph ownership, and failure behavior. A Spring bean definition has no place in shared orchestration. An Android background worker has no meaning on

WasmJS. A browser `fetch` wrapper is not a portable storage abstraction. When you keep those pieces in platform modules, you preserve the shared core's ability to compile, test, and evolve independently.

Prefer interfaces over large `expect/actual` surfaces

Kotlin's platform guidance is clear on a point that maps directly onto agent systems: use multiplatform libraries first, reserve `expect/actual` for small seams, and prefer common interfaces with platform implementations when the behavior is substantial or environment specific.

That recommendation protects you from a frequent Multiplatform failure mode. Teams start with a clean `expect/actual` declaration for a small function, then add more parameters, more side effects, and more target-specific branches until the shared declaration becomes a poor abstraction for all platforms involved. At that point the common code is no longer portable design; it is a negotiation artifact between incompatible runtimes.

A small seam is appropriate for values such as clock access, random IDs, or a minimal platform capability probe. Large integrations such as model provider access, checkpoint storage, vector retrieval, push notifications, or secure key management fit better behind interfaces.

```
interface CredentialProvider {
 suspend fun loadApiKey(name: String): String?
}

interface AgentCheckpointStore {
 suspend fun load(runId: String): ByteArray?
 suspend fun save(runId: String, payload: ByteArray)
}
```

On JVM, the credential provider may read from an external secret manager or environment-backed configuration. On Android, it may bind to platform secure storage. On iOS, it may use Keychain-backed logic. On JS or WasmJS, you may avoid local provider credentials entirely and

route model access through a backend. The interface remains stable; the platform risks do not leak inward.

Keep dependency injection consistent

If your project already uses a DI framework on a given platform, keep using it there. Kotlin's guidance explicitly warns against mixing styles casually once a DI framework is in play. In practice, that means you should not combine a framework-managed graph with an ad hoc lattice of `expect/actual` singletons and global service locators.

The clean pattern is simple. Shared code defines constructor-injected classes and capability interfaces. Each platform assembles implementations using its native composition style. Spring and Ktor can inject JVM implementations. Android can wire dependencies from its own application composition root. iOS can pass concrete implementations through a bridge layer. JS and WasmJS can compose browser-safe adapters or remote clients.

The shared core should not know whether a dependency came from Spring, Ktor, a manual factory, or an iOS entrypoint. Constructor injection is enough.

```
class SharedAgentRuntime(
 private val knowledgeGateway: KnowledgeGateway,
 private val ticketTool: TicketTool,
 private val checkpointStore: AgentCheckpointStore
)
```

That small decision keeps the core free of annotations, global lookups, and container assumptions. It also makes tests far easier to write because you can construct the runtime directly with fakes.

Organize source sets around ownership, not convenience

A healthy multiplatform project usually reflects architectural ownership in the module layout.

- `commonMain`: state models, orchestration, prompts, tool interfaces, serialization contracts, shared policies.
- `jvmMain`: server adapters, provider clients, JDBC or ORM-backed persistence, Spring or Ktor composition, observability exporters.
- `androidMain`: mobile transport adapters, local storage bindings, background work integration, secure secret access where appropriate.
- `iosMain`: platform bridges, native storage and key management, app lifecycle hooks.
- `jsMain` or `wasmJsMain`: browser-safe transport wrappers, sandbox-compatible storage, streaming adapters, local cache bindings if available.

Do not move code into `commonMain` merely because two platforms happen to use the same library today. The better question is whether the abstraction would still make sense if one of those platforms changed implementation later. If the answer is no, the code belongs in the platform layer.

Shared orchestration should remain pure enough to replay

Shared agent logic is strongest when it behaves like a deterministic orchestration layer around nondeterministic dependencies. You will never make model inference deterministic in every environment, but you can still make the surrounding control flow explicit and portable. That means the shared runtime should accept typed inputs, call capability interfaces, and produce typed outputs or state transitions.

This style improves more than portability. It makes checkpointing cleaner because the state you need to persist is explicit. It makes tracing cleaner because important execution steps have names in your own

domain rather than being hidden inside framework handlers. It also keeps retry and resumption behavior consistent across platforms because the rules live in one place.

A practical technique is to separate *policy state* from *resource handles*. Policy state belongs in shared code and can be serialized. Resource handles belong in platform code and must be recreated on resume. If a shared class needs a live HTTP client or a platform context object, portability is already eroding.

Do not leak server frameworks into common code

The most damaging portability bug is often small at first. A shared helper accepts a Spring request-scoped component because it is convenient. A workflow state class gains a Ktor `ApplicationCall` reference for logging. A tool contract returns a platform HTTP response type because rewrapping feels wasteful. Each decision looks local. Together they destroy the shared boundary.

