

29 ARE THE CLASSIC DESIGN PATTERNS
STILL VALID WITH MODERN JAVA?

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

Edition 5 / 2026

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY #JAVAPRO

From AI as a Feature to AI as Infrastructure

55 LANGCHAIN4J AGENTIC WORKFLOWS:
FROM AI CALLS TO MULTI-AGENT SYSTEMS IN JAVA

69 SPEC DRIVEN DEVELOPMENT: WHY THE FUTURE OF AI-
ASSISTED DEVELOPMENT STARTS WITH THE SPECIFICATION

75 BEYOND UML: CLEAN SOFTWARE ARCHITECTURE
IN THE AGE OF AI GENERATED CODE

80 FROM PROTOTYPE TO PRODUCTION: BUILDING SAFE
& RELIABLE AI AGENTS WITH SPRING AI & MCP

96 HEXAGONAL ARCHITECTURE FOR ML: PORTS,
ADAPTERS, & MODEL VERSIONING IN SPRING BOOT

123 DEVOPS PATTERNS & JAVA 26 FOR ON-PREMISES LLM
PLATFORMS IN SAFETY-CRITICAL ENVIRONMENTS



JCON
EUROPE
COLOGNE
2027

April 26 - 29, 2027
www.jcon.one



SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

From AI as a Feature to AI as Infrastructure

Artificial Intelligence has moved beyond the experimental phase.

The first wave of AI adoption focused on exploration: integrating language models into applications, experimenting with assistants, generating content, or automating isolated tasks. It was an exciting time driven by rapid innovation, curiosity, and the promise of unprecedented productivity gains.

But reality has changed the conversation.

Today, organizations are no longer asking whether AI can generate useful results. They are asking much harder questions:

- How do we operate AI systems reliably?
- How do we version and audit models?
- How do we secure AI-driven workflows?
- How do we govern prompts, embeddings, and data access?
- How do we integrate AI into long-lived enterprise systems?
- How do we build architectures that remain maintainable once the hype fades?

This issue focuses on exactly that transition: from AI experimentation to AI engineering.

Across the articles in this edition, one theme repeatedly emerges: AI is no longer just a feature. It has become an architectural concern.

Modern Java is uniquely positioned for this new reality. While other ecosystems often dominate rapid prototyping, Java continues to excel where systems must remain stable, observable, governable, and operational for years. Features introduced in recent Java releases - from Virtual Threads and Structured Concurrency to improvements in startup performance and vectorized computation - are not merely language enhancements. They are foundational building blocks for modern AI platforms and large-scale backend systems.

At the same time, the industry is learning that introducing AI into production systems also introduces entirely new responsibilities: governance, observability, reproducibility, auditability, security, and operational control.

This issue explores those challenges from multiple perspectives.

We examine how classic software architecture evolves in the presence of machine learning systems. We discuss hexagonal architectures for AI applications, model versioning strategies, retrieval-augmented generation pipelines, and on-premises LLM platforms for safety-critical environments. We look at DevOps patterns for AI systems, reproducible development environments, and the growing importance of Prompts-as-Code and semantic testing.

Beyond the purely technical dimension, we also address organizational and legal realities. AI-first cultures, enterprise adoption strategies, source code as legal evidence, software forensics, and the implications of AI-generated code are no longer theoretical concerns. They are rapidly becoming part of everyday software engineering practice.

At the same time, this issue revisits a fundamental question for the Java ecosystem itself: What is the role of Java in the AI era?

The answer emerging throughout these pages is both clear and encouraging. Java is not merely adapting to AI.

Java is becoming one of the most compelling platforms for building reliable AI systems. Not because it follows every trend, but because it provides what production systems ultimately require: clarity, structure, operational stability, scalability, and long-term maintainability.

The future of AI will not be defined solely by larger models or more autonomous agents. It will also be defined by the quality of the systems surrounding them.

That future requires software engineering. And software engineering remains one of Java's greatest strengths.

Enjoy this issue.

Author: Markus Kett

Markus Kett and his teams have been working on IDE tools for Java and database development, as well as various open-source projects for 20 years. Markus is CEO and co-founder of MicroStream, the company behind the Eclipse open-source projects EclipseStore, Eclipse Serializer, and RapidClipse IDE. He is also the editor-in-chief for the free JAVAPRO magazine in Germany and the founder and co-organizer of the Java community conference series JCON. He is an independent editor for several magazines, and a speaker at numerous international developer conferences, user groups, and meetups.



LinkedIn



JAVAPRO

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

Contents

JAVAPRO

Richard Fichtner

JCON EUROPE 2026: Why the Java Community Still Meets in Person	10
Why the Venue Matters	11
The Real Conference Happens Between the Talks	12
AI Was Everywhere - But in a Pragmatic Way	13
The Java Ecosystem Continues to Evolve	13
Community Is Still the Foundation	14
Looking Ahead to JCON EUROPE 2027	15

Core Java

Lutske de Leeuw & Johannes Bechberger

Java 26 Is Boring	16
Which Is Why It Is Brilliant	17
How to Upgrade	17
What Java 26 Actually Changes	18
G1 GC: Improve Throughput by Reducing Synchronization (JEP 522)	18
HTTP/3 in the Standard HTTP Client (JEP 517)	18
PEM Encodings of Cryptographic Objects (JEP 524)	19
The End of Java Applets	19
Preview and Incubating Features	20
Conclusion	21

Core Java

Babneet Singh

Structured Concurrency in Java 26 (JEP 525 Deep Dive)	23
The Java 26 Evolution of Safer, Predictable Multi-threading	24
Concurrency Without Structure: Ambiguity at Runtime	24
Lexical Scope as a Lifetime Boundary	24
What Changed in the Sixth Preview (JEP 525)	24
Structured Concurrency in Practice: An E-Commerce Product Page	25
Graceful Degradation with onTimeout ()	26
Joiners as Explicit State Machines	27
Interruption and Deterministic Teardown	27
Memory Visibility and Happens-Before	27
Virtual Threads and Bounded Abundance	28
Observability Through Hierarchy	28
When Not to Use Structured Concurrency	28
Conclusion: Concurrency with Enforced Boundaries	28

Core Java

Andreas Jürgensen

Are Classic Design Patterns Still Valid With Modern Java?	29
Once upon a time...	30
Gof Design Patterns	30
Java	30
The GoF Design Patterns	30
The Evolution of Java	31
Conclusion	37
Lookout	37

API & Frameworks

Artur Skowronski

The Gen AI Iceberg - Java Tooling Edition	38
--	-----------

Level 1: The Tip of the Iceberg (The “@Autowired ChatClient” Zone)	39
Level 2: Just Below the Surface (The “Wait, What’s Under That Starter?” Zone)	41
Level 3: The Depths (The “My pom.xml Has An ONNX Dependency” Zone)	42
Level 4: The Abyss (The “I’m Running CUDA From My JVM” Zone)	44
Level 5: The Void (The “I Read JEP Drafts For Fun” Zone)	45

API & Frameworks

Damiana Nascimento

Building AI-Driven Java Systems with Spring AI and Java 26	47
AI Has Finally Found a Practical Path in the Java Ecosystem	48
What Makes AI-Driven Java Systems Practical Now	48
Reference Architecture for AI-Driven Java Systems	49
Recommended Architectural Layers	49
Figure 1 — RAG Pipeline	49
RAG Trade-Offs in the Architecture	50
Applying Modern Concurrency in AI-Driven Java Systems	50
Virtual Threads: Concurrency Without Accidental Complexity	50
Structured Concurrency: Coordinating Composite Workflows	50
Where Virtual Threads Help — and Where They Do Not	50
Spring AI as an Integration Layer, Not a Magic Shortcut	51
ChatClient and ChatModel: When to Abstract and When to Control	51
Tool Calling: Power and Risk in the Same Package	51
Spring AI and Observable Integration	52
Operating AI-Driven Java Systems in Production	52
AI Orchestration With Spring AI	53
AI as Part of the Architecture, Not an Exception	54

API & Frameworks

Loïc Magnette

LangChain4j Agentic Workflows: From AI Calls to Multi-Agent Systems in Java	55
Getting Started with LangChain4j Agentic Workflows	56
The Building Block of Agentic Workflows: What Is an Agent?	56
Shared State with AgenticScope	57
Agentic Workflow Pattern 1: Sequential	59
Agentic Workflow Pattern 2: Loop	60
Agentic Workflow Pattern 3: Parallel	61
Agentic Workflow Pattern 4: Conditional	61
Composing Agentic Workflow Patterns	62
Error Handling as a First-Class Concern	63
Goal-Oriented Agents	63
Going Agentic: The Supervisor Pattern	64
Mixing AI and Non-AI Agents	65
Human-in-the-Loop	66
When to Use What	67
Conclusion	67

AI & ML

Daniel Sogl

Spec Driven Development: Why the Future of AI-Assisted Software Development Starts with the Specification	69
The Promise and the Disillusionment	70
The Root of the Problem: Context Is King	70
What Is Spec Driven Development?	71
The Tools: Spec Kit and OpenSpec in Detail	71
What Companies Report	72
The Broader Ecosystem	73
Critical Assessment	73

Conclusion and Outlook	73
Sources	74

AI & ML Sven Reinck

Beyond UML: Clean Software Architecture in the Age of AI-Generated Code	75
We Write More Code Than Ever Before	76
Two Ways to Use AI in Programming	76
If Only This Existed	77
UML Is the Solution, but Not How We Know It	77
Dependencies as a Common Language	77
Interactive Dependency Visualization as a Thinking Tool	78
From Understanding to Action: Refactoring With Less Risk	79
Conclusion	79

AI & ML Mihaela Gheorghe-Roman

From Prototype to Production: Building Safe and Reliable AI Agents with Spring AI and MCP	80
Spring AI Production Requirements: What Changes	81
Foundation for Spring AI Production: Why Spring AI and MCP	81
Building the Foundation: Basic Agent Setup	82
Adding Tools: MCP Integration	83
Production Safety: Implementing Guardrails for Spring AI Agents	84
Failure Recovery in Spring AI Production Systems	87
Spring AI Production Observability: Metrics and Logging	88
Lessons Learned and Key Takeaways	90
References and Further Reading	90

AI & ML Lokesh Chenta

Enterprise Playbook for Generative AI Adoption	91
Era of Generative AI and Agentic Systems	92
Generative AI for Organizational Productivity	92
Enterprise Readiness for AI-Native World	94
Adopting AI-First Culture	95
Final Thoughts	95

Architecture & Microservices Nikita Golovko

Hexagonal Architecture for ML: Ports, Adapters, and Model Versioning in Spring Boot	96
The Model Changed - The Architecture Wasn't Ready	97
Hexagonal Architecture in 60 Seconds	97
Domain Model	98
Define the Driven and Driving Ports	99
The Domain Service	100
Driving Adapter	101
Driven Adapters	102
Connecting Everything in One Place	103
Model Versioning Strategy	104
The Model Changed. This Time, the Architecture Was Ready	107

Architecture & Microservices Sven Ruppert

Open-Core with Core Java and Vaadin — Part 1	108
Architectural overview and project structure	109
The domain: counter and events	110

The extension API	111
Discovery via the ServiceLoader and the FeatureRegistry	113
Vaadin integration: AppLayout and dynamic routes	114
Transition to part two	117

Serverless

Bhupendra Mishra

Serverless Functions Reimagined with Java 25	118
The Serverless Challenge for Java	119
Faster Startup and Cold-Start Improvements	119
Garbage Collection and Predictable Latency	120
Leaner Java Objects for Serverless Workloads	120
Virtual Threads for Scalable Concurrency	120
Structured Concurrency for Reliable Execution	121
Simplifying Serverless Architecture with Java 25	121
Conclusion	121

DevOps

Cornelius May & Eldar Sultanow

DevOps Patterns and Java 26 for On-Premises LLM Platforms in Safety-Critical Environments	123
Governance Before Feature Adoption: Stability, Support, and Blast Radius	124
On-Premises RAG on Kubernetes	124
Java 26: Building Blocks for Efficient AI Backends	126
DevOps Patterns for the LLM Era	127
Security & Governance: The Five Control Points	128
Implementation Strategies: Criteria for Framework Selection	128
Sovereignty Through Technology	129

IDE & Tools

Sascha Selzer

Reproducible Dev Environments Made Easy	130
SDKMAN! – The De Facto Standard for Java Projects	131
asdf – When You Need More Than Java	131
mise – The Better asdf?	132
Devbox – The Easier Way To Nix	133
Conclusion	135
References	135

Project Management

Stefan Wagenpfeil

Code at Court – Java as Evidence	136
The Algorithmization of Law	137
Legal framework	137
Identifiability, Security, and Integrity	138
Source Code Changes	138
Analysis Methodology – From Bytecode to Business Logic	139
Artificial Intelligence	139
Conclusion and Outlook	140

Project Management

Marco Schulz

RTFM – Usable Documentation	141
Why Documentation Matters	142
The Problem with Documentation as a “Victorian Novella”	142
Writing Skills Are a Team Capability	142
Why Tests Alone Are Not Documentation	142

Tooling Can Help, But It Cannot Replace Discipline	143
Wikis Need Structure, Formatting, and Care	143
Graphics Must Add Value	143
Keep Documentation Concise, Centralized, and Maintainable	143
Conclusion: Documentation Enables Agile and DevOps	144

Security	<i>Mohammad-Ali A'râbi</i>
-----------------	----------------------------

The Whispering JAR: Java Security Lessons Hidden in a Fantasy Tale	145
Threats from the Past	146
The Fishy JAR	147
The Worm Spreading	149
The Box of Sand	150
Conclusion	150
References	151



Publisher:

JAVAPRO
 Im Gewerbepark 29
 92681 Erbendorf
 Germany
 E-Mail: info@javapro.io
 Website: <http://www.javapro.io>

Editor in Chief:
 Markus Kett (V.i.S.d.P.)

Editorial:
info@javapro.io

Design, Layout & Print:
 Impuls Mediengruppe GmbH
 Im Gewerbepark 29
 92681 Erbendorf
 Germany

Copyright (c) 2025
 Impuls Mediengruppe GmbH

All rights reserved.

Java(TM) is a registered trademark of
 Oracle Corporation.

Javapro is an independent magazine
 and is not sponsored by Oracle
 Corporation.

Articles marked with a name do not
 necessarily reflect the opinion of the
 editors.

The images featured in this issue are
 sourced from royalty-free platforms
 such as Pixabay and Unsplash,
 contributed by our authors, or
 creatively generated with the help of
 artificial intelligence.

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors





APR
20-23



JCON EUROPE 2026: Why the Java Community Still Meets in Person

JAVAPRO

JCON

JavaCommunity

Author: Richard Fichtner

Richard Fichtner is the Principal Software Architect at XDEV Software GmbH, with over 20 years of experience in the software industry. He works at the intersection of business and technology, contributing to the open-source community through projects like RapidClipse (<https://www.rapidclipse.com>) and advocating for best practices in Apache Maven. Richard is passionate about sharing knowledge on Java technologies, making him a frequent conference speaker. He leads the Java User Group Oberpfalz, co-organizes the JCON conferences (<https://jcon.one>), and has been recognized as a Java Champion, Oracle ACE Pro, IBM Champion and Vaadin Champion. With a Master of Science in Applied Computer Science, his expertise spans Java, clean code, cloud technologies, and pragmatic development practices, focusing on improving developer productivity through cloud solutions. Richard enjoys relaxing by a campfire and listening to/playing country music in his spare time.



LinkedIn

Every year, when we start planning JCON EUROPE, one question always guides us:

How do we create a conference that developers genuinely enjoy attending?

Not just another event with slides and sponsor booths. Not a place where people passively consume information and leave. We wanted to create an environment where developers actively connect, discuss, challenge ideas, and solve real-world problems together.

Looking back at JCON EUROPE 2026, I believe this spirit was visible everywhere.