Keep common code ignorant of framework classes, annotations, and runtime objects. Shared code should work equally well inside a JVM backend, an Android app, an iOS host, or a browser-oriented target that delegates provider calls to a backend service. If you need request metadata, define your own transport-free context model.

```
@Serializable
data class InvocationContext(
 val tenantId: String,
 val userId: String?,
 val locale: String?,
 val correlationId: String?
)
```

A Spring controller or Ktor route can populate this model from HTTP headers and auth state. An Android entrypoint can populate it from account state and app settings. A JS client can populate it from browser session context. The shared runtime only sees the normalized form.

Model access is often the first thing you should *not* share

Multiplatform code reuse does not imply that every target should talk to the model provider directly. In many systems, direct provider access belongs only on the backend because that is where you can protect secrets, centralize governance, and maintain consistent observability. The shared part, then, is not the provider binding but the orchestration contract.

That trade-off matters most on JS and WasmJS, where browser and sandbox constraints can sharply limit direct SDK use, credential storage, and network behavior. It also matters on mobile platforms, where embedding provider secrets inside the app often violates your own security model. In those cases, keep shared logic focused on request shaping, local policy, and result handling, while model invocation stays behind a backend facade.

A good litmus test is whether the target can safely hold long-lived provider credentials and emit complete operational telemetry. If not, avoid direct model bindings there even if a technical path exists.

Memory strategy should vary by platform

Shared orchestration does not mean identical memory behavior everywhere. A JVM service can rely on durable databases, vector stores, and checkpoint backends. A mobile client may have limited disk, background restrictions, and stricter privacy expectations. A browser target may need ephemeral storage or server-mediated retrieval. WasmJS may impose even tighter constraints on local persistence and available libraries.

What you should share is the *policy contract*: what kinds of memory the runtime can ask for, what metadata it expects back, and how memory influences the workflow. What you should not force into common code is a one-size-fits-all storage assumption.

For example, you may define a shared interface:

```
interface ConversationMemory {
 suspend fun recentMessages(
 conversationId: String,
 limit: Int
 ): List<String>
}
```

The JVM implementation may read from a durable store. Android may read from a local database synchronized opportunistically. JS may delegate to a backend session service. The orchestration rule stays shared, but the persistence model remains realistic for each target.

Serialization contracts deserve early attention

Portable state lives or dies on serialization. If workflow state, checkpoint payloads, or tool results are part of the shared core, define them as stable, explicit models and version them consciously. Do this early, not after the first failed migration.

Use serializable data classes for anything that must cross process or target boundaries. Avoid exposing raw exceptions, framework response objects, or platform-native collection wrappers as part of shared contracts. Those values serialize poorly, evolve unpredictably, or fail to round-trip consistently.

Versioning matters even more in multiplatform systems because release cadences differ. A backend may deploy daily. Mobile clients may lag for weeks. Browser assets may update quickly but not atomically with the backend. A shared state contract should therefore tolerate minor version skew and prefer additive change when possible.

Release coordination and debugging trade-offs

Multiplatform sharing reduces duplicated policy code, but it also introduces a coordination cost. A change to a shared orchestration model can affect JVM, Android, iOS, JS, and WasmJS simultaneously. That

is a strength when you need consistency. It is a liability when platform needs diverge.

Share code when the benefit is semantic alignment. Keep code local when the platform-specific behavior dominates the maintenance cost. Three signals suggest that sharing is still paying off:

- bug fixes in orchestration logic should land once and propagate,
- state and result models need to remain identical across targets,
- tests in common code catch defects before platform integration begins.

Three signals suggest you are over-sharing:

- common code is full of platform flags and target-specific branching,
- interfaces grow to accommodate one awkward platform after another,
- debugging requires understanding host framework behavior from inside the shared core.

When those symptoms appear, move behavior outward. Preserve the common policy, but let platforms own more of the execution plumbing.

A practical layering rule

You can keep the architecture stable with a simple layering rule.

1. Put prompts, workflow state, orchestration, validation, and tool contracts in shared code.

2. Put provider SDKs, HTTP clients, secure storage, persistence drivers, and framework composition in platform code.
3. Pass capabilities inward as interfaces and data outward as serializable models.
4. Use `expect/actual` only for small seams that are truly platform primitives.
5. Keep transport, credentials, and native side effects at the edge.

That rule preserves portability without pretending every target has the same execution environment. The reward is not just cleaner source sets. You get one place to reason about agent policy, one place to test workflow behavior, and clear boundaries for the parts of the system that must adapt to JVM servers, mobile devices, browser sandboxes, and emerging runtimes such as WasmJS.