From April 20 to 23, more than 1,000 developers from over 60 countries came together at the Cinedom cinema complex in Cologne to exchange ideas around Java, AI, cloud-native architectures, software modernization, developer tools, and the future of our ecosystem.

And despite all the changes in our industry, one thing became very clear once again:

In-person community still matters.

Why the Venue Matters

One of the most distinctive aspects of JCON remains the cinema venue itself.

At first glance, hosting a software conference in a multiplex cinema sounds unusual. But once you experience live coding on a massive cinema screen, the idea immediately makes sense.

Complex code examples remain readable even from the back rows. Architecture diagrams become easier to follow. And during debugging sessions or deep technical explanations, attendees can actually follow along without struggling to read tiny fonts on overloaded presentation slides.



As organizers, we often hear attendees mention that this changes the entire learning experience. It creates a stronger sense of focus and immersion than traditional conference rooms.

And Java developers appreciate details.

The Real Conference Happens Between the Talks

Of course, JCON had more than 100 sessions, over 100 speakers, and workshops covering everything from OpenJDK internals to AI-assisted development.



Abb. 1: AI Workshop with Don Bourne & Michal Broz

But if you ask me what makes JCON special, the answer is simple:

The hallway track.



Abb. 2: The hallway 2026

Some of the most valuable moments at JCON never appear on the official schedule. They happen after a session ends, when small groups form in the hallways and conversations continue naturally.

Someone asks a follow-up question.

Another developer explains how they solved a similar issue in production.

A framework maintainer suddenly joins the discussion.

Five minutes later, what started as a quick chat turns into a practical architecture discussion with direct relevance to somebody's daily work.

This is difficult to reproduce online. It cannot really be streamed. And it is one of the main reasons why physical conferences remain highly relevant in 2026.

One of my favorite pieces of feedback this year came from Adam Bien:

“How do you measure the greatness of a conference? I arrived here, tried to walk around this small circle and it took me 3 hours, since I had so many nice people approaching me with interesting conversations.”

I think this perfectly captures what makes JCON special. The value of the conference is not only measured by the sessions on the schedule, but by the spontaneous discussions, the reconnecting with old friends, and the unexpected conversations that turn into new ideas, collaborations, or sometimes even solutions to real production problems.

AI Was Everywhere - But in a Pragmatic Way

Artificial Intelligence was naturally one of the major themes at this year's conference.

With the new Premium AI Track, we wanted to approach the topic from a practical Java developer perspective instead of simply following industry hype.

The discussions focused on questions developers are actually facing today:

- How can AI improve developer productivity without creating dependency?
- Which models can realistically run locally?
- How do organizations handle privacy-sensitive environments?
- What does AI integration look like inside existing enterprise Java systems?
- How do teams maintain control over architecture and maintainability?

The interesting part was that the conversations were far more nuanced than a year ago.

The industry is slowly moving beyond simplistic “AI replaces developers” narratives. Instead, teams are beginning to understand where AI genuinely helps and where engineering experience still matters enormously.

Another highlight around the AI track was the presentation of the **AI Sustainability Award** by the **eco Verband der Internetwirtschaft e.V.** and **KI Bundesverband** to [Dell Technologies](#) and [Cyrock.AI](#).

The award recognized a remarkable engineering achievement focused on one of the biggest challenges in modern AI infrastructure: energy consumption.

The companies demonstrated an architecture capable of reducing compute resources, energy usage, and CO2 emissions for generative AI workloads by up to 80 percent. Particularly interesting for many attendees was the pragmatic engineering approach behind it: instead of relying on permanently running monolithic systems, the platform uses highly modular compute units that are only active when actually needed.

The discussions around this award showed that the future of AI will not only be defined by model capabilities, but also by efficiency, sustainability, and the ability to operate powerful AI systems responsibly at scale.



Abb. 3: AI Sustainability Award 2026: Markus Kett (Cyrock.AI) and Johannes Strauss (Dell Technologies)

The Java Ecosystem Continues to Evolve

One thing that always stands out at JCON is how broad and healthy the Java ecosystem remains.

You meet contributors from OpenJDK, maintainers from Apache Maven, experts from Quarkus, Jakarta EE, Vaadin, Testcontainers, Spring, AI tooling vendors, cloud providers, and many more.



Abb. 4: Workshop 2026

And perhaps most importantly, these people are approachable.

One of the strengths of the Java community has always been its openness. Junior developers can walk up to highly experienced engineers and simply start a conversation.

That culture still exists.

This year also showed how strongly companies continue to invest in Java.

IBM presented “IBM Bob,” their new AI-supported developer platform. Microsoft expanded its Java community engagement further. Companies like Azul, JetBrains, Moderne, and many others actively contributed to discussions around tooling, modernization, observability, and developer productivity.

The ecosystem is evolving quickly, but Java continues to provide something extremely valuable:

Stability combined with continuous innovation.

Community Is Still the Foundation

One of the most rewarding aspects of organizing JCON is seeing how international the event has become.

Developers from more than 60 countries attended this year.

Different industries. Different cultures. Different architectural challenges.

But once conversations begin, everybody speaks the same technical language.

That creates a very special atmosphere.

You quickly realize that many teams around the world struggle with similar questions:

- How do we modernize legacy systems safely?
- How do we adopt AI responsibly?
- How do we keep systems maintainable over decades?
- How do we help younger developers grow?

These discussions create perspective. And often, they create friendships as well.

As organizers, that is ultimately what we try to build:

A conference where technology matters, but people matter even more.

One moment that reflected this spirit particularly well was the presentation of the **Java Community Leadership Award** to Sharat Chander. The award recognized 25 years of passion, leadership, and commitment to the global Java community.

Over the years, Sharat has played an important role in connecting developers, companies, user groups, and technology leaders across the ecosystem while continuously supporting and encouraging new generations of developers.

Honoring him at JCON felt very fitting because the conference itself is built around many of the same ideas: openness, mentorship, collaboration, and long-term community engagement.



Abb. 5: Java Community Leadership Award 2026: Sharat Chander and Richard Fichtner

Looking Ahead to JCON EUROPE 2027

Planning for the next edition already started while this year's conference was still running.

[JCON EUROPE 2027](#) will take place from **April 26th to 29th** in Cologne.

The challenge now is not simply to grow larger. The challenge is preserving the atmosphere that makes JCON valuable in the first place.

As AI accelerates software development and remote work continues reshaping collaboration, I believe conferences like JCON become even more important. Not because information is unavailable online. But because trust, collaboration, mentorship, and real technical exchange still happen best face to face.

The Java ecosystem has always been unusually community-driven. JCON exists because thousands of people actively contribute knowledge, organize meetups, mentor newcomers, build tools, maintain frameworks, and help each other improve. That spirit was visible everywhere in Cologne this year. And honestly, seeing that gives me a lot of optimism for where Java is heading next.

More impressions and photos from JCON EUROPE 2026 are available on Flickr:

<https://www.flickr.com/photos/jcon-conference/albums/>

www.jcon.one

› [Back to the Table of Contents](#) ‹



Java 26 Is Boring

CoreJava

Java26

JEP522

Authors: Lutske de Leeuw & Johannes Bechberger

JCON SPEAKER

Lutske de Leeuw is a Software Engineer at Craftsmen, specializing in full-stack development with a focus on Java and Angular. Passionate about machine learning, board games, and cats. Committed to sharing knowledge so that everyone can benefit. Loves organizing events like Devox4Kids, ApeldoornJUG, JUG Noord, and other knowledge-sharing initiatives.



LinkedIn

JCON SPEAKER

Johannes Bechberger is a JVM developer working on profilers and their underlying technology in the SapMachine team at SAP. This includes improvements to async-profiler and its ecosystem, a website to view the different JFR event types, and improvements to the FirefoxProfiler, making it usable in the Java world. His work today comprises many open-source contributions and his blog, where he regularly writes on in-depth profiling and debugging topics. He also works on hello-ebpf, the first eBPF library for Java. His most recent contribution is the new CPU Time Profiler in JDK 25.



LinkedIn

Want to Dive Deeper? They are speaker at JCON! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!

Which Is Why It Is Brilliant

When people hear “boring tech”, they usually mean old, slow, or not innovative. But in production, boring implies something very different. Boring means predictable, with no surprises. Boring means your system still works at 3 a.m. when nobody wants to debug a memory leak. And boring also means that you can understand your system years after you wrote it.

Many platforms try to impress developers with significant changes, shiny rewrites, or breaking updates. Java took another path. Java optimizes for trust.

If a Java release feels boring, that usually means:

- Your code still compiles
- Your API's will still work
- Your upgrade does not turn into a rewrite project

And that's not a weakness. That's why Java has survived for decades.

If you look back at the different Java versions, you notice that not much has changed since Java 8. The significant changes to the language itself are minor: `var` and `record` are just syntax sugar, and even the new `switch` expression and pattern matching feel like an evolution of existing syntax rather than something completely new. Most additions, like the foreign function interface or virtual threads, are only beneficial for library developers and not something the average developer will likely use. But allowing library developers to write faster, better code means that anyone gets better foundations for their applications. So Java can be proud to be boring.

For example, take:

```
1 import java.util.*;
2 class Example {
3     Map<String, List<Set<Map<Integer, String>>>> complex;
4     List<? extends Number> wildcardExtends;
5     List<? super Integer> wildcardSuper;
6 }
```

Can you guess the Java version of the above snippet?

It's Java 5.

```
1 class Example {
2     record Pair<T, U>(T first, U second) {}
3     Pair<String, Integer> pair = new Pair<>("test", 42);
4 }
```

What is the Java version of the above snippet?

It's Java 16, released in 2021.

But it still feels modern. Improvements in Java are driven by its libraries and ecosystem, not by the Java language itself.

How to Upgrade

For most projects, upgrading from Java 25 to Java 26 is straightforward. If you use Maven, the change often starts and ends with updating the Java release version in your build configuration:

```
1 <properties>
2     <maven.compiler.release>26</maven.compiler.release>
3 </properties>
```

Maven will automatically use the configured JDK, and you can then rebuild and run your tests. Modern IDEs typically highlight deprecated or removed APIs and suggest alternatives where needed. Thanks to Java's strong backward compatibility guarantees, most applications compile and run without changes, making the upgrade a routine maintenance task rather than a migration project.

What Java 26 Actually Changes

Java 26 introduces many JEPs (Java Enhancement Proposals), but not all have the same impact on day-to-day development. Some changes affect how Java behaves at runtime or how you interact with core APIs. Others are previews or incubating features that are intentionally not final yet.

Honestly, the most considerable improvement in Java 26 is not in the language or its APIs; it is the improved throughput with the G1 garbage collector. So an upgrade gives you better performance.

In this section, we focus on the changes most likely to matter to Java developers today. These are features that are enabled by default, influence performance or behavior, or represent a clear step forward for the platform.

Preview and incubating features are treated differently. They are still evolving, require explicit opt-in, and may change or even be removed in future releases. Because of that, they are summarized later in this article, with a short explanation for each JEP. The goal there is awareness, not immediate adoption.

The following paragraphs, therefore, highlight the most concrete changes in Java 26. After that, we will briefly step through the preview and incubating features to give you an idea of what is being explored and what might become part of Java in a future release.

G1 GC: Improve Throughput by Reducing Synchronization (JEP 522)

The Garbage-First collector (G1) is the default garbage collector, so its performance is critical. Ivan Walulya and Thomas Schatzl improved G1 by reducing the synchronization required between application and GC threads. This significantly improves the throughput and latency.

When you move to the new JVM, your throughput improves by 5-15% without you doing anything. This is the best kind of change. Of course, this is aided by the JVM developers not having to chase the newest cool language feature: they would rather spend their time improving the JVM implementation.

But be aware that the new changes might introduce bugs; use `-XX:-G1UseConcRefinement` to disable them.

HTTP/3 in the Standard HTTP Client (JEP 517)

Java 26 adds support for HTTP/3 to the standard *java.net.http* HTTP Client API. This allows Java applications to communicate with HTTP/3 servers using the same client that was introduced in Java 11, without requiring a new API or a rewrite. Support for HTTP/3 is explicitly opt-in and does not change the default protocol behavior.

Why HTTP/3 Matters Today?

HTTP/3 is the successor to HTTP/2 and is built on top of the QUIC transport protocol rather than TCP. QUIC is a modern transport protocol designed for web traffic and runs over UDP, with TLS 1.3 integrated by default. This design reduces connection setup time, avoids head-of-line blocking, and improves performance on unreliable or high-latency networks. HTTP/3 is already widely used by modern browsers and supported by a large portion of public web infrastructure, making it increasingly relevant for client-side applications.

What Does It Mean That HTTP/3 Lives in JDK 26?

By including HTTP/3 support in the standard HTTP Client API, Java removes the need for third-party networking libraries to access modern web protocols. Existing application code remains essentially unchanged, and developers can opt in to HTTP/3 on a per-client or per-request basis. If a server does not support HTTP/3, the client transparently falls back to HTTP/2 or HTTP/1.1. This approach preserves backward compatibility while allowing applications to adopt newer protocols at their own pace.

Do You Need HTTP/3?

Honestly, you'll be fine without it. As most of the time, HTTP isn't the bottleneck (and HTTP/2 is good enough). It's nice that Java keeps pace with the new protocols, but you'll probably only use it once your server frameworks adopt them or you really know what you're doing.

PEM Encodings of Cryptographic Objects (JEP 524)

Java 26 continues its focus on usability and security with a preview API for working with PEM (Privacy-Enhanced Mail) formatted cryptographic data.

PEM is a text-based format defined by RFC 7468 and is widely used for exchanging public keys, private keys, certificates, and certificate revocation lists. It's the format you see everywhere in practice from TLS certificates to OpenSSH keys and hardware security devices.

Until now, the Java Platform lacked a straightforward way to encode and decode PEM. While Java's cryptography APIs can already produce and consume binary formats like PKCS#8 and X.509, developers were left to manually handle Base64 encoding, PEM headers and footers, encryption, and provider selection.

Why It Matters

This API significantly reduces boilerplate and the risk of security mistakes when working with PEM files. Tasks that once took dozens of lines of careful, error-prone code can now be done in a few clear method calls, all using standard Java APIs.

As a preview feature, it must be explicitly enabled in Java 26, and its design may evolve. Still, it's a strong signal that Java is closing long-standing gaps in its cryptography tooling.

The End of Java Applets

Java applets were once the way you implemented interactive web applications if you didn't want to use Flash. As an example:

```
1  import java.applet.Applet;
2  import java.awt.Button;
3  import java.awt.Graphics;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6
7  public class HelloApplet extends Applet {
8      private String message = "Hello from an Applet";
9      private int clicks = 0;
10     @Override
11     public void init() {
12         Button button = new Button("Click me");
13         button.addActionListener(new ActionListener() {
14             @Override
15             public void actionPerformed(ActionEvent e) {
16                 clicks++;
17                 message = "Clicks: " + clicks;
18                 repaint(); // trigger redraw
19             }
20         });
21         add(button);
22     }
23     @Override
24     public void paint(Graphics g) {
25         g.drawString(message, 20, 40);
26     }
27 }
```

You could embed this via:

```
1 <!doctype html>
2 <html>
3 <body>
4 <applet code="HelloApplet.class" archive="hello-applet.jar" width="300" height="120">
5     Your browser does not support Java applets.
6 </applet>
7 </body>
8 </html>
```

Java 26 removes the Applet API. This is not a breaking change in practice, but the final step of a very long deprecation process.

Applets have been obsolete for years. Web browsers stopped supporting them long ago; the appletviewer tool was removed in Java 11. And the API itself was deprecated for removal in Java 17. With the Security Manager now permanently disabled, there is no longer any technical foundation to run applets safely.

As a result, Java 26 removes the `java.applet` package. Along with related classes such as `JApplet`. For most developers, this has zero impact. Code that was still dependent on applets was already tied to older Java versions. And modern alternatives have existed for a long, long time.

This change is a good example of Java's careful evolution: once a core feature, the Java platform phased it out gradually with years of notice, clear migration paths, and a quiet removal when the API lost its relevance.

Preview and Incubating Features

Java 26 includes several preview and incubating features, each described in a JEP (Java Enhancement Proposal). A JEP is a design document that explains a proposed change to the Java platform, including its motivation, scope, and current status.

Preview and incubating features are disabled by default, and developers must explicitly enable them using preview feature flags at compile time and runtime. They exist to gather feedback and allow features to mature safely before becoming part of the standard language or API. Don't use them in production; they might disappear in the next Java version.

To try them out, you must explicitly enable previews, for example:

```
1 javac --enable-preview --release 26 PrimitivePatternExample.java
2 java --enable-preview PrimitivePatternExample
```

Primitive Patterns (JEP 530)

Primitive patterns extend pattern matching so that it works consistently with primitive types in `instanceof` and `switch`. This allows Java to safely check and bind primitive values without manual casts or error-prone range checks. The compiler can now automatically reject unsafe conversions. This feature is still in preview, and while it improves language consistency, most production code does not yet rely on advanced pattern matching with primitives.

Eventually, you'll be able to use primitive types like any other type in the pattern-matching expressions:

```
1 switch (value) {
2     case int i -> "integer: " + i;
3 }
```

Lazy constants introduce a way to define immutable values that the JVM initializes only when first used, instead of eagerly at startup. They combine the safety and performance benefits of final fields with the flexibility of lazy initialization, including correct behavior in multi-threaded code. This API helps optimize startup time, but as a preview feature, it is intended primarily for experimentation rather than widespread production use.

This also makes your code far more readable, as you don't have to write your own caching logic. Now you can write the following to lazily, for example create a database connection:

```
1 private final LazyConstant<Connection> connection
2     = LazyConstant.of(() -> {
3         try {
4             return DriverManager.getConnection(
5                 "jdbc:postgresql://localhost:5432/appdb",
6                 "appuser", "apppass");
7         } catch (SQLException e) {
8             throw new IllegalStateException("Failed to create DB connection", e);
9         }
10    });
```

Vector API (JEP 529)

The Vector API provides an explicit way to write vectorized computations that map reliably to modern CPU SIMD instructions. It is designed for performance-critical workloads such as numerical processing, cryptography, or machine learning. While powerful, it remains an incubating API and is deliberately niche. Most Java applications do not need explicit vectorization, and auto-vectorization already covers many common cases. It's mostly aimed at libraries like Lucene or AI.

Structured Concurrency (JEP 525)

Structured concurrency groups related concurrent tasks into a single unit of work, making error handling, cancellation, and observability more predictable. It aligns concurrent code structure with normal block structure, reducing thread leaks and subtle bugs. Although promising, this API is still in preview, and many production systems continue to rely on established concurrency abstractions while it matures. Good APIs take time.

You Don't Need This (Yet)

All of these features are previews for a reason. They allow developers to explore new ideas, provide feedback, and shape the platform's future without forcing adoption. Java deliberately keeps preview features opt-in, so production systems remain stable by default. You can safely ignore these features today and still benefit from Java 26, and that is very much by design.

The same goes for the non-Long-Term-Support releases: with versions like Java 26, the JVM developers give you access to new language features, whether they're final or preview. So you don't have to spend so much time waiting, like when switching from Java 7 to Java 8.

But still: Please try out these features if you want and give the JVM developers feedback. Java is not developed in a separate reality, it's developed in conjunction with the changing needs of its users.

Conclusion

Well, Java 26 is boring, except for the JVM. So upgrading to the next JVM might be great, even if the language didn't change. The great thing about Java is that you can write Java 8 code and nobody will notice. You can focus on your actual application without having to follow all the language news like in Scala or Rust.

Evolution over revolution is what kept Java alive for 30 years. Careful language design and runtime development will hopefully keep it relevant for far longer.

Can Java be improved? Are some existing features not the best? Does it miss features like string interpolation that other languages already have? Yes to all.

But also: Has it been around far longer than its competitors? Is it a vibrant community? Is the implementation of the runtime managed by more than one company? Does it run almost everywhere? Also yes to all.

It has never been a better time to be a Java developer, see you in another 30 years!

Vector Search with Pure Java

No External Database. Extremely fast. No Compromises.

NEW:

EclipseStore 4 - Java Embedded Vector Database

EclipseStore 4 integrates JVector, turning your app into a Java-native vector database. It eliminates 'Mapping Hell' and the need to manage external database processes - you get elite vector search performance with zero external dependencies.

EclipseStore 4 is just a tiny Java library available on Maven Central. The project is open-source under the Eclipse Public License (EPL 2.0) which allows you to use EclipseStore 4 for any commercial purposes. Check it out and if you like it, please star us on GitHub.

Project: www.eclipsestore.io

Code on GitHub: github.com/eclipse-store/store

NEW:

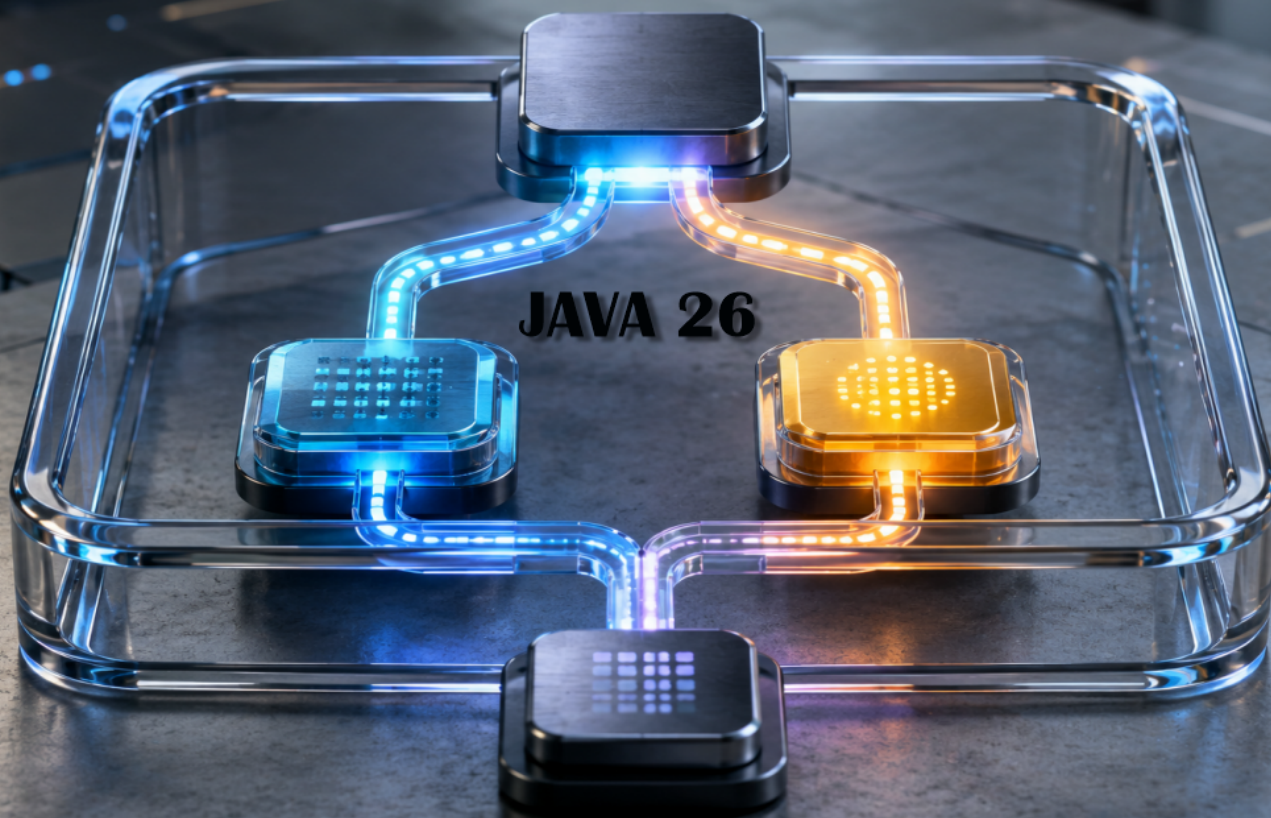
Eclipse Data Grid 4 - Distributed In-Memory Vector Search

Eclipse Data Grid is a Software-as-Code project that automates complex cluster orchestration, giving you a production-ready environment on your own terms. Built for on-premises and private cloud deployments, it enables you to run EclipseStore 4 as a distributed, in-memory vector search grid with full data sovereignty and zero reliance on public cloud providers.

Air-Gapped Ready: Standardized deployment logic for high-security, disconnected environments.

The project is open-source (EPL 2.0) - check it out on GitHub to bridge the gap between Java code and your private infrastructure, and please give us a star!

Code on GitHub: github.com/eclipse-datagrid/datagrid

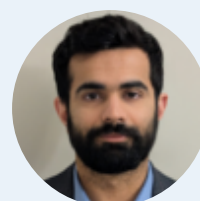


Structured Concurrency in Java 26 (JEP 525 Deep Dive)

CoreJava # Java26 # StructuredConcurrency

Author: Babneet Singh

Babneet Singh is a software engineer working on runtime components in IBM Semeru Runtimes, which is based on the Eclipse OpenJ9 and OMR projects. He is a committer on both open-source projects and contributes to JVM runtime development with a focus on correctness, maintainability, and long-term sustainability. He collaborates with engineers across the JVM stack, supports low-level runtime work, and participates in open-source discussions on JVM architecture and design. His interests include JVM internals, runtime concurrency, and building reliable virtual machines for modern Java workloads.



LinkedIn

The Java 26 Evolution of Safer, Predictable Multi-threading

Java has long provided powerful concurrency primitives. Threads, executors, futures, and more recently virtual threads allow applications to scale across cores and I/O boundaries. Yet many production failures are not caused by insufficient parallelism, but by insufficient structure.

Tasks outlive the operations that spawned them. Failures are detected but not propagated consistently. Interrupts are delivered but not observed in time. Thread dumps reveal activity, but not ownership or intent.

With the sixth preview of Structured Concurrency in Java 26 (JEP 525), the platform moves closer to a model where concurrency is not merely available, but constrained. The API refinements are incremental. The semantic shift is not. Lifetime, failure, cancellation, visibility, and completion are now encoded into program structure and enforced by the runtime.

Concurrency Without Structure: Ambiguity at Runtime

Traditional executor-based concurrency separates task submission from task ownership. A pool schedules work. Futures represent results. Cancellation is cooperative and often optional.

From a runtime perspective, this introduces ambiguity:

- A task may continue executing after its parent operation has returned.
- Failure in one subtask does not automatically affect siblings.
- Interrupts may be delivered but not observed promptly.
- Cleanup logic must avoid subtle races.

Virtual threads remove the cost barrier to spawning threads, but they do not impose discipline. In fact, cheaper threads make unbounded fan-out easier. Without structure, abundance increases risk.

Structured concurrency addresses this directly.

Lexical Scope as a Lifetime Boundary

Structured concurrency restores a property sequential code has always had: containment.

If a block of code forks subtasks, those subtasks must complete before control leaves that block. They cannot outlive their lexical scope, and they cannot be joined elsewhere.

This is not just a coding style. It is a runtime contract. When the scope closes, the runtime ensures teardown is deterministic: remaining subtasks are interrupted, and the owner thread does not continue until the scope has fully finished cleaning up.

Structural violations fail deterministically. Concurrency becomes hierarchical rather than ambient. The runtime can reason about execution as a tree rooted in lexical structure.

What Changed in the Sixth Preview (JEP 525)

JEP 525 focuses on refinement and stabilization.

1. `allSuccessfulOrThrow()` Returns `List<T>`

In Java 26, `Joiner.allSuccessfulOrThrow()` returns a materialized `List<T>` rather than a stream of subtask handles. The list is in the order the subtasks were forked. This reduces boilerplate and reinforces a simple pattern: join first, then consume results.

2. `anySuccessfulOrThrow()` Renaming

`anySuccessfulResultOrThrow()` was shortened to `anySuccessfulOrThrow()`, clarifying intent and aligning naming across joiners.

3. Tighter Configuration API

`StructuredTaskScope.open()` now accepts a `UnaryOperator<Configuration>` rather than a generic `Function`. Configuration is constrained to mutation of the provided configuration object, preventing unintended type mixing.

4. The `onTimeout()` Hook

Timeouts are no longer limited to immediate exceptions. The `Joiner` interface now includes:

```
1 void onTimeout()
```

The default implementation throws `StructuredTaskScope.TimeoutException`, which makes `join()` fail fast on timeout as before. The key change is that custom joiners can override `onTimeout()` and choose not to throw. If `onTimeout()` does not throw, then `join()` proceeds to call `result()`, allowing the joiner to return a partial or fallback result based on whatever completed before the timeout.

Structured Concurrency in Practice: An E-Commerce Product Page

Consider an e-commerce product page that must fetch:

- Real-time inventory
- Dynamic pricing

Both services are independent and can run concurrently. However, the page must respond within 800 milliseconds.

Using Built-In Joiners (Java 26)

In the example below, the subtasks return different types, so the correct built-in joiner is `awaitAllSuccessfulOrThrow()`. It enforces that all subtasks must complete successfully (or the scope fails), while results are retrieved from strongly typed subtask handles.

```
1 public ProductDetails fetchProductDetails(String productId)
2     throws InterruptedException {
3
4     try (var scope = StructuredTaskScope.open(
5         StructuredTaskScope.Joiner.awaitAllSuccessfulOrThrow(),
6         cfg -> cfg.withTimeout(Duration.ofMillis(800)))) {
7
8         Subtask<Inventory> inventoryTask =
9             scope.fork(() -> checkInventory(productId));
10
11         Subtask<Pricing> pricingTask =
12             scope.fork(() -> fetchDynamicPricing(productId));
13
14         scope.join(); // throws if failure or timeout
15
16         return new ProductDetails(
17             inventoryTask.get(),
18             pricingTask.get());
19     }
20 }
```

Several Java 26 refinements are visible:

- The configuration lambda is a `UnaryOperator<Configuration>`.
- Timeout is applied at the scope level.
- Completion policy is centralized.
- No positional indexing or unsafe casting is required.

If a subtask fails, or if the deadline is exceeded, the scope interrupts remaining subtasks and `join()` fails. No manual cancellation logic is needed.

When all subtasks return the same type, `allSuccessfulOrThrow()` is the simplest option, because `join()` can return a `List<T>` of results directly.

Graceful Degradation with `onTimeout()`

Now consider a different policy.

If dynamic pricing times out under heavy load, we still want to display inventory and fall back to a base price rather than fail the page entirely. This is the scenario that the Java 26 `onTimeout()` hook makes practical.

The important nuance is that timeouts still cancel the scope. The difference is what happens next:

- Default behavior: `onTimeout()` throws, so `join()` throws `TimeoutException`.
- Custom behavior: override `onTimeout()` and do not throw, so `join()` can call `result()` and return a degraded response.

Here is a custom joiner that requires inventory, but allows pricing to fall back:

```

1  final class ProductJoiner
2      implements StructuredTaskScope.Joiner<ProductData, ProductDetails> {
3
4      private final AtomicReference<Inventory> inventory =
5          new AtomicReference<>();
6
7      private final AtomicReference<Pricing> pricing =
8          new AtomicReference<>(Pricing.basePrice());
9
10     @Override
11     public boolean onComplete(Subtask<ProductData> subtask) {
12         if (subtask.state() == Subtask.State.SUCCESS) {
13             ProductData result = subtask.get();
14
15             if (result instanceof Inventory inv) {
16                 inventory.compareAndSet(null, inv);
17             } else if (result instanceof Pricing prc) {
18                 pricing.set(prc);
19             }
20         }
21         return false; // continue waiting for remaining subtasks
22     }
23
24     @Override
25     public void onTimeout() {
26         // Do not throw.
27         // Timeout policy:
28         //   - keep fallback pricing, and
29         //   - return partial result if possible.
30     }
31
32     @Override
33     public ProductDetails result() {
34         Inventory inv = inventory.get();
35         Pricing prc = pricing.get();
36
37         if (inv == null) {
38             throw new IllegalStateException("Inventory unavailable");
39         }
40
41         return new ProductDetails(inv, prc);
42     }
43 }

```

Used as:

```

1  try (var scope = StructuredTaskScope.open(
2      new ProductJoiner(),
3      cfg -> cfg.withTimeout(Duration.ofMillis(500)))) {
4
5      scope.fork(() -> checkInventory(productId));
6      scope.fork(() -> fetchDynamicPricing(productId));
7
8      return scope.join();
9  }

```

If pricing exceeds the timeout:

- The scope is cancelled and remaining subtasks are interrupted.
- `onTimeout()` is invoked.
- Because `onTimeout()` does not throw, `join()` calls `result()`.
- `result()` returns a degraded but valid response (inventory plus fallback price).