8.4. Choosing Transport Boundaries and Deployment Topologies

An agent architecture starts to harden the moment you decide where execution lives and how callers reach it. That decision is not merely about HTTP endpoints or deployment convenience. It sets your latency floor, determines where secrets can safely reside, constrains how memory and checkpoints move, and decides whether observability can tell a coherent story across the system. Once a team has a portable core and clean application boundaries, the next pressure point is runtime placement: backend-only execution, shared-core clients with a backend agent, edge-hosted services, or a split model where some work happens locally and some remotely.

A common failure mode looks deceptively reasonable. A team begins with a server-hosted Koog agent behind a JSON API. Later, product

wants offline drafting on mobile, streaming in the browser, and lower latency for region-specific users. The first response is often to add more endpoints, cache more aggressively, and let clients call provider APIs directly where convenient. That usually produces a tangled system: some logic lives in mobile apps, some in controllers, some in server-side tools, and no one can state clearly which actions require the central agent service and which do not. Transport has become a hidden part of the business model.

The stronger approach is to treat transport as a replaceable boundary and topology as an explicit architectural choice. Koog 1.0 reinforces that direction. HTTP transport is no longer coupled to Ktor internals, and the release line moved toward a transport factory boundary instead of directly passing a Ktor `HttpClient` into LLM client constructors. That is more than API cleanup. It is an architectural signal: keep agent logic independent of the outer transport stack.

A stable invocation contract

Start by defining a transport-neutral invocation model. Controllers, routes, queue consumers, mobile bridges, and browser adapters should all translate into the same application-facing request shape.

```
import kotlinx.serialization.Serializable

@Serializable
data class AgentInvocation(
 val tenantId: String,
 val userId: String?,
 val sessionId: String,
 val message: String,
 val locale: String? = null,
 val stream: Boolean = false
)

@Serializable
data class AgentReply(
 val text: String,
 val runId: String,
 val requiresContinuation: Boolean
)
```

This model is deliberately ordinary. It carries identity, session scope, content, and a hint about response style, but it says nothing about HTTP headers, WebSocket sessions, Android lifecycle callbacks, or browser events. That separation is what keeps topology flexible. You can expose `AgentInvocation` through a Spring controller, a Ktor route, a message consumer, a scheduled worker, or a multiplatform bridge without rewriting orchestration or persistence rules.

On the server, the transport adapter should remain thin:

```
@RestController
class AgentController(
 private val service: AgentApplicationService
) {
 @PostMapping("/agent/invoke")
 suspend fun invoke(
 @RequestBody request: AgentInvocation
 ): AgentReply {
 return service.handle(request)
 }
}
```

The same principle applies in Ktor. The route parses the inbound payload, delegates to the application service, and maps the result back to the transport. Once you keep that boundary strict, the runtime placement question becomes independent of controller code.

Transport boundaries are control boundaries

A transport boundary is where the system translates external intent into internal semantics. It is the correct place to attach identity, correlation IDs, deadlines, streaming mode, rate limiting, and audit metadata. It is the wrong place to build prompts, decide tool policy, or perform direct provider calls. Those belong behind the application service boundary so the same behavior can be reused from non-HTTP contexts.

That rule becomes more important as you expand beyond backend HTTP. Suppose a mobile client needs local preprocessing for privacy-sensitive redaction. Suppose the browser needs token streaming while background jobs need resumable batch execution. If the transport layer owns prompt assembly or workflow branching, every new channel duplicates policy. When the transport layer only adapts requests, you can swap channels without changing orchestration.

The boundary also affects failure handling. If the client disconnects, does the run stop, continue in the background, or checkpoint for later resumption? That answer belongs to the application service and runtime policy, not to a route handler. The route can observe the disconnect and propagate cancellation, but it should not invent its own execution semantics.

Backend-only execution

Backend-only execution is the most conservative and often the most governable topology. The client sends requests to a server-hosted agent service. The service owns model access, tool invocation, memory, persistence, and observability. Clients remain thin and do not hold provider credentials.

This model gives you strong central control. Secrets stay server-side. Tool access stays close to internal systems. Memory and checkpoints sit near the databases and event infrastructure they depend on. Observability is cohesive because traces, logs, and metrics can cover the entire request path from inbound transport through provider calls and tool side effects.

The trade-off is latency and centralization. Every interactive turn pays for a network round trip before model execution even begins. Mobile and browser experiences depend on network quality. Offline capability is minimal unless you add local caches or fallback logic. Compute costs are also fully centralized, which is often desirable for governance but

may increase infrastructure spend.

Use backend-only execution when secret safety, policy centralization, tool locality, and audit requirements outweigh the benefit of local execution. It is usually the default for enterprise systems, especially when agents can mutate business state or touch internal services.

Client-server shared-core topology

A shared-core topology keeps orchestration logic and state models in common code while still centralizing the actual model invocation on the backend. Clients may perform local preprocessing, UI-specific formatting, or limited context preparation, but the server remains the execution authority for provider calls and privileged tools.

This model is attractive when you want consistent behavior across JVM, Android, iOS, JS, and WasmJS without exposing secrets or duplicating policy. The shared core can validate inputs, structure prompts, classify intent, and interpret results in the same way everywhere. The backend then performs model access and domain-side effects.

The complexity comes from split debugging. A defect may appear to be in the client because the user sees it there, while the actual cause is a backend execution path or a mismatch between shared-core version and server deployment. Release coordination becomes more important because a change to the shared request model may affect both clients and the server.

This topology works best when local execution offers real user value but privileged operations must remain centralized. It is a strong fit for multiplatform applications with rich clients and server-controlled model access.

Edge-hosted execution

Edge-hosted execution places the agent or a stateless portion of its serv-

ing layer close to the user, often in regionally distributed infrastructure. The goal is lower perceived latency and better geographic responsiveness. In practice, this works best when the runtime can operate with minimal dependency on centralized stateful systems.

The advantage is obvious: shorter network paths for request-response traffic. For read-heavy or stateless workloads, that can improve interaction quality. It can also reduce the pressure to build elaborate client-side fallbacks solely to mask round-trip delay.

The hard part is data gravity. If tools, memory, and checkpoints live in central systems, an edge-hosted agent still spends much of its time calling back to the core region. You have then added deployment complexity without truly localizing the latency-sensitive work. Observability also becomes harder because traces span more infrastructure zones, and failure diagnosis often crosses regional boundaries.

Edge hosting is strongest when the agent can remain relatively stateless, or when the state it needs can be cached or replicated safely. It is weaker when the workflow is deeply entangled with transactional business systems.

Split execution

Split execution deliberately divides the work across environments. A client or edge component may handle redaction, lightweight retrieval, draft generation, or user-interface shaping, while a backend agent service performs model calls that require protected credentials, privileged tools, or durable workflow control. Another version of split execution moves non-sensitive preprocessing to the device while the final decision-making and action execution remain centralized.

This topology can satisfy product requirements that no single placement can handle alone. You might support offline drafting on mobile, then sync and enrich server-side later. You might allow browser-

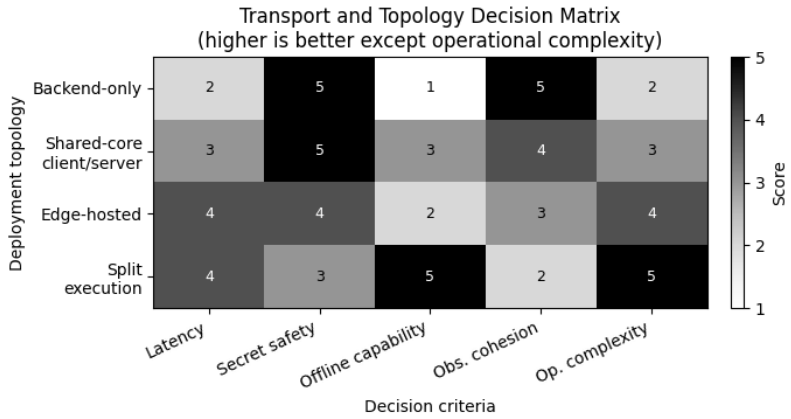
side formatting or prompt scaffolding but keep provider access server-side. You might let an edge worker handle region-local request admission and stream management while a core backend owns memory and checkpointing.

The risk is over-fragmentation. Once execution spans too many places, causality becomes difficult to reason about. Which component owns the canonical run ID? Which component may emit durable memory writes? Where does cancellation terminate? Which layer is allowed to retry? Without clear answers, split execution turns into a distributed guessing game.

Use split execution when a specific requirement justifies the added complexity: offline behavior, privacy-preserving preprocessing, strict regional latency, or staged user interaction. Do not adopt it merely because the shared code structure makes it possible.

Decision signals

A topology decision becomes easier when you score it against the characteristics that matter operationally: latency, secret safety, offline capability, observability cohesion, and complexity of operation. The point is not mathematical precision. The point is to force the team to state which trade-offs it is accepting.



The matrix is useful because it exposes hidden assumptions. If your system must support offline behavior, backend-only execution starts at a disadvantage no matter how clean the server architecture is. If secret safety and observability cohesion are non-negotiable, split execution deserves extra scrutiny even if it improves user-perceived latency.

Transport decoupling inside the runtime

Koog's decoupling of HTTP transport from Ktor is a practical reminder that the runtime should depend on a transport abstraction rather than a specific server stack. Older examples that pushed Ktor client objects deep into the LLM setup path age poorly because they hard-code one host library into a concern that should be replaceable.