Timeout becomes policy rather than an automatic exception.

Joiners as Explicit State Machines

Joiners centralize completion logic. They evaluate:

- Success
- Failure
- Timeout
- Cancellation

Rather than racing independent futures, completion becomes a controlled state transition. This improves predictability and reasoning at both the application and runtime levels.

A practical guideline: keep joiners general-purpose and mechanical. Collect outcomes, decide when to cancel, and produce a result. Keep business logic at the use-site.

Also note that joiners must be thread-safe. Subtasks can complete concurrently, so `onComplete` may be invoked in parallel. The use of `AtomicReference` above is not decoration, it is required for correctness.

Interruption and Deterministic Teardown

When a scope resolves, remaining subtasks are interrupted. Scope closure blocks until they terminate.

This guarantees:

- No subtask survives past lexical exit.
- No late results are observed after failure.
- No resource leaks remain hidden.

Structured concurrency aligns computation lifetime with the lifetime of the initiating request.

Memory Visibility and Happens-Before

Structured concurrency is not only about lifetime. It also makes visibility easier to reason about.

At a high level:

- Work done by a subtask becomes safely observable when the owner thread obtains the outcome (via `Subtask.get()` or via a successful `join()` that incorporates that outcome).
- The scope boundary becomes both a lifetime boundary and a visibility boundary.

This matters in practice because it reduces accidental reliance on timing. You can treat `join()` and `get()` as the points where “this result is now safe to read”, without inventing additional synchronization.

Virtual Threads and Bounded Abundance

Virtual threads make thread-per-task scalable. Blocking no longer monopolizes kernel threads.

Structured scopes ensure that abundance remains bounded. When a scope resolves, all associated virtual threads terminate. Scalability and predictability reinforce each other.

Observability Through Hierarchy

Structured concurrency encodes parent-child relationships directly.

Diagnostics improve because execution mirrors code structure. Failures, waits, and cancellations map cleanly to lexical boundaries. Observability improves not through extra tooling, but through structural clarity.

When Not to Use Structured Concurrency

Structured concurrency is designed for bounded operations. Long-lived schedulers, streaming pipelines, or compute-heavy frameworks may require different coordination models.

It excels where there is a single initiating thread and a finite unit of work.

Conclusion: Concurrency with Enforced Boundaries

The sixth preview of Structured Concurrency in Java 26 refines and strengthens the model.

- Completion policies are explicit.
- Timeout handling is customizable.
- Configuration is constrained.
- Results are strongly typed.
- Lifetimes are lexically bounded.
- Visibility guarantees are preserved.

Concurrency in modern Java is no longer merely about parallel execution. It is about enforcing correctness through structure, ensuring work begins, resolves, and terminates within boundaries the runtime understands and enforces.

Structured concurrency is not just easier concurrency.

It is more predictable concurrency, by design.

› [Back to the Table of Contents](#) ‹

APR
26-29



www.jcon.one

JCON2027

JCON EUROPE 2027
International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



Are Classic Design Patterns Still Valid With Modern Java?

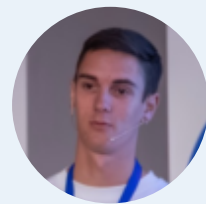
CoreJava

DesignPatterns

ModernJava

Author: Andreas Jürgensen

Andreas is a software architect with several years of experience as a software developer. His particular focus is on automated testing, which he understands as an essential component of sustainable and maintainable software architectures. He is the founder and CEO of FOURTEEN IT, where he supports companies and development teams as a coach and consultant in the design and implementation of modern software systems.



LinkedIn

The classic GoF design patterns helped developers build maintainable object-oriented software for a long time. In the past years the languages, like Java, have changed a lot; they evolved! What was a design pattern back then became a language feature today. This article analyses which patterns are still needed and which became obsolete.

Once upon a time...

Object-Oriented Programming

Object-oriented programming emerged with the invention of Simula 67. Ole-Johan Dahl and Kristen Nygaard invented Simula 67 while working at the Norwegian Computing Center in Oslo to run simulations. A few years later Alan Kay et al. invented Smalltalk as a purely object-oriented programming language. In the 80s, C++ brought the breakthrough in the mainstream for object-oriented programming. The first large-scale object-oriented systems were developed. In the 90s, object-oriented programming was the dominating programming paradigm.

GoF Design Patterns

In 1994 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, often referred to as the *Gang of Four* (GoF), published “Design Patterns: Elements of Reusable Object-Oriented Software”. The 23 design patterns described in this book became very popular and shaped the common interpretation of object-oriented design. Over the years they helped loads of developers to build maintainable software.

Java

In the mid-90s something else happened: the first version of Java was released in 1995. Java was planned to be a simple yet robust object-oriented programming language which is similar to C++. Java is class-based which means that the behavior of an object is described via its class. Each object can have its own state but all objects of the same class behave in the same way.

Java allows inheritance to create hierarchies and extend or override the behavior of a superclass. To loosely couple the method signature and its implementation(s) Java offers the concept of interfaces. An interface defines method signatures without method bodies. All implementing classes of an interface have to provide an implementation for each defined method signature. Additionally, it is possible to define static methods, variables and constants to define class-level behavior and state. With all these language features combined it is possible to implement every GoF design pattern.

30 Years Later

Over the past 30 years a lot has changed: other programming paradigms became popular, new frameworks and libraries arose and Java evolved by adding some new language features. Therefore it's time to have a look at the GoF design patterns and evaluate if they are still needed or have become obsolete.

The GoF Design Patterns

The 23 GoF design patterns focus on different aspects of object-oriented design. To make them easier to understand and apply, they are commonly grouped into three categories: creational patterns, structural patterns, and behavioral patterns. This classification helps in identifying a suitable pattern for a given problem.

As the name suggests, creational patterns deal with different ways of creating objects. Structural patterns focus on how classes and objects are composed to form larger structures. Behavioral patterns describe how objects interact and how responsibilities are distributed among them, often allowing behavior to vary at runtime.

Since the GoF design patterns are commonly known, they won't be described in this article. For detailed information, consult the design patterns book by Gamma et al., visit refactoring.guru/design-patterns, or similar resources.

The Evolution of Java

Java has evolved steadily over the years, balancing backward compatibility with modern language features. A major turning point came with Java 8, which introduced Lambdas and Method References. These features enabled a more functional programming style, allowing behavior to be passed as data and significantly reducing boilerplate code.

Closely related, the Stream API also arrived in Java 8. It provides a powerful and expressive way to process collections declaratively, with built-in support for parallelism. In Java 24 the Stream API has been extended to provide more flexibility. The Gatherer interface allows a wide range of intermediate operations to be performed during stream processing.

New releases continued to focus on improving expressiveness and maintainability. Records were introduced in Java 16. They offer a concise way to model immutable data carriers, making intent explicit and reducing the need for verbose boilerplate such as constructors, getters, and equals/hashCode. With Java 17, Sealed Interfaces further strengthened the type system by allowing developers to precisely control which classes or interfaces may implement or extend them. This enables more robust domain modeling and better support for exhaustive checks. Records and sealed interfaces work great in combination with pattern matching, which was also introduced in Java 17.

Together, these features reflect Java's evolution from a purely object-oriented language toward multi-paradigmatism focused on clarity, safety, and long-term maintainability. This path continued in the latest releases. The newest features don't have such a big impact on the relevance of design patterns, therefore they won't be looked at in this article.

The Impact of Lambdas, Method References and Functional Interfaces

Lambdas, method references and functional interfaces, have significantly influenced how several classic design patterns are applied, without making them obsolete.

Factory Method

The Factory Method pattern is used to separate object creation from its usage by encapsulating the creation logic in a dedicated factory method. With lambdas and method references, object creation can now be expressed concisely using constructor references or lambdas passed directly to the client. Therefore explicit factory classes are often unnecessary.

Command Pattern

The Command pattern encapsulates a request as an object, enabling requests to be parameterized, queued, or logged, and also supporting undo/redo functionality. In modern Java, simple commands can often be implemented as a Runnable, Consumer, or other functional interface, dramatically reducing boilerplate and making them easier to compose. This does not work with commands that have more than one method, like an additional undo method.

Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically. Lambdas allow observers or listeners to be expressed concisely as inline callbacks, avoiding the need for full observer classes.

Strategy Pattern

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable so that the algorithm can vary independently from the client. With functional interfaces, strategies can now often be passed directly as lambdas, simplifying the code and improving readability. This is only recommended for simple strategies fitting in a simple lambda expression. Complex strategies, especially with further dependencies, should still be implemented as classes.

Adapter Pattern

The Adapter pattern allows incompatible interfaces to work together by translating one interface into another. It is still relevant when integrating existing or third-party code, but one particular scenario has changed: if a static method exists but an object is wanted, there is no longer a need to implement an adapter object. Instead, a method reference to the static method can serve as the adapter object.

In contrast to the Factory Method, Command, Observer and Strategy pattern, the implications for the Adapter pattern may not be well known and may need an example:

The class `java.time.LocalDateTime` provides a static method `now` that returns the current local date and time. To unit-test a class that uses this method it can be necessary to stub the returned value. Because static methods are difficult to replace with test doubles and static mocking is often unwanted an adapter can be used to provide a non-static function that returns the current local date and time:

```
1 <code>interface LocalDateTimeAdapter {<br>    LocalDateTime now();<br>}</code>
```

```
1 class LocalDateTimeAdapterImpl implements LocalDateTimeAdapter { @Override
2     public LocalDateTime now() {
3         return LocalDateTime.<em>now</em>();
4     }
5 }
```

The adapter can now be injected into the constructor of another class, for example a controller:

```
1 class Controller {
2     private final AService aService;
3     private final LocalDateTimeAdapter localDateTimeAdapter;
4
5     Controller(AService aService) {
6         this(aService, new LocalDateTimeAdapterImpl());
7     }
8
9     Controller(AService aService, LocalDateTimeAdapter localDateTimeAdapter) {
10        this.aService = aService;
11        this.localDateTimeAdapter = localDateTimeAdapter;
12    }
13
14    // ...
15 }
```

Instead of creating a new interface, the interface `Supplier<T>` from `java.util.function` can be used. And instead of implementing the interface in an adapter class a method reference directly to the static method can be used in the secondary constructor of the controller:

```
1 import java.util.function.Supplier;
2
3 class Controller {
4     private final AService aService;
5     private final Supplier<LocalDateTime> now;
6
7     Controller(AService aService) {
8         this(aService, LocalDateTime::now);
9     }
10
11    Controller(AService aService) {
12        this.aService = aService;
13        this.now = now;
14    }
15
16    // ...
17 }
```

The Impact of Streams

The Iterator pattern provides a way to access elements of a collection sequentially without exposing its internal representation. In modern Java, the Stream API largely replaces explicit iteration. Streams allow developers to declaratively express what should be done with elements, such as filtering, mapping, or transforming, without worrying about the mechanics of traversal.

Collectors are extension points for terminal operations, which mark the end of a stream and aggregate elements into a list, map, sum, or other results. They replace many manual iterations, such as building a list or summing values.

Introduced in Java 24, Stream Gatherers differ from collectors in that they are designed for intermediate operations. Gatherers allow intermediate results to be collected or transformed during processing before the stream reaches its final aggregation. Together with collectors, gatherers extend the Stream API to handle both intermediate and terminal processing declaratively, effectively absorbing much of the traditional iterator logic into the stream abstraction.

The Iterator pattern could be used to iterate a list of integers while calculating the running sum for each iteration:

```
1 record NumberAndRunningSum(int number, int runningSum) { }

1 class RunningSumIterator implements Iterator<NumberAndRunningSum> {
2     private final List<Integer> numbers;
3     private int index = 0;
4     private int runningSum = 0;
5
6     RunningSumIterator(List<Integer> numbers) {
7         this.numbers = numbers;
8     }
9
10    @Override
11    public boolean hasNext() {
12        return index < numbers.size();
13    }
14
15    @Override
16    public NumberAndRunningSum next() {
17        if(!hasNext()) {
18            throw new NoSuchElementException();
19        }
20        int number = numbers.get(index++);
21        runningSum += number;
22        return new NumberAndRunningSum(number, runningSum);
23    }
24 }
```

This Iterator can be used to iterate a list of integers and print out every number and the corresponding running sum:

```
1 void main() {
2     List<Integer> numbers = List.of(1, 2, 3, 4, 5);
3     RunningSumIterator iterator = new RunningSumIterator(numbers);
4     while (iterator.hasNext()) {
5         System.out.println(iterator.next());
6     }
}
```

The same logic can be expressed using a stream gatherer:

```
1 void main() {
2     var runningSum = Gatherer.ofSequential(
3         () -> new int[]{0},
4         (state, number, downstream) -> {
5             state[0] += number;
6             downstream.push(new NumberAndRunningSum(number, state[0]));
7             return true; // continue
8         }
9     );
10    Stream.of(1, 2, 3, 4, 5)
11        .gather(runningSum)
12        .forEach(System.out::println);
13 }
```

The Impact of Records and Sealed Interfaces

The Visitor pattern separates an algorithm from the object structure it operates on. It allows new operations to be added without modifying the existing object classes, at the cost of making the object structure harder to change.