Treat outbound transport the same way you treat inbound transport. The agent runtime should depend on a provider-facing client abstraction or factory that the host environment supplies. That keeps the core service portable across Spring, Ktor, background workers, and other JVM entry points. It also makes testing easier because you can substitute controlled transport behavior without booting the full host stack.

This design has a second benefit: it simplifies topology changes. If you decide to move from in-process provider access to an internal model gateway service, you can replace the transport adapter without rewriting orchestration logic. If the runtime is tightly coupled to a specific network stack, the topology decision becomes a refactor instead of a deployment choice.

In-process versus out-of-process serving

A separate but related decision is whether the agent runtime executes inside the host service process or behind a separate service boundary. In-process embedding reduces latency and keeps tool invocation close to application services. It also avoids extra serialization hops and allows the host service to share lifecycle and dependency injection with the agent runtime.

The cost is failure-domain coupling. A memory leak, blocking call, or runaway prompt path in the agent can degrade the host service itself. Scaling is also coupled. If one API endpoint triggers expensive model work, you may need to scale the whole application tier rather than only the agent-serving slice.

Out-of-process deployment improves isolation and allows independent scaling, but it adds network overhead and more consistency challenges for memory, tools, and side effects. If a tool needs a transactional domain write, an out-of-process agent either calls back into domain services or owns a privileged path that must mirror business rules. Neither option is free.

Choose in-process embedding when tool locality and low latency matter more than failure-domain isolation. Choose out-of-process deployment when the agent workload is volatile enough, expensive enough, or operationally distinct enough to justify a dedicated service boundary.

Streaming, resumability, and channel-specific adaptation

Transport choice is not only about request placement; it is also about response shape. Some channels want a synchronous JSON reply. Others need incremental token streaming. Others still want a fire-and-poll job model or resumable interaction tied to a long-running workflow.

Do not embed those channel mechanics into the shared runtime. Instead, let the runtime produce a stable execution result and let the transport layer adapt it. For streaming, the transport may expose partial tokens or structured events while the runtime remains responsible for the underlying execution state. For long-running workflows, the transport may return a run ID and current status while the checkpointing layer enables resumption.

This separation matters because channel needs change quickly. Browsers may move from polling to server-sent events. Mobile clients may switch between live streaming and background completion depending on battery or network conditions. If transport mechanics leak into orchestration, each channel evolution becomes a workflow rewrite.

Public APIs should stay thinner than internal execution

When agents are exposed through public or partner-facing APIs, the route or controller layer must become even thinner, not thicker. Public APIs need strict contracts, rate limiting, authentication, compatibility management, and audit behavior. They should translate external payloads into an application-owned invocation model, then delegate inward.

Do not let public API DTOs become internal workflow state. Public contracts evolve for backward compatibility and client ergonomics. Internal state evolves for orchestration correctness. If you merge them, you either freeze internal design for the sake of external consumers or you break clients every time the agent model changes.

A thin public API layer also helps when you expose the same capabilities behind different channels: HTTP for third parties, internal queues for batch work, and direct application calls for in-process UI components. One execution path remains authoritative.

Topology and artifact maturity

Koog's stable and beta module streams should influence deployment decisions. A production-critical public agent service should prefer stable artifacts and use beta modules only behind an explicit review gate. This matters more for distributed topologies than for isolated experiments because topology multiplies the cost of behavioral surprises. A beta feature used inside an internal proof of concept may be acceptable. The same feature embedded in a split-execution or edge-distributed system can turn an upgrade into a cross-platform incident.

Treat artifact maturity as part of topology risk. The more services, clients, or regions a feature touches, the higher the bar for adopting non-stable modules. That does not forbid experimentation. It simply means you should align experimental APIs with contained blast radius.

Anti-patterns that distort topology decisions

Several mistakes make teams choose topology for the wrong reasons.

- **Coupling transport DTOs to workflow state.** This makes every channel change look like a workflow change and pushes teams toward unnecessary duplication.
- **Embedding provider secrets in clients.** This may appear to reduce latency or simplify deployment, but it sacrifices governance and often forces a future retreat back to the backend.
- **Using local execution to compensate for poor service design.** If the server path is slow because tools are chatty, retrieval is unscoped, or persistence is inefficient, moving work to

the client treats the symptom rather than the cause.

- **Fragmenting execution without a canonical run model.** Once multiple components can create or mutate run state independently, checkpointing and auditability deteriorate quickly.
- **Assuming one topology must satisfy every requirement.** Many systems need more than one serving shape: synchronous backend calls for privileged operations, local draft assistance for UX, and asynchronous queue-driven execution for long tasks. The architectural goal is coherence, not uniformity.

A practical selection process

A short architecture review for transport and topology should answer five questions.

- Where must provider credentials live to satisfy your security model?
- Which tool calls require low-latency access to internal systems?
- Which user journeys truly require offline or near-offline behavior?
- Where must traces, audit records, and checkpoints remain coherent?
- Which parts of the design can tolerate independent scaling and deployment?

If credentials and privileged tools must remain centralized, start from a backend authority model. If only some journeys require offline support, split just those capabilities rather than moving the whole runtime. If observability and audit are mandatory across the full action path,

avoid distributed arrangements that cannot emit a coherent run identity.

A durable placement rule

Keep transport adapters shallow, keep the invocation contract stable, and treat runtime placement as a policy decision about secrets, latency, state, and failure domains. Backend-only execution remains the safest default when governance dominates. Shared-core clients with backend execution work well when you need consistent behavior across platforms without exposing privileged capabilities. Edge-hosted execution pays off when state can stay light and latency budgets are tight. Split execution earns its cost only when a concrete requirement such as offline work, privacy-preserving preprocessing, or staged interaction cannot be met any other way. Once those choices are explicit, transport stops being a framework detail and becomes a tool you can reshape without tearing through the orchestration core.

8.5. Assembling a Production Reference Architecture for Agentic Applications

A production agent system usually fails at the seams before it fails in the model. The prompt may be sound, the workflow may be typed, and the tool contracts may look clean, yet the deployed system still becomes fragile because transport owns business policy, tools bypass authorization, checkpoints capture the wrong state, or memory writes escape transaction boundaries. A durable Koog architecture avoids that failure pattern by separating concerns aggressively enough that each layer can evolve, fail, and recover without corrupting the others.

A useful reference architecture has five concerns that remain visible in code and deployment:

- transport adapters,
- application and service orchestration,
- agent runtime,
- tool and memory adapters,
- durable persistence and observability.

That list is small enough to reason about during reviews and precise enough to catch most architectural mistakes early. If a class or service does not fit clearly into one of these concerns, it often means two boundaries have already collapsed.

A concrete composition root

Start with an application composition root that assembles capabilities explicitly and keeps framework details outside the runtime core. The example below is intentionally generic in the Koog-facing pieces where exact constructor signatures are version-sensitive, but the structural pattern is the one you want in production.

```
import kotlinx.serialization.Serializable

@Serializable
data class WorkflowRequest(
 val tenantId: String,
 val userId: String?,
 val sessionId: String,
 val message: String
)

@Serializable
data class WorkflowReply(
 val runId: String,
 val text: String,
 val suspended: Boolean
)

class AgentFacade(
 private val appService: AgentApplicationService
) {
```

```
    suspend fun handle(request: WorkflowRequest): WorkflowReply {
    return appService.execute(request)
    }
}
```

That shape matters because it keeps the public entrypoint thin and keeps all meaningful decisions below it. The façade can sit behind Spring controllers, Ktor routes, queue consumers, or scheduled jobs. The rest of the system only cares that it receives a normalized request and returns a normalized reply.

Layer 1: transport adapters

Transport adapters own external protocols and little else. They parse HTTP payloads, WebSocket messages, queue envelopes, or scheduled-job inputs. They attach request metadata such as authentication details, tenant identity, correlation IDs, deadlines, and streaming preferences. They then delegate to application services.

Keep transport code boring. If a controller, route, or message consumer starts building prompts, choosing tools, or orchestrating retries, the architecture is already drifting. Transport types also should not become internal workflow state. External contracts evolve for compatibility; internal state evolves for correctness. Those change vectors are different.

A stable transport adapter usually performs only four steps:

- authenticate and validate the external request,
- translate it into an application-owned command,
- invoke the application service,
- map the result back to the channel format.

That discipline pays off when you add channels. The same execution

path can sit behind synchronous HTTP, server-sent event streaming, batch queue processing, or a mobile bridge without duplicating policy.

Layer 2: application and service orchestration

Application services are the policy heart of the system. They decide which workflow to run, which identity and tenant scope apply, when to load memory or checkpoints, where transaction boundaries begin and end, and what happens when external side effects fail. This is the layer that turns a generic agent runtime into an application.

A strong application service does not know about HTTP routing or mobile callbacks, and it does not expose raw repositories to the agent. It coordinates the run using stable interfaces.

```
interface AgentApplicationService {
 suspend fun execute(
 request: WorkflowRequest
 ): WorkflowReply
}
```

Behind that interface, the service typically performs these steps:

- resolve tenant, actor, and authorization scope,
- load or initialize workflow state,
- fetch retrieval context and memory under policy,
- restore from checkpoint when applicable,
- invoke the agent runtime,
- commit domain side effects through governed services,
- emit post-commit memory writes and events,
- persist the next checkpoint or final result,
- record traces, metrics, and audit artifacts.