An example of using the Visitor pattern is to search in a binary tree. The binary tree can be implemented like this:

```

1  interface Node<K, V> {
2
3      void accept(NodeVisitor<K, V> visitor);
4
5      K key();
6
7      V value();
8  }

```

```

1  class BinaryTree<K, V> implements Node<K, V> {
2      private final K key;
3      private final V value;
4      private final Node<K, V> left;
5      private final Node<K, V> right;
6
7      BinaryTree(K key, V value, Node<K, V> left, Node<K, V> right) {
8          this.key = key;
9          this.value = value;
10         this.left = left;
11         this.right = right;
12     }
13
14     @Override
15     public void accept(NodeVisitor<K, V> visitor) {
16         visitor.visit(this);
17     }
18
19     @Override
20     public K key() { return key; }
21
22     @Override
23     public V value() { return value; }
24     Node<K, V> left() { return left; }
25     Node<K, V> right() { return right; }
26 }

```

```

1  class Leaf<K, V> implements Node<K, V> {
2      private final K key;
3      private final V value;
4
5      Leaf(K key, V value) {
6          this.key = key;
7          this.value = value;
8      }
9
10     @Override
11     public void accept(NodeVisitor<K, V> visitor) {
12         visitor.visit(this);
13     }
14
15     @Override
16     public K key() { return key; }
17
18     @Override
19     public V value() { return value; }
20 }

```

```

1  class EmptyNode<K, V> implements Node<K, V> {
2
3      @Override
4      public void accept(NodeVisitor<K, V> visitor) {
5          visitor.visit(this);
6      }
7      @Override
8      public K key() {
9          throw new UnsupportedOperationException();
10     }
11     @Override
12     public V value() {
13         throw new UnsupportedOperationException();
14     }
15 }

```

To search the binary tree a visitor is needed:

```

1  interface NodeVisitor<K, V> {
2      void visit(BinaryTree<K, V> binaryTree);
3      void visit(Leaf<K, V> leaf);
4      void visit(EmptyNode<K, V> emptyNode);
5      Optional<V> result();
6  }

```

```

1  class SearchInDictionary implements NodeVisitor<String, String> {
2      private final String searched;
3      private String result;
4
5      SearchInDictionary(String searched) {
6          this.searched = searched;
7      }
8
9      @Override
10     public void visit(BinaryTree<String, String> binaryTree) {
11         int comparisonResult = searched.compareTo(binaryTree.key());
12         if (comparisonResult == 0) {
13             result = binaryTree.value();
14         } else if (comparisonResult < 0) {
15             binaryTree.left().accept(this);
16         } else {
17             binaryTree.right().accept(this);
18         }
19     }
20
21     @Override
22     public void visit(Leaf<String, String> leaf) {
23         if (searched.equals(leaf.key())) {
24             result = leaf.value();
25         }
26     }
27
28     @Override
29     public void visit(EmptyNode<String, String> emptyNode) { }
30
31     @Override
32     public Optional<String> result() {
33         return Optional.ofNullable(result);
34     }
35 }

```

Since every visitor needs to implement a visit-method for every implementation of Node, every visitor must be edited if the data structure changes. In modern Java records can be used to represent simple data structures like this:

```
1 record Leaf<K, V> (K key, V value) implements Node<K, V> { }
```

```
1 record BinaryTree<K, V> (
2     K key, V value,
3     Node<K, V> left, Node<K, V> right
4 ) implements Node<K, V> { }
```

```
1 class EmptyNode<K, V> implements Node<K, V> {
2     @Override
3     public K key() { throw new UnsupportedOperationException(); }
4     @Override
5     public V value() { throw new UnsupportedOperationException(); }
6 }
```

In order to make sure that these records are the only implementations of Node, the interface can be sealed:

```
1 sealed interface Node<K, V> permits Leaf, BinaryTree, EmptyNode {
2     K key(); V value();
3 }
```

The binary tree can be iterated using switch expressions and pattern matching:

```
1 class SearchInDictionary {
2     Optional<String> result(
3         Node<String, String> dictionary,
4         String searched
5     ) {
6         return switch(dictionary) {
7             case BinaryTree<String, String> binaryTree ->
8                 resultFromBinaryTree(binaryTree, searched);
9             case Leaf<String, String> leaf -> resultFromLeaf(leaf, searched);
10            case EmptyNode<String, String> _ -> Optional.empty();
11        };
12    }
13
14    private Optional<String> resultFromBinaryTree(
15        BinaryTree<String, String> binaryTree,
16        String searched
17    ) {
18        int comparisonResult = searched.compareTo(binaryTree.key());
19        if (comparisonResult == 0) {
20            return Optional.of(binaryTree.value());
21        } else if (comparisonResult < 0) {
22            return result(binaryTree.left(), searched);
23        } else {
24            return result(binaryTree.right(), searched);
25        }
26    }
27
28    private Optional<String> resultFromLeaf(
29        Leaf<String, String> leaf,
30        String searched
31    ) {
32        return searched.equals(leaf.key()) ?
33            Optional.of(leaf.value()) :
34            Optional.empty();
35    }
36 }
```

Because the interface is sealed, there is no need for a default-case in the switch-expression. The compiler assures that every possible case is covered. This adds some extra safety net compared to the classic Visitor pattern.

Conclusion

The analysis shows that the classic GoF design patterns are neither obsolete nor universally applicable in their original form when working with modern Java. Instead, their role has shifted. Many patterns emerged as structured solutions to limitations of early object-oriented languages. With the evolution of Java, especially through lambdas, functional interfaces, streams, records, sealed interfaces, and pattern matching, some of these limitations have been addressed directly at the language level.

Modern Java strongly supports functional programming concepts. As a consequence, patterns that primarily exist to separate behavior from data, such as Strategy, Command, Visitor, or parts of Factory Method, can often be expressed more directly and with far less boilerplate. This does not mean these patterns are “wrong” today, but rather that their intent is frequently better served by language features instead of explicit pattern implementations. In idiomatic modern Java, overusing classic object-heavy pattern structures can even work against clarity and simplicity.

Lookout

This observation aligns with earlier research. Peter Norvig showed that languages like Lisp or Dylan already provided features that simplified or eliminated the need for a large portion of the GoF design patterns. Similarly, Hannemann and Kiczales demonstrated that aspect-oriented programming could simplify or replace many patterns by modularizing cross-cutting concerns. Java has not adopted AOP at the language level, but its steady move toward multi-paradigm programming confirms the same general direction: patterns migrate into the language over time.

Looking ahead, future language features could further change the relevance of classic patterns. True first-class functions beyond today’s functional interfaces, or built-in delegation support, could make additional patterns redundant or significantly simpler. While there are currently no concrete plans in Java to introduce such features, the language’s evolution over the last decades suggests that expressiveness, safety, and reduction of boilerplate will remain key goals.

In conclusion, the GoF design patterns are still valuable. They work primarily as a vocabulary and a way of thinking about design trade-offs, not as templates to be applied mechanically. In modern Java, the question is no longer “*Which pattern should I implement?*” but rather “*Is a pattern still the best solution, or does the language already provide a simpler, clearer abstraction?*” Developers who understand both classic patterns and modern Java features are best equipped to make that decision and to design software that is not on

› [Back to the Table of Contents](#) ‹

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



The Gen AI Iceberg - Java Tooling Edition

API & Frameworks

GenAI

SpringAI

LangChain4j

Author: Artur Skowronski

Head of Java/Kotlin Engineering at VirtusLab, Artur Skowronski has been in the industry for ten years. During this time, he has had the opportunity to work in various roles, such as Software Engineer, Tech Lead, Architect, and even Technical Product Manager. This diverse experience enables him to approach problems from a holistic perspective. He still loves to get his hands dirty - for more than two years, he has been publishing weekly reviews of events in the JVM world - <https://jvm-weekly.com/>



LinkedIn

From @Autowired ChatClient to Compiling GPU Kernels From Pure Java Bytecode: A Lovecraftian Descent Through Everything a JVM Developer Needs (or Fears) to Build AI-Powered Applications

The “Iceberg” meme is an internet phenomenon that humorously, and sometimes unsettlingly, illustrates levels of knowledge or initiation into a given topic – from simple, widely known facts at the tip of the iceberg to the dark, esoteric depths comprehensible only to the most battle-hardened veterans. Picture an iceberg floating on water: what’s visible on the surface is just the beginning, while the real magic (or nightmare) lurks beneath, in increasingly inaccessible layers.

Personally, I love it. I’ve already done icebergs for the JVM ecosystem, performance, and even AI-on-Java from the platform perspective. But this time, let’s shift the lens. Instead of asking “how does the JVM *support* AI?”, let’s ask the question every Java developer is actually Googling at 11 PM: **“What tools do I actually use to build Gen AI stuff in Java?”**

The common narrative says Java developers just slap `spring-ai-openai-spring-boot-starter` in their `pom.xml` and call it a day. And for many, that’s true — and that’s fine! But under that single Maven dependency lies a sprawling, chaotic, beautiful ecosystem of tools, SDKs, inference engines, and protocols that go from “my PM approved this” all the way down to “I’m running a LLM inside my JVM process and my heap dump weighs 14 gigabytes.”

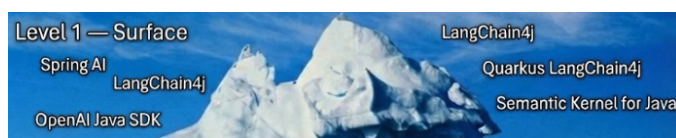
The deeper you go, the harder it is to explain what you do at stand-ups.

PS: For using the phrase “it’s just a REST call” unironically at this point, you’d be thrown overboard on this ship! Consider yourself warned.

Buckle up!



Level 1: The Tip of the Iceberg (The “@Autowired ChatClient” Zone)



Welcome to the surface! The sun is shining, the water is warm, and your biggest drama is choosing between `spring-ai-openai` and `spring-ai-anthropic` starters.

This is the world 90% of Java AI developers see - a world where AI is just another bean to inject and every second Spring blog post starts with “Building a RAG application in 5 minutes with...”

Spring AI

TLDR: Spring Magic meets LLMs. [@Autowired](#) your way into the AI revolution. If Spring Boot is your religion, this is your AI gospel.

Spring AI is the ultimate expression of “if it exists, Spring will eventually have an abstraction for it.” Launched as a proper 1.0 GA in May 2025, it brings the full power of Spring’s “convention over configuration” philosophy to the world of language models. You don’t build AI pipelines; you *inject* them. [ChatClient](#), [EmbeddingModel](#), [VectorStore](#) - they’re all just beans now, auto-configured and ready to roll.

The beauty is in the portability: switch from OpenAI to Anthropic to Ollama by changing a property, not your code. Add a [VectorStore](#) backed by PGVector, Milvus, or Pinecone with a starter dependency. Plug in document ETL, structured output mapping to POJOs (yes, your AI now returns proper Java objects), and observability via Micrometer out of the box. With version 2.0 now on the horizon (built on Spring Boot 4 and Spring Framework 7), the team is pushing into MCP support, agent orchestration, and memory management. Mark Pollack even managed to get AI-generated background music for the release notes. Peak Spring Energy.

LangChain4j

TLDR: The scrappy, LLM-native alternative for those not married to Spring. Explicit composition, Java-first idioms, and surprisingly snappy performance.

If Spring AI says “inject it,” LangChain4j says “build it.” Created by Dmytro Liubarskyi in 2023 and hitting 1.0 GA in May 2025 (the same month as Spring AI - coincidence? I think not), LangChain4j is the Java port of the LangChain philosophy, but with a twist: it actually feels like Java, not like Python code forcibly crammed into type safety.

The killer feature? AI Services - declarative interfaces where you define what you want, and LangChain4j figures out the plumbing. RAG, tool calling, memory, MCP - it’s all there, with broader provider support and vector store integrations than you can shake a [pom.xml](#) at. Microsoft and Red Hat both back the project, with hundreds of production deployments. And with the [langchain4j-agentic](#) and [langchain4j-agentic-a2a](#) modules landing in version 1.3, multi-agent orchestration became a first-class citizen.

The real kicker? Both Spring AI and LangChain4j are production-ready, and the community debates between them are surprisingly civil (by Java standards, at least).

OpenAI Java SDK

TLDR: When you just want to call the API. No frameworks, no magic - just HTTP and JSON, the way Gosling intended.

Sometimes you don’t need a framework. Sometimes you just need to call GPT-4o from your service layer, get a response, and move on with your life. The official OpenAI Java SDK (and its community counterparts) gives you exactly that: typed request/response objects, streaming support, and function calling without any framework opinions about how you should structure your application.

It’s the [URLConnection](#) of Gen AI - nobody brags about using it, but everybody’s done it at 2 AM to get a demo working. Pair it with your existing dependency injection framework (or, God forbid, static factory methods), and you’ve got AI integration without a single new annotation on your classpath.

Quarkus LangChain4j Extension

TLDR: LangChain4j, but supersonic, subatomic, and with Dev Mode. Quarkus does AI the Quarkus way.

The Quarkus team, in collaboration with the LangChain4j project, built a first-class extension that wraps LangChain4j in the Quarkus experience: CDI-based [@RegisterAiService](#), Dev UI for testing prompts in real-time, hot reload that actually works during AI development, and native compilation support for those who want their RAG pipeline to start in 50 milliseconds.

The really interesting part is the guardrails system - input/output validation, observability via OpenTelemetry, and semantic testing strategies built right in. Quarkus doesn’t just help you *build* AI apps; it helps you build *production-ready* AI apps. And

with the MCP server extension turning CDI beans into MCP-compliant tools with a single annotation, the “developer joy” factor is genuinely high.

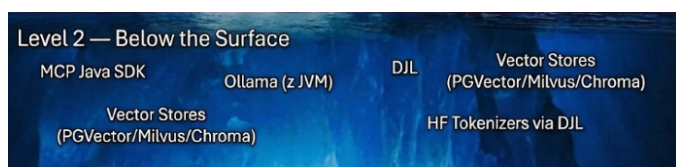
Microsoft Semantic Kernel for Java

TLDR: Microsoft’s Java AI framework. Enterprise pedigree, cross-language ambitions, slightly trailing in the Java-specific race.

Semantic Kernel, Microsoft’s entry into the AI orchestration space, supports Java alongside C# and Python. It brings concepts like Plugins, Planners, and Memory, and integrates naturally with Azure OpenAI services. The multi-agent architecture ambitions are big.

But here’s the honest truth: in the Java world, Semantic Kernel feels like it’s still warming up. The C# and Python implementations are further ahead, and the Java community has largely rallied around Spring AI and LangChain4j. That said, if you’re in an Azure-first shop, or if your team already uses Semantic Kernel in C#, the Java SDK provides a consistent cross-platform experience. Just don’t expect it to keep pace with LangChain4j’s weekly releases.

Level 2: Just Below the Surface (The “Wait, What’s Under That Starter?” Zone)



Here, you start asking questions. “But how does `ChatClient` *actually* call the model?” “What is this MCP thing everyone keeps talking about?” “My architect says we need a vector store - what even *is* that in Java terms?”

The water is getting colder. Welcome to the infrastructure layer, where you realize every framework is just orchestrating a set of building blocks you should probably understand.

MCP Java SDK (Model Context Protocol)

TLDR: The USB-C of AI tooling. One protocol to connect any LLM to any tool, and Java has the official SDK.

MCP - the Model Context Protocol - started as an Anthropic initiative, and the Spring AI team jumped on it early, eventually contributing work that became the official Java SDK. The idea is brilliant in its simplicity: standardize how AI agents discover and call external tools. No more bespoke function-calling glue code for every provider.

In practice, you define MCP servers (which expose tools, resources, and prompts) and MCP clients (which consume them). Both Spring AI and LangChain4j have native MCP support. The Java SDK supports STDIO, HTTP SSE, and the newer Streamable HTTP transports. Quarkus even has a dedicated MCP server extension that turns your CDI beans into MCP tools with annotations.

Why does this matter? Because MCP is becoming the lingua franca for AI-tool integration. Your Java backend exposing a database query as an MCP tool can be consumed by Claude, by a Spring AI agent, by a LangChain4j service, or by any MCP-compliant client. Write once, integrate everywhere - now *that’s* a Java promise worth keeping.

Ollama (from the JVM Side)

TLDR: Local LLMs without leaving localhost. Both Spring AI and LangChain4j have starters for it. Your dev environment just got an AI backend.

Ollama is the darling of the local LLM movement - `ollama run llama3` and you’ve got a model running on your machine. But for Java developers, the story is even better: both Spring AI (`spring-ai-ollama-spring-boot-starter`) and LangChain4j (`langchain4j-ollama`) have first-class integration, meaning you can develop your AI application entirely offline.

The real power move? Testcontainers. Spin up an Ollama container in your integration tests, load a small model, and actually test your RAG pipeline against a real LLM. No API keys, no rate limits, no “works on my machine” excuses (well, fewer of them). For dev/test environments in enterprises that can’t send data to external APIs, Ollama via Docker is the answer that makes compliance teams stop twitching.

Deep Java Library (DJL)

TLDR: Amazon’s engine room for AI inference in Java. Spoiler: Spring AI is basically DJL in a Spring Boot trench coat.

You peek into the transitive dependencies of your Spring AI project and find `ai.djl`. Surprise! DJL (Deep Java Library), Amazon’s open-source project, is the actual workhorse doing inference in many Java AI setups. It’s an “agnostic engine” - an abstraction layer that lets you write one codebase and swap backends between PyTorch, TensorFlow, MXNet, and ONNX Runtime.

DJL has its own Spring Boot Starter, a clean Java API for model loading and inference, and native support for Hugging Face model repositories. It handles the dirty work of tensor manipulation, model format conversion, and memory management so that higher-level frameworks don’t have to. If Spring AI and LangChain4j are the face of Java AI, DJL is the backbone nobody talks about at conferences.

Vector Stores (PGVector, Milvus, Chroma, Qdrant...)

TLDR: Where your embeddings live. The “database” in RAG, but nobody taught this in your university’s database course.

RAG without a vector store is like Spring Boot without a database — technically possible but existentially pointless. The Java AI ecosystem has converged on a surprisingly rich set of vector store integrations: PGVector (for the “just add an extension to Postgres” crowd), Milvus (for the “we need billion-scale vector search” crowd), Chroma (for the “I saw it in a Python tutorial and want the Java equivalent” crowd), and at least a dozen more.

Both Spring AI and LangChain4j provide portable abstractions with metadata filtering, so switching between stores is mostly configuration. The real gotcha? Understanding that embedding quality matters more than store choice. You can have the fanciest Milvus cluster in the world, but if your embeddings are garbage, your RAG is garbage too. GIGO transcends paradigms.

Hugging Face Tokenizers (via DJL)

TLDR: The tokenization layer nobody thinks about until their prompt mysteriously gets truncated at 4,096 tokens.

Here’s a dirty secret: every time you send text to an LLM, it gets broken into tokens by a tokenizer. The tokenizer determines how your text maps to the model’s vocabulary. Use the wrong one and your carefully crafted prompt becomes numerical nonsense.

DJL provides `HuggingFaceTokenizer` - a Java binding for the Hugging Face tokenizers library - letting you tokenize locally before sending requests, count tokens accurately, and avoid the “your prompt was too long” surprise. It’s not glamorous work, but if you’re building production RAG systems, accurate tokenization is the difference between “works in demo” and “works in production.”

Level 3: The Depths (The “My pom.xml Has An ONNX Dependency” Zone)



The descent is complete. Sunlight is gone. You start caring about “inference latency” and “model format compatibility.” You understand why running a model inside your JVM process might actually be a good idea. You’ve read the ONNX Runtime Java docs and thought “this isn’t that bad, actually.”

Here, you stop asking “Which framework should I use?” and start asking: “Can I skip the HTTP call entirely?”

ONNX Runtime Java

TLDR: The Rosetta Stone of model formats. Train in Python, run natively in Java, skip the REST middle-man.

ONNX (Open Neural Network Exchange) is the “Esperanto” of machine learning - a universal model format that lets you train in PyTorch or TensorFlow and run inference in any language that has an ONNX Runtime. Java happens to have an excellent one.

The `onnxruntime` Java library lets you load `.onnx` model files and run inference directly in your JVM process. No external servers, no HTTP latency, no serialization overhead. For use cases like text classification, toxicity detection, embedding generation, or named entity recognition, ONNX in Java is blazingly fast and terrifyingly efficient. Combined with DJL’s tokenizer support, you can build a complete inference pipeline - from raw text to model output - entirely in Java.

The workflow? Data scientist trains model in Python → exports to ONNX → you load it in Java. Clean handoff, no Python in production. This is the bridge between worlds that actually works.

Jlama

TLDR: A full LLM inference engine written in 100% pure Java. Yes, really. No JNI, no C++, no Python. Just Java 20+, the Vector API, and audacity.

Jlama is the project that makes people do a double-take. Created by Jake Luciani, it’s a complete LLM inference engine written entirely in Java, leveraging the Vector API (Project Panama) for SIMD-level performance. It loads GGUF and SafeTensors model files, supports Llama, Mistral, Gemma, Qwen2, and more, and runs them directly in your JVM.

The implications are wild: embed a model *inside* your Java application. Same process, same classloader, same monitoring. No Ollama server, no Docker container, no REST call. Just `new LlamaModel(...)` and go. Jlama integrates with LangChain4j, has a Quarkus extension, supports quantized models (Q4, Q8), and even does distributed inference across multiple JVM nodes.

Is it as fast as llama.cpp? No - C++ with CUDA will always win the raw benchmark game. But for Java-native deployment, edge computing, air-gapped environments, and the sheer developer experience of “everything is Java” - Jlama is genuinely revolutionary. This is the llama.cpp of the JVM world, and the Vector API in JDK 25 will only make it faster.

Testcontainers + Ollama (AI Testing)

TLDR: Integration tests that actually talk to an LLM. Testcontainers makes it real, reproducible, and CI-friendly.

Here’s a question that haunts every AI application team: “How do you test this thing?” Mocking LLM responses is fragile and doesn’t catch prompt regressions. Calling a real API in CI is expensive and flaky. Enter Testcontainers with Ollama.

Spin up an `OllamaContainer`, pull a small model (like `tinylama` or `phi`), and run your RAG pipeline against a real model in your JUnit test. The result is deterministic enough to catch regressions, real enough to validate your prompt engineering, and cheap enough to run on every PR. Both Quarkus (via Dev Services) and Spring Boot (via Testcontainers auto-configuration) support this pattern natively.

The Quarkus LangChain4j extension even has built-in semantic testing strategies — compare actual LLM output against expected output using embedding similarity or an “AI judge.” It’s not perfect (nothing with LLMs is), but it’s light-years better than “I clicked it manually and it looked fine.”

LangChain4j-CDI (Jakarta EE Integration)

TLDR: AI for the Jakarta EE world. Because not everything runs on Spring Boot, and that’s okay.

The JVM ecosystem isn’t just Spring. There are WildFly installations, Payara clusters, and Open Liberty instances quietly running the world’s banking systems. LangChain4j-CDI (originally SmallRye-LLM, donated to the LangChain4j umbrella) brings AI capabilities to Jakarta EE via CDI.

MicroProfile Config for model configuration, MicroProfile Fault Tolerance for retries and circuit breakers on LLM calls, MicroProfile Telemetry for OpenTelemetry-based observability — it’s the full MicroProfile treatment for AI workloads. `@RegisterAiService` works just like in Quarkus, but on any CDI 4.x runtime. For enterprises that can’t (or won’t) migrate to Spring Boot, this is the on-ramp to Gen AI that doesn’t require rewriting the platform.

DevovxGenie / JetBrains AI Assistant

TLDR: AI coding assistance *inside* your Java IDE. When you want AI to help you write the code that calls AI.

Meta moment: you're using an AI tool to help you build an AI-powered application. DevovxGenie (the IntelliJ IDEA plugin created by Stephan Janssen) and JetBrains' own AI Assistant bring LLM-powered code generation, explanation, and refactoring directly into the IDE where Java developers live.

DevovxGenie is particularly interesting because it supports local models via Ollama and Jlama — meaning your code completions can run entirely on your hardware. No data leaves your machine, no API costs, just you and a local model arguing about whether that [Stream](#) pipeline is readable. For enterprises with strict data governance, this is the AI coding assistant that legal actually approves.

Level 4: The Abyss (The “I’m Running CUDA From My JVM” Zone)



Here, you lose all sense of normal development workflows. You know what [MemorySegment](#) does and you have opinions about [Arena](#) lifecycle management. You've read JEP 489 and know why [Float16](#) matters for AI. Your colleagues think you're building a REST API; you're actually wiring GPU kernels.

The depth crushes your mind, yet you can't stop descending - something is calling you, something closer to the essence of the virtual machine than yet another CRUD (or another chatbot wrapper).

TornadoVM

TLDR: Write standard Java. Run it on a GPU. No CUDA, no JNI, no native code. Just annotations and black magic.

TornadoVM, from the University of Manchester's Beehive Lab, is an approach from another dimension. It extends OpenJDK/GraalVM with a compiler that takes standard Java code — annotated with task graphs - and compiles it Ahead-of-Time directly to GPU kernels (OpenCL or Nvidia PTX). You write matrix multiplication in Java; TornadoVM runs it on your GPU.

For AI workloads, this means: write your inference logic, your embedding similarity calculations, your attention mechanism in *pure Java*, and run it on heterogeneous hardware without ever touching JNI or C++. It's “write once, accelerate everywhere” taken to its logical extreme.

Is it production-ready for all workloads? Not yet. But the trajectory is clear, and the research is genuinely impressive. When combined with the Vector API and Project Panama, TornadoVM represents a possible future where Java doesn't just *call* AI - it *is* the AI runtime.

GPULLama3.java

TLDR: LLM inference on a GPU, written in pure Java. The TornadoVM team decided not to wait for the OpenJDK committees.

While Project Babylon and HAT are being designed in OpenJDK committee meetings (more on those in a moment), the TornadoVM team shipped GPULLama3.java - the first Java-native implementation of Llama3 with automatic GPU acceleration. Built on top of Alfonso Peterssen's original Llama3.java (a brilliant CPU-only implementation) and TornadoVM's GPU compilation, it proves the concept: you *can* run LLM inference on a GPU from pure Java.

It's a proof of concept, not a production engine. But it's the kind of proof of concept that makes JVM architects sit up straight and think “wait, maybe we don't need that Python sidecar after all.” The performance gap with C++ is still real, but it's narrowing with every JDK release.

Project Panama FFM for Native AI Libraries

TLDR: The new JNI. Call CUDA, cuDNN, and ONNX Runtime C APIs directly from Java without writing a single line of C.

The old evil was JNI (Java Native Interface) - slow, unsafe, and requiring C glue code that made everyone miserable. Project Panama's Foreign Function & Memory (FFM) API, finalized in JDK 22, replaces it entirely.

For AI specifically, FFM is critical: it lets you allocate off-heap memory ([MemorySegment](#), [Arena](#)) for huge tensors and model weights that can't (and shouldn't) live on the GC-managed heap. It lets you call `cuda.h` functions directly from Java via `jextract`-generated bindings. It lets you share memory buffers between Java and native libraries without copying.

ONNX Runtime Java, DJL, and Jlama all benefit from FFM under the hood. It's not a tool you "use" directly as an AI developer — it's the foundation that makes everything else possible. Like garbage collection, you don't think about it until it's not there. And then you think about it *a lot*.

Vector API (JEP 508, Incubator Round 10)

TLDR: SIMD in pure Java. Tell the CPU "do eight float operations at once" and watch your embeddings fly.

AI math is 99% matrix operations. Java loops are naturally scalar. The JIT compiler *tries* to auto-vectorize them, but it often fails on complex patterns. The Vector API is your manual override: explicitly tell the JVM to use SIMD (Single Instruction, Multiple Data) CPU instructions for parallel numeric computation.

For AI workloads, this means: embedding distance calculations, cosine similarity, dot products, softmax - all dramatically faster. Jlama is built on top of it. DJL benefits from it. Any Java code doing numeric computation for AI gets a massive boost.

The catch? It's been in incubator since JDK 16, and we're now at the *tenth* round (JEP 508). The dependency on Project Valhalla's `Float16` value type keeps pushing the timeline. But the JDK 25 release brings it closer than ever, and when it finally goes GA, the performance ceiling for pure Java AI inference will jump significantly.

Level 5: The Void (The "I Read JEP Drafts For Fun" Zone)



You are beyond help now. You attend Project Babylon mailing list discussions. You know what "Code Reflection" means and why it matters for GPU compilation. Your stand-up updates sound like research papers. "What did you do yesterday?" "I was analyzing the implications of JEP 489's `Float16` primitive class on Vector API specialization for transformer attention head computation." *Silence in the room.*

Welcome to the bottom. No light reaches here. Only TRUE Machine Learning stuff.

Project Valhalla's Float16

TLDR: A true 16-bit float type for Java, with zero overhead. The missing primitive that AI workloads have been begging for.

AI loves `float16`. It's faster, uses half the memory of `float32`, and most model weights don't need full precision anyway. But Java doesn't have a native `float16` type. The current workaround? Wrap a `short` and pray. No type safety, no operator overloading, no Vector API specialization.

Project Valhalla - Java's decade-long quest to fix the primitive/object schism - will deliver "primitive classes" that allow a `Float16` value class wrapping a `short` with **zero performance overhead** and full type safety. When this lands, the Vector API can specialize on it, Jlama gets faster, ONNX Runtime interop becomes cleaner, and every numeric AI computation in Java becomes more efficient.

The catch? It's Valhalla. We've been waiting since 2014. Tonight, we (rather not) dine in Valhalla — but this time, the appetizers are closer than they've ever been.

Project Babylon (Code Reflection)

TLDR: Java gets the ability to inspect and transform its own code at a semantic level. The GPU compilation endgame.

Project Babylon is the new mythical OpenJDK project that could change everything for Java AI. It introduces “Code Reflection” - not the old `java.lang.reflect` that looks at classes and methods, but a deep, structured representation of what your Java code *actually does*. Method bodies, control flow, data flow - all available as an analyzable tree.

The AI implication is mind-blowing: write your model (a neural network, an attention mechanism, a loss function) in standard Java. Babylon’s Code Reflection analyzes that code and automatically translates it into optimized GPU kernels. No Python. No external model files. No JNI. Just Java → GPU.

It’s the endgame: if Valhalla gives us the right data types, and Panama gives us the hardware bridge, Babylon gives us the compiler that turns Java code into accelerator code. The Holy Trinity of Java’s AI future.

HAT (Heterogeneous Accelerator Toolkit)

TLDR: Babylon’s chariot into battle. The concrete implementation that takes Code Reflection and turns it into GPU kernels via Panama.

HAT shouldn’t be considered a separate project - it’s the chariot Babylon rides into battle. It takes Babylon’s Code Reflection output and, using Panama’s FFM API as the transport layer, compiles Java methods into OpenCL or PTX kernels that run on GPUs, FPGAs, and other accelerators.

No JNI, no ad-hoc glue: Babylon describes the computation, HAT turns it into kernels, and Panama wires the whole thing into whatever silicon you have underneath. It’s three once-separate OpenJDK efforts snapping together into a single pipeline that lets plain Java reach all the weird and wonderful hardware in your data center.

Production-ready? Not yet. But the research implementations exist, the JEPs are progressing, and the trajectory is unmistakable. The JVM is building toward a future where “Java for AI” doesn’t mean “Java calling Python” - it means “Java *being* the AI runtime.”

Llama3.java / Pure Java Model Implementations

TLDR: When someone says “can you implement LLM inference from scratch in Java?” and the answer is “hold my espresso.”

Alfonso Peterssen’s Llama3.java proved something important: you can implement the entire Llama3 inference stack in a single Java file. No frameworks, no dependencies, just the Vector API, some clever memory mapping, and a deep understanding of the transformer architecture. It’s educational, it’s surprisingly performant, and it spawned GPULLama3.java.

These “from scratch” implementations matter because they demonstrate that the JVM’s modern features - Vector API, Panama, value types - are sufficient to express AI workloads natively. They’re not production tools; they’re proof that the platform is ready. When Valhalla, Babylon, and HAT converge, these experiments will have been the rehearsals.

And that’s our iceberg, folks. From [@Autowired ChatClient](#) all the way down to hand-crafting GPU kernels from Java bytecode, the Gen AI tooling ecosystem in the JVM world is deeper, richer, and more exciting than the “Java can’t do AI” crowd would have you believe.

The surface is friendly and productive — you can build a production RAG application in an afternoon with Spring AI or LangChain4j, and that’s genuinely impressive. But the depths? The depths are where the JVM is quietly reinventing itself as a platform that doesn’t just *call* AI services, but *runs* AI workloads natively.

Python built the research lab. Java is building the factory. And the factory is starting to build its own machines.

Just don’t say I didn’t warn you about the `pom.xml` file length.

Now you’ve got enough material on Gen AI tooling in Java to keep you busy for the next month - consider it your honorary PhD on the topic!



Building AI-Driven Java Systems with Spring AI and Java 26

API & Frameworks

SpringAI

Java26

StructuredConcurrency

RAG

VirtualThreads

Author: Damiana Nascimento

Damiana Costa is a Software Engineer and Architect with nearly 20 years of experience in backend development, distributed systems, and software architecture. She currently works as a Principal Engineer, focusing on building scalable, resilient platforms and designing high-impact technical solutions for large-scale systems. Her professional background spans multiple industries, including finance, e-commerce, telecommunications, and healthcare. She has extensive experience with Java, cloud-native architectures, microservices, distributed systems, observability, performance, and security. In addition to her industry work, Damiana is a university lecturer in Software Engineering, teaching topics such as Java, software architecture, and system design. She is also a frequent speaker at technology conferences, where she shares practical insights on architecture, resilience, and modern backend development.