This is also where you enforce the ordering rule that keeps production systems honest: business transaction first, durable commit second, post-commit memory or notification side effects third. If you violate that sequence, the system begins inventing history.

Layer 3: agent runtime

The runtime is where Koog contributes most directly. It owns the execution loop, workflow semantics, tool invocation boundaries, memory integration, checkpoint restoration, and model interactions. In a clean architecture, the runtime remains powerful but not omniscient. It should not know how Spring wires beans, how Ktor resolves resources, how your ORM session is scoped, or how your browser client reconnects after a network break.

The runtime should receive capability interfaces and typed state. That keeps workflows portable and testable. It also makes failure handling explicit because each capability boundary can declare how it behaves: fail the run, degrade visibly, retry within bounds, or suspend for later resumption.

Koog's recent changelog history is instructive here. Swallowed memory storage errors, mutable prompt races, streaming flow invariant problems in Ktor-related contexts, and breaking changes after deprecation removals all point to the same architectural lesson: runtime state surfaces must be explicit, concurrency assumptions must be conservative, and version-sensitive APIs need containment. You should design as if mutable state will eventually be touched concurrently, as if storage writes can fail independently of model calls, and as if library upgrades will surface any hidden dependency on deprecated behavior.

Layer 4: tool and memory adapters

Tool and memory adapters are the bridge between orchestration and the rest of the application. They are not the domain model itself and

they are not raw infrastructure wrappers. Their job is to expose exactly the capabilities the runtime should have.

A productive pattern is to keep three sub-layers inside this concern:

- domain-facing tool services,
- retrieval and memory adapters,
- checkpoint and resumability adapters.

A domain-facing tool service should call application-owned services that already enforce authorization, validation, and idempotency. A retrieval adapter should encode tenant and policy scope before any ranking or augmentation logic runs. A memory adapter should make write failure behavior explicit instead of quietly dropping state. A checkpoint adapter should persist execution state, not transport or framework objects.

The distinction between memory and checkpointing is especially important. Memory stores reusable context. Checkpoints store resumable execution state. If you conflate them, you either reconstruct workflows from lossy transcript fragments or you pollute long-term knowledge with runtime internals.

Layer 5: durable persistence and observability

Durable persistence and observability form the architectural floor. They should not be scattered ad hoc through controllers, tools, and provider adapters. Persisted state must have a clear owner, and telemetry must be rich enough to reconstruct how a run behaved under failure.

Persistence in a production agent system usually spans at least four distinct stores:

- business data stores,
- memory stores,
- checkpoint stores,
- telemetry and audit sinks.

These may share physical infrastructure, but they should remain different logical concerns. Business data is authoritative for domain truth. Memory is selective context for future runs. Checkpoints are resumption artifacts. Telemetry explains behavior. Audit records explain accountability. When one store tries to play all roles, schema design and retention policy become impossible to reason about.

Observability should capture more than provider latency. A useful run trace includes transport ingress, request normalization, workflow selection, retrieval fetches, memory reads and writes, tool invocation boundaries, checkpoint load and save operations, final outcome, and any policy-driven suspension or retry decision. If you only instrument model calls, you will know token timing but not why the system actually failed.

An end-to-end operational flow

A production reference architecture becomes concrete when you walk a single request through it.

A request enters through HTTP, a queue, or another transport. The transport adapter authenticates the caller, validates the external payload, and constructs a transport-neutral command. The application service resolves tenant context, actor permissions, and workflow selection. If the request resumes an interrupted run, it loads a checkpoint. If it starts a fresh run, it initializes typed workflow state.

The service then asks retrieval and memory adapters for scoped context. Those adapters operate under application policy rather than

prompt-time convenience. The runtime executes using Koog workflows, tools, and model bindings supplied by the composition root. Tool calls cross back into governed application services or constrained adapters rather than touching repositories directly.

If the workflow performs domain mutations, the application service controls the transaction boundary. Once the transaction commits, the system can safely emit memory updates, events, or downstream notifications. If the workflow may continue later, the next checkpoint is written as durable execution state. Observability spans the whole path, and the final response is mapped back into the channel-specific format.

That flow sounds conservative because it is. Conservatism here is an asset. Production agent systems tend to accumulate asynchronous work, streaming responses, retries, and partial failures quickly. A strict execution path gives you somewhere stable to attach those features.

Concurrency rules that prevent subtle corruption

Concurrency problems often emerge late because low-volume testing rarely exercises the interleavings that production traffic creates. Koog's changelog history already offers several warnings: mutable prompt races, streaming-related flow issues in some transport contexts, and reliability fixes around memory persistence. Take those as design input, not merely release notes.