LinkedIn

Spring AI and Java 26 make building AI-powered Java applications finally practical. This article shows how developers can integrate LLMs into production systems using modern concurrency, performance improvements and clean architectures. It discusses real-world concerns such as scalability, latency, observability and reliability when bringing AI into Java back-ends.

AI Has Finally Found a Practical Path in the Java Ecosystem

The adoption of LLMs in Java systems introduces new architectural pressures, including variable latency, per-request cost, non-deterministic behavior, dependency on external providers, and increased sensitivity to observability and governance. This means AI cannot be treated as just another HTTP call embedded within an existing service. When this decision is made too early, systems quickly become difficult to evolve, expensive to operate, and opaque to those responsible for maintaining them.

As a result, the most important consequence of this convergence is not just increased productivity. More than simplifying LLM integration, it reshapes the role of AI within the backend. Instead of adapting the entire application to the model, it becomes viable to adapt how the model is used within the system's architecture.

This article is built on a simple but decisive premise: **LLMs do not replace architecture; they put it under pressure.** By introducing a generative model into the business flow, applications must now handle new requirements around isolation, measurement, fallback strategies, security, and predictability. The role of Spring AI and Java 26 is to reduce the technical friction of this integration—not to eliminate the architectural responsibility of those designing the system.

What Makes AI-Driven Java Systems Practical Now

The viability of AI in Java today is not driven solely by model evolution, but by the maturation of the ecosystem across three fronts: integration abstraction, modern concurrency, and operational efficiency.

AI Abstraction at the Framework Level

A major challenge in LLM adoption is tight coupling with providers, where API contracts and parameters leak into the code-base. **Spring AI** addresses this by providing core abstractions (such as `ChatModel` and `VectorStore`), containing complexity within well-defined boundaries.

By structuring integration this way, developers maintain clarity over key decisions—like RAG strategies and token usage—without compromising architectural evolution. The central benefit is the ability to switch providers or adjust strategies without rewriting the application's core logic.

Modern Concurrency in Java 26

AI-driven Java systems are predominantly **I/O-bound**, spending most of their time waiting for responses from external APIs and vector stores. The bottleneck is not CPU capacity, but the system's ability to coordinate these waiting periods efficiently. With **Virtual Threads**, Java allows developers to write simple, synchronous flows while the JVM manages thousands of concurrent tasks with minimal overhead. **JDK 26** further reinforces this through **Structured Concurrency**, treating AI pipeline subtasks—such as context retrieval and model invocation—as a single logical unit. This ensures consistent failure propagation, shared deadlines, and cleaner resource management.

Performance Considerations in AI-Driven Java Systems

There is no “AI mode” in the JVM, but incremental platform improvements directly impact applications heavy on remote calls. JDK 26 introduces refinements in AOT caching, G1 collector optimizations, and better virtual thread behavior during class initialization.

These updates increase operational predictability and reduce internal overhead. The fundamental shift is that the platform now provides a natural, scalable foundation for integrating AI without distorting the application's design.

Reference Architecture for AI-Driven Java Systems

To support these requirements, a clear architectural approach is necessary. Integrating LLMs into Java applications requires treating AI as a first-class component with explicit responsibilities. A common mistake is scattering model calls across controllers or services, which compromises maintainability. Instead, a robust architecture introduces an **AI Orchestration Layer** to shield the system from model-specific variability, concentrating decisions like prompt construction, model selection, and fallback strategies in one place.

Recommended Architectural Layers

To ensure scalability and observability, the architecture should be organized into five complementary layers:

1. **API/Application Layer:** Exposes interfaces (REST/GraphQL) and remains focused on use cases, isolated from LLM implementation details.
2. **AI Orchestration Layer:** Coordinates model integration, including context assembly, parameter tuning (temperature, tokens), and output parsing.
3. **Domain Layer:** Preserves business rules, preventing AI constraints from contaminating core logic.
4. **Context/Retrieval Layer:** Manages structured data, embeddings, and vector stores to enrich model input (RAG).
5. **Observability & Resilience Layer:** Implements telemetry, circuit breakers, and retries to make the system measurable and defensible.

Execution Flow With RAG

Retrieval-Augmented Generation (RAG) is the standard pattern for providing up-to-date context without retraining models. It separates knowledge preparation from response generation, essential for enterprise systems where information evolves rapidly.

A typical execution flow follows these steps:

1. **Request:** The client sends a command; the controller maps it to a use case.
2. **Retrieval:** The orchestrator performs a semantic search in a vector store to find relevant data chunks.
3. **Augmentation:** These chunks enrich the prompt, providing the model with specific context.
4. **Inference:** **Spring AI** invokes the model via `ChatClient`.
5. **Validation:** The response is parsed, validated, and returned.
6. **Telemetry:** Logs, metrics, and traces are captured throughout the process.

Figure 1 — RAG Pipeline

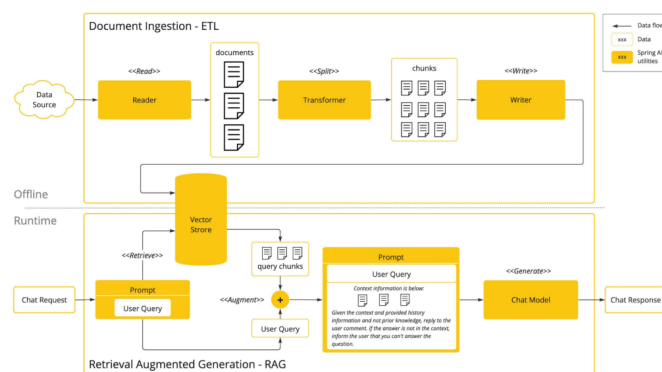


Abb. 6: Figure 1 — Retrieval-Augmented Generation (RAG) pipeline, illustrating document ingestion, chunking, vector store indexing, and context-enriched response generation. Source: [Spring AI Reference Documentation](#).

RAG Trade-Offs in the Architecture

From an architectural perspective, RAG introduces a clear separation between offline processing (ingesting, chunking, and vectorizing documents) and runtime processing (querying and prompt assembly). This separation improves governance, as knowledge updates happen independently of the inference stage.

That said, RAG should not be a default for every scenario. In highly structured domains with predictable queries, traditional data retrieval might be more efficient than semantic search. **Crucially, RAG does not eliminate hallucination; it only improves grounding.** If retrieval is poor, context is biased, or the prompt is weak, the model will still produce errors. The decision to implement RAG must weigh domain variability, latency constraints, and operational costs against the actual benefit of semantic retrieval.

Applying Modern Concurrency in AI-Driven Java Systems

From a runtime perspective, AI-driven Java systems are fundamentally **I/O-bound**. Most of the lifecycle of a request is spent waiting for LLM responses, vector store queries, and embedding retrievals.

Virtual Threads: Concurrency Without Accidental Complexity

Virtual threads offer a low-cost model for handling thousands of concurrent tasks while preserving a straightforward, synchronous programming style.

In AI systems—where logic is already burdened by prompt design and output validation—reducing the “accidental complexity” of concurrency is vital. The code stays readable and close to the domain language, while the platform handles the operational heavy lifting.

Structured Concurrency: Coordinating Composite Workflows

An AI request usually involves multiple steps: loading user profiles, retrieving context, and invoking the model. Coordinating these via scattered `Future` objects or callbacks increases coupling and obscures the flow.

Structured Concurrency (refined in JDK 26) treats related subtasks as a single logical unit. This ensures that cancellation, exception propagation, and deadlines are managed consistently across the entire workflow.

The following example illustrates how Structured Concurrency can coordinate independent tasks safely:

```
1  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
2      var contextTask = scope.fork(() -> retrievalService.search(question));
3      var profileTask = scope.fork(() -> userService.loadProfile(userId));
4
5      scope.join().throwIfFailed();
6
7      var context = contextTask.get();
8      var profile = profileTask.get();
9
10     return aiOrchestrator.generateAnswer(question, context, profile);
11 }
```

This reflects a disciplined architectural decision: parallelize independent tasks while keeping failure handling under explicit coordination.

Where Virtual Threads Help — and Where They Do Not

Virtual threads are highly effective for waiting-heavy workloads but do not solve CPU-bound bottlenecks or external constraints like **rate limits** and **token costs**. While they improve the system’s ability to coordinate waiting locally, they cannot eliminate the saturation of external providers.

JDK 26 introduces refinements to mitigate issues like thread pinning, but engineering discipline remains necessary. Indiscriminately parallelizing every remote call can lead to excessive fan-out and overloaded services. In the context of AI-driven Java systems, **useful concurrency** is prioritized over maximum concurrency, focusing on operational predictability and clean design.

Spring AI as an Integration Layer, Not a Magic Shortcut

Generative models introduce architectural decisions that cannot be automated away. Instead, Spring AI provides a consistent integration surface aligned with the Spring ecosystem. It transforms AI into an explicit system responsibility rather than a collection of scattered HTTP calls.

ChatClient and ChatModel: When to Abstract and When to Control

Spring AI offers two primary interfaces for model interaction:

- **ChatClient:** A high-level API designed to reduce boilerplate for straightforward flows.
- **ChatModel:** A lower-level interface providing granular control over metadata, tool calling, and specialized integration strategies.

The goal is to encapsulate these choices within an orchestration layer, shielding the domain from model-specific details.

```

1  @Service
2  public class AiOrchestrator {
3
4      private final ChatClient chatClient;
5
6      public AiOrchestrator(ChatClient.Builder builder) {
7          this.chatClient = builder.build();
8      }
9
10     public String generateAnswer(String question, String context) {
11         String prompt = buildPrompt(question, context);
12         return chatClient.prompt(prompt).call().content();
13     }
14
15     private String buildPrompt(String question, String context) {
16         return """
17             Use the context below to answer the question.
18
19             Context:
20             %s
21
22             Question:
23             %s
24             """.formatted(context, question);
25     }
26 }

```

This encapsulation allows prompts, models, and fallback policies to evolve without leaking dependencies across the codebase.

Tool Calling: Power and Risk in the Same Package

Tool calling allows LLMs to interact with internal functions, transforming text generation into operational workflows. However, this increases the system's error surface. When a model triggers a function, its output directly influences system state. Consequently, inputs must be validated, permissions constrained, and results meticulously observed. In enterprise environments, tool calling without governance is merely fragile automation.

Spring AI and Observable Integration

Spring AI provides built-in observability for components like `ChatModel` and `VectorStore`, integrating metrics and tracing for generation and retrieval operations. This is critical for answering production-level questions regarding latency, token costs, and bottlenecks.

By leveraging the broader Spring observability ecosystem, AI stops being a “black box” and becomes a measurable part of the backend. The framework’s value lies in reducing integration friction while maintaining architectural responsibility—simplifying instrumentation without hiding the need for robust output validation and cost control.

Operating AI-Driven Java Systems in Production

In AI-driven Java systems, scalability is inextricably linked to economics. Each model invocation carries a direct usage cost, creating a “triangle of trade-offs” between **latency, scalability, and cost**. Improving one dimension—such as using a more capable model for better quality—inevitably puts pressure on the others.

The most effective architectures maximize the value of each call rather than the quantity. This involves reserving LLMs for tasks where natural language synthesis is truly required, while handling structured queries or simple logic through traditional business rules.

When to Use (and Not Use) LLMs

One of the most expensive mistakes in AI-driven Java systems is turning the model into a mandatory step for every decision. Many requests can—and should—be handled through business rules, structured queries, heuristics, or simple data retrieval. Using an LLM here adds latency and cost without proportional benefit.

The best architecture does not maximize the number of model calls. It maximizes the value of each model call. This means reserving generation for tasks where natural language, synthesis, interpretation, or composition genuinely add capabilities that the rest of the system does not provide effectively.

Latency, Scalability, and Cost

Since models are typically billed by token, cost must be treated as a first-class metric.

To manage this triangle, architects should adopt three core strategies: model routing (using smaller models for classification and reserving frontier models for synthesis), context optimization (reducing token usage and eliminating redundant inputs), and predictable degradation (using streaming and timeouts to preserve responsiveness).

True scalability in AI-driven Java systems now depends on controlling fan-out and limiting useful concurrency rather than simply adding more CPU instances.

Observability Beyond Traditional Metrics

Traditional metrics are no longer sufficient; AI introduces an operational “black box.” Teams must track **cost per use case**, **token consumption**, and **segmental latency** (distinguishing retrieval from inference).

In RAG architectures, distributed tracing is essential to prevent retrieval issues from being mistaken for model hallucinations. By instrumenting every stage—from vector search to context assembly—teams can fine-tune the pipeline with precision. Furthermore, Spring AI ensures security by not exporting sensitive prompt content by default, enabling metadata-driven observability without exposing sensitive data.

Reliability and Control in AI Systems

Integrating LLMs amplifies complexity by adding probabilistic behavior. Reliability here is about maintaining system integrity when a model fails or produces misaligned results.

Maintaining execution control requires a disciplined approach:

- **Timeouts & Circuit Breakers:** Prevent cascading failures and avoid repeatedly issuing expensive calls that are unlikely to succeed during provider saturation.
- **Selective Retries:** Apply retries only to transient network issues, using exponential backoff to avoid cost spikes.
- **Fallback & Validation:** Treat model output as **untrusted external input**. Whether generating text or triggering tools, results must undergo structural validation. For operations with side effects, **idempotency** is critical to ensure that semantic variability does not lead to duplicated actions.

Practical Example: an AI-Driven Enterprise Assistant

To consolidate the concepts discussed, consider an internal assistant capable of answering technical questions based on company documentation. This scenario brings together key concerns such as LLM integration, context retrieval, parallelism, observability, and reliability. More importantly, it demonstrates that AI can participate in the business flow without taking control of the system. The model operates as a specialized component within a broader architecture—not as a replacement for application logic.

Orchestration With Parallelism

In this scenario, Structured Concurrency is used to coordinate independent operations such as context retrieval and user data loading. Instead of executing these steps sequentially, the system performs them in parallel while maintaining a single logical unit of work.

This approach reduces overall latency without introducing asynchronous complexity into the business logic. More importantly, it preserves control over failure propagation, cancellation, and execution boundaries—ensuring that the orchestration remains both efficient and predictable ([OpenJDK Project Loom](#) / JDK 26).

AI Orchestration With Spring AI

Interaction with the model should be encapsulated within a dedicated service:

```

1  @Service
2  public class AiOrchestrator {
3
4      private final ChatClient chatClient;
5
6      public AiOrchestrator(ChatClient.Builder builder) {
7          this.chatClient = builder.build();
8      }
9
10     public String generateAnswer(String question, String context, UserProfile profile) {
11
12         String prompt = ""
13         You are a technical assistant.
14
15         Context:
16         %s
17
18         User profile:
19         %s
20
21         Question:
22         %s
23         """.formatted(context, profile.summary(), question);
24
25         return chatClient.prompt(prompt).call().content();
26     }
27 }
```

This design makes it easier to evolve prompts, switch models, adjust parameters, and introduce structured validation later—without contaminating the rest of the application.

AI as Part of the Architecture, Not an Exception

Integrating LLMs into Java applications is no longer experimental. With **Spring AI** and **Java 26**, the ecosystem provides a solid foundation for building real-world systems with reduced technical friction and superior concurrency ergonomics via Virtual Threads and Structured Concurrency.

However, technology alone is not the decisive factor—architectural discipline is. LLMs introduce unpredictable latency, variable costs, and probabilistic behavior. Developing AI-driven Java systems does not mean abandoning established best practices; it means extending them. Separation of concerns, observability, and dependency isolation must now encompass model behavior. This implies:

- **Encapsulation:** Treating the LLM as a critical dependency with well-defined boundaries.
- **Measurement:** Tracking latency and token consumption as first-class metrics.
- **Governance:** Validating outputs and designing controlled degradation.