Keep mutable workflow state confined to a single execution context whenever possible. Do not share mutable prompt builders, per-run memory buffers, or partial response assemblers across requests. If you need shared caches or registries, make them immutable or synchronize them at the host layer rather than inside prompt logic.

Streaming deserves special care. A streaming response crosses dispatcher and cancellation boundaries more often than synchronous request-reply flows. Treat stream emission as a transport concern lay-

ered on top of runtime events, not as a side channel woven through business logic. That separation makes it easier to preserve flow invariants and easier to fallback from live streaming to buffered completion if the channel cannot sustain it.

Resilience rules that preserve correctness

A production agent system is not resilient because it retries aggressively. It is resilient because it knows which failures are safe to retry and which ones must suspend, surface an error, or require human review.

A practical resilience policy distinguishes at least four cases:

- transient provider or transport failures,
- deterministic validation or authorization failures,
- partial side-effect failures after useful progress,
- corrupted or incompatible checkpoint or memory state.

Transient provider failures may justify bounded retries with jitter, as long as the surrounding tool actions are idempotent or have not yet occurred. Validation and authorization failures should fail fast. Partial side-effect failures often require suspension and resumption rather than blind retry. Corrupted checkpoint state should surface explicitly, ideally with versioned state models that allow migration or controlled rejection.

Checkpoint restoration is particularly sensitive. Koog provides checkpoint persistence and restoration primitives, but the host architecture determines whether resumed runs remain meaningful. A checkpoint that references deleted business objects, expired credentials, or an incompatible tool contract should not quietly continue as if nothing

changed. Resumption logic should re-validate the minimal context needed for correctness.

Versioning and upgrade containment

Production architectures live longer than any one library version. Koog 1.0 established a stable baseline and formalized stable versus beta module streams. Use that distinction. Keep production-critical paths on stable artifacts unless a beta capability is justified by a specific business requirement and isolated behind a narrow boundary.

Contain version-sensitive APIs inside adapters. If a provider binding, a framework integration module, or a checkpoint-related helper changes, you want that impact to stop at the boundary layer rather than ripple through workflows and application services. This is the same reason to avoid scattering direct framework-specific types through shared logic. Every leaked dependency widens the blast radius of upgrades.

State schema versioning deserves the same discipline. Typed workflow state, checkpoint payloads, and memory metadata should evolve intentionally. Additive changes are easier to roll out across mixed client and server releases. Breaking state changes need explicit migration or controlled invalidation. Production failures caused by silent schema drift are among the hardest to diagnose because the code may still compile cleanly while recovery behavior decays.

Operational ownership and deployment boundaries

A reference architecture also makes ownership visible. Platform teams often struggle when no one knows whether a failure belongs to the API team, the data platform team, the model provider integration team, or the workflow authors. Architecture can reduce that ambiguity if each concern maps to a clear operational boundary.

Transport adapters belong to the teams that own channels and exter-

nal contracts. Application services belong to teams that own business policy. Runtime workflows belong to teams that own agent behavior. Tool and memory adapters belong to teams that bridge policy to infrastructure. Persistence and observability belong to platform-facing teams that ensure durability and diagnosability.

You do not need separate org charts for each concern, but you do need clear ownership lines. Without them, incident response degenerates into everyone inspecting prompts while the actual bug lives in a checkpoint serializer or post-commit event path.

A production hardening checklist

Before you call an architecture production-ready, verify the following properties.

- Transport adapters are thin and do not assemble prompts or invoke repositories directly.
- Application services own tenant scope, transaction boundaries, and workflow selection.
- Runtime state is explicit, typed, and not shared mutably across requests.
- Tool calls pass through governed services or constrained adapters.
- Memory write failures have explicit policy; they are not swallowed.
- Checkpoints store resumable execution state, not framework objects.
- Streaming paths respect cancellation and dispatcher boundaries.
- Observability covers tools, memory, checkpoints, and transport, not only provider calls.

- Stable and beta dependencies are reviewed according to blast radius.
- Schema evolution for workflow state and checkpoints is deliberate.

A system that passes this checklist is not guaranteed to be flawless, but it is much easier to reason about under pressure.

The architecture that scales with complexity

As the system grows, you will add more workflows, more channels, more state, and more operational requirements. The only way that growth stays manageable is if the architecture continues to enforce the same basic discipline: transports adapt, application services govern, the runtime orchestrates, adapters mediate capabilities, and persistence plus observability make behavior durable and visible. Koog fits naturally into that model because it provides the agent runtime, tools, memory, checkpoint persistence, observability hooks, Java interop, and framework integration pieces you need. The sustainability comes from where you put those pieces and how strictly you keep their boundaries intact.