The difference between an AI prototype and a production system lies in the architecture. While prototypes tolerate direct calls and minimal instrumentation, real systems require explicit limits, fallback strategies, and rigorous validation.

Systems that treat AI as a detail tend to fail silently.

Systems that treat it as architecture tend to evolve.

› [Back to the Table of Contents](#) ‹

JCON2027
www.jcon.one

JAVAPRO



📍 CINEDOM COLOGNE

JCON EUROPE 2027

International Java Community Conference



LangChain4j Agentic Workflows: From AI Calls to Multi-Agent Systems in Java

API & Frameworks

AgenticAI

AI

LangChain4j

Author: Loïc Magnette

JCON SPEAKER

Loïc Magnette is a seasoned software developer with a strong background in consulting. Currently a senior developer at Oniryx, he specializes in Java and Angular, delivering innovative solutions and sharing his knowledge as a speaker. As a co-organizer of the Belgian Java User Group (BeJUG), he fosters connections within the developer community. Outside of tech, Loïc's passion for wildlife inspires his work and creativity.

Want to Dive Deeper?

Loïc Magnette is a speaker at [JCON](#)! If you can't attend live, The [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn

You have built AI features into your Java application. Your model is wrapped in a service, RAG is feeding it context, tools are wired, and calls are flowing. It works. Then requirements evolve. A single prompt-and-response is no longer enough. You need steps that follow each other, branches based on decisions, retries when things fail, and multiple actions running concurrently. The question shifts from “how do I call an LLM?” to “how do I orchestrate multiple LLM-driven tasks into a coherent system?”

This article walks through that transition using LangChain4j’s agentic module. We will start from simple agents, move through four [workflow patterns](#) and their composition, then progress to [goal-oriented](#) planning and [fully agentic systems](#), covering shared state, error handling, non-AI agents, human-in-the-loop, and the critical question of when to use which approach.

The intended audience is Java developers who already have basic AI integration experience. If you know how to call a model and wire a tool, you are ready for what comes next.

Getting Started with LangChain4j Agentic Workflows

LangChain4j’s agentic module is currently in beta. Add it alongside the core library and your preferred model provider:

```
1 <dependency>
2   <groupId>dev.langchain4j</groupId>
3   <artifactId>langchain4j-agentic</artifactId>
4   <version>1.12.1-beta21</version>
5 </dependency>
```

The goal-oriented planner lives in a separate module:

```
1 <dependency>
2   <groupId>dev.langchain4j</groupId>
3   <artifactId>langchain4j-agentic-patterns</artifactId>
4   <version>1.12.1-beta21</version>
5 </dependency>
```

All examples in this article use `AgenticServices` as the entry point for building agents and workflows. If you have used `AI Services` to build AI-backed interfaces, `AgenticServices` extends that same pattern for multi-agent orchestration.

The examples configure a chat model like this:

```
1 ChatModel model = OllamaChatModel.builder()
2   .baseUrl("http://localhost:11434")
3   .modelName("llama3.2:1b")
4   .build();
```

Any LangChain4j-supported model provider works. Swap in OpenAI, Anthropic, or Azure as needed. Check the LangChain4j [documentation](#) for the latest artifact coordinates and version.

Code examples use Java 25 features (`IO.println`, `IO.readLine`) for concise I/O. Replace with `System.out.println` and `BufferedReader` if you are on an earlier version.

The Building Block of Agentic Workflows: What Is an Agent?

Before we orchestrate anything, we need a common unit of work. In LangChain4j, an agent is a Java interface with a single method that performs a task, typically backed by an LLM. You define it declaratively:

```
1 public interface SithNameGenerator {
2     @UserMessage("""
3         You are a Sith naming council. Transform this boring name into something \
4         properly villainous. Return ONLY the Sith name, nothing else: {{jediName}}""")
5     @Agent("Transforms a Jedi name into a suitably menacing Sith identity")
6     String darken(@V("jediName") String jediName);
7 }
```


The `@Agent` annotation marks this as an agent with a description. The `@UserMessage` provides the prompt template. `@V` binds the method parameter to a template variable. We will see shortly why `@V` matters, as it becomes essential when agents communicate through shared state.

To instantiate an agent, you use the builder:

```
1 SithNameGenerator sithNamer = AgenticServices
2     .agentBuilder(SithNameGenerator.class)
3     .chatModel(model)
4     .outputKey("sithName")
5     .build();
6
7 String sithName = sithNamer.darken("Obi-Wan Kenobi");
```

Two details matter here. First, every agent has a **name** that uniquely identifies it in a system (derived from the method name or set explicitly). Second, every agent has an **output key**, a named slot in shared state where it writes its result. That output key is how agents communicate with each other without direct coupling.

This is the fundamental contract: an agent reads inputs, does its work, and writes its result to a shared location. Every agentic workflow in this article builds on that idea.

Shared State with AgenticScope

Once you have more than one agent, the question becomes: how do they exchange data? If agents communicate through named keys rather than direct method calls, there needs to be a shared container holding those keys. That container is the `AgenticScope`.

The `AgenticScope` implements the blackboard pattern: a shared state object that multiple independent components read from and write to, with no direct coupling between them. When the root agent is invoked, `LangChain4j` automatically create the scope. Each subsequent agent reads what it needs, does its work, and writes its output back.

The core operations are straightforward:

- `writeState(key, value)`: store a result
- `readState(key)`: retrieve a value
- `readState(key, defaultValue)`: retrieve with a fallback
- `hasState(key)`: check if a key exists

This pattern enables loose coupling. Agent A does not call Agent B. Agent A writes `"interceptedTransmission"` to the scope. Agent B reads `"interceptedTransmission"` from the scope. Neither knows about the other. The orchestration layer decides the order.

The scope also maintains an invocation history: `invocations()` returns the full list of agent executions and their responses, giving you a built-in audit trail.

@V vs @K: External Input vs Scope Input

Before we look at more examples, one distinction is critical. `LangChain4j` uses two annotations to feed data into agents:

- `@V("name")` binds a method parameter directly, used when calling an agent from outside or as the entry point of a pipeline. The value is also written to the scope, making it available to downstream agents via `@K`.
- `@K(Key.class)` reads a value from the `AgenticScope`, used when an agent runs inside a workflow and consumes data written by a previous agent.

Typed Keys

For production use, LangChain4j offers typed keys to prevent spelling errors and type mismatches:

```

1  public static class InterceptedTransmission implements TypedKey<String> { }
2
3  public interface ThreatClassifier {
4      @Agent("Classifies the threat level of an intercepted Imperial transmission")
5      @UserMessage("""
6          You are a Rebel Alliance intelligence analyst. \
7          Classify the threat level of this intercepted Imperial transmission as \
8          LOW, MEDIUM, HIGH, or CRITICAL. Respond with the level and a brief
9              justification.
10
11          Transmission: {{InterceptedTransmission}}""")
12      String classify(@K(InterceptedTransmission.class) String transmission);
13  }
```

The `@K` annotation maps a method parameter to a typed scope variable. The `TypedKey` class it references defines both the key name and the expected type, letting the compiler catch mismatches rather than your production logs.

Typed keys can also define a `defaultValue()` for when the key has not been written yet:

```

1  public static class MasterApprovals implements TypedKey<Integer> {
2      @Override
3      public Integer defaultValue() {
4          return 0;
5      }
6  }
```

You can chain agents in a typed sequence. The `TransmissionInterceptor` receives input via `@V` and writes to `InterceptedTransmission`; the `ThreatClassifier` reads it via `@K`:

```

1  IntelPipeline pipeline = AgenticServices
2      .sequenceBuilder(IntelPipeline.class)
3      .subAgents(interceptor, classifier)
4      .outputKey(ThreatLevel.class)
5      .build();
6
7  String threatAssessment = pipeline.analyze(scrambledInput);
```

When your system grows to dozens of agents, catching a `ThreatLevel` misread as a `String` at compile time is not a convenience. It is survival.

Agentic Workflow Pattern 1: Sequential

The simplest orchestration pattern is the sequential workflow. Agents execute one after another in a defined order, each reading from and writing to the AgenticScope:

```

1  public static class AptitudeReport implements TypedKey<String> { }
2  public static class AssignedMaster implements TypedKey<String> { }
3  public static class LightsaberRecommendation implements TypedKey<String> { }
4
5  public interface AptitudeTester {
6      @Agent("Evaluates a youngling's Force sensitivity and aptitudes")
7      @UserMessage(""""
8          You are a Jedi Temple instructor. Evaluate this youngling's Force aptitude. \
9          Provide a short report covering their strengths.
10         Youngling: {{name}}""")
11      String test(@V("name") String name);
12  }
13  public interface MasterAssigner {
14      @Agent("Assigns the most suitable Jedi Master based on the youngling's aptitudes")
15      @UserMessage(""""
16          You are the Jedi Council. Based on this aptitude report, assign the most \
17          suitable Jedi Master from the Order.
18          Aptitude report: {{AptitudeReport}}""")
19      String assign(@K(AptitudeReport.class) String aptitude);
20  }
21  public interface LightsaberGuide {
22      @Agent("Recommends lightsaber form and crystal based on master assignment and
23          aptitude")
24      @UserMessage(""""
25          You are the Jedi Temple lightsaber instructor. Based on the assigned master \
26          and training path, recommend a lightsaber form and kyber crystal color.
27          Assigned master: {{AssignedMaster}}""")
28      String guide(@K(AssignedMaster.class) String master);
29  }

```

Each agent declares what it reads (@K) and writes (outputKey). The pipeline chains them together:

```

1  AptitudeTester aptitudeTester = AgenticServices
2      .agentBuilder(AptitudeTester.class)
3      .chatModel(model)
4      .outputKey(AptitudeReport.class)
5      .build();
6
7  MasterAssigner masterAssigner = AgenticServices
8      .agentBuilder(MasterAssigner.class)
9      .chatModel(model)
10     .outputKey(AssignedMaster.class)
11     .build();
12
13  LightsaberGuide lightsaberGuide = AgenticServices
14     .agentBuilder(LightsaberGuide.class)
15     .chatModel(model)
16     .outputKey(LightsaberRecommendation.class)
17     .build();
18
19  JediTrainingPipeline pipeline = AgenticServices
20     .sequenceBuilder(JediTrainingPipeline.class)
21     .subAgents(aptitudeTester, masterAssigner, lightsaberGuide)
22     .outputKey(LightsaberRecommendation.class)
23     .build();
24
25  String result = pipeline.train("Ahsoka Tano");

```

Here, `aptitudeTester` measures Force aptitude and writes an `AptitudeReport` to the scope. Then `masterAssigner` reads the report via `@K` and pairs the Padawan with a compatible Master. Finally, `lightsaberGuide` reads the `AssignedMaster` and produces the weapon construction specifications.

This is the pattern you reach for when tasks have a natural ordering and each step depends on the previous one. Think of content pipelines, multi-stage data processing, or any chain-of-thought decomposition where you want explicit control over each step.

One note on the builder: `sequenceBuilder(JediTrainingPipeline.class)` creates a typed pipeline, giving you a concrete interface to call. When you don't need a typed interface (for example, when a workflow is only used as a sub-agent inside another workflow), you can use `sequenceBuilder()` without a type argument, which returns an `UntypedAgent`. We will see this in the composing patterns section.

Agentic Workflow Pattern 2: Loop

Not every problem is solved in a single pass. Sometimes you need iterative refinement: write a draft, score it, revise, score again, until the quality threshold is met. That is the loop workflow:

```

1  public interface AdmiralAckbar {
2      @Agent("Reviews an attack plan for traps and tactical weaknesses")
3      @UserMessage(""""
4          You are Admiral Ackbar. Review this attack plan. If you find issues, \
5          explain them briefly and say REJECTED. If the plan is solid, say APPROVED.
6
7          Attack plan: {{AttackPlan}}""")
8      String review(@K(AttackPlan.class) String plan);
9  }
10
11 public interface PlanReviser {
12     @Agent("Revises the attack plan based on Admiral Ackbar's feedback")
13     @UserMessage(""""
14         You are a Rebel Alliance battle strategist. Revise the attack plan \
15         to address Admiral Ackbar's concerns.
16
17         Current plan: {{AttackPlan}}
18
19         Ackbar's feedback: {{ReviewFeedback}}""")
20     String revise(@K(AttackPlan.class) String plan, @K(ReviewFeedback.class) String
21                 feedback);
22 }
```

The loop runs `ackbar` (reviews the current attack plan, says APPROVED when no critical weaknesses remain, otherwise produces a list of concerns, ideally including at least one “It’s a trap!”) then `reviser` (addresses the concerns and updates the plan). After three iterations, they fly in anyway and trust the Force.

Two configuration points matter:

- **maxIterations(n)**: a hard safety cap. Without this, a loop that never satisfies its exit condition runs forever. Always set this.
- **exitCondition(Predicate<AgenticScope>)**: an optional condition that ends the loop early. It has access to the full scope, so you can base it on scores, flags, or any combination. The composing patterns section will show this in action.

The loop pattern is powerful for quality gates. A gatekeeper agent evaluates the output, a refinement agent improves it, and the loop continues until the bar is cleared or the maximum iterations are reached.

Agentic Workflow Pattern 3: Parallel

When agents are independent and can work without each other's results, running them

```

1 SpaceFleetStrategist spaceFleetStrategist = AgenticServices
2   .agentBuilder(SpaceFleetStrategist.class)
3   .chatModel(model)
4   .outputKey("fleetDisposition").build();
5 GroundAssaultAgent groundAssaultAgent = AgenticServices
6   .agentBuilder(GroundAssaultAgent.class)
7   .chatModel(model)
8   .outputKey("ewokGroundStrategy").build();
9 JediMissionPlanner jediMissionPlanner = AgenticServices
10  .agentBuilder(JediMissionPlanner.class)
11  .chatModel(model)
12  .outputKey("lukeObjective").build();
13 BattleOfEndorPipeline pipeline = AgenticServices
14  .parallelBuilder(BattleOfEndorPipeline.class)
15  .subAgents(spaceFleetStrategist, groundAssaultAgent, jediMissionPlanner)
16  .executor(Executors.newFixedThreadPool(3))
17  .output(scope -> assembleBattlePlan(
18      (String) scope.readState("fleetDisposition"),
19      (String) scope.readState("ewokGroundStrategy"),
20      (String) scope.readState("lukeObjective")))
21  .outputKey("battleOfEndor").build();

```

Note that this example uses string-based output keys for brevity; typed keys work identically and are recommended for production code.

`spaceFleetStrategist`, `groundAssaultAgent`, and `jediMissionPlanner` run simultaneously on separate threads. The fleet engagement above Endor, the Ewok ground coordination, and Luke's solo mission into the Death Star are all planned at the same time. When all three finish, the `output()` function assembles the final battle order. If each agent takes 3 seconds, the total is 3 seconds, not 9.

You provide the thread pool via `executor()`, so you control the concurrency model. The `output()` function merges the parallel results into a single coherent output.

Because agents run on separate threads, treat the `AgenticScope` as the single point of coordination. Each agent should write to its own distinct output key. The scope handles concurrent writes safely.

Agentic Workflow Pattern 4: Conditional

When different inputs require different processing paths, you need conditional branching. The conditional workflow routes execution based on the current state.

A notable LangChain4j feature at play here: agents can return Java enums directly. The framework maps the LLM's text response to the enum value, which makes enums natural for conditional routing predicates:

```

1 public enum AlignmentType { LIGHT_SIDE, DARK_SIDE, NEUTRAL }
2 public interface AlignmentClassifier {
3     @Agent(value = "Classifies a Star Wars character's Force alignment")
4     @UserMessage("""
5         You are a Force-sensitive oracle. Classify this character's alignment as
6         exactly one of: LIGHT_SIDE, DARK_SIDE, or NEUTRAL.
7         Respond with ONLY the alignment label, nothing else.
8         Character: {{CharacterName}}""")
9     AlignmentType classify(@V("CharacterName") String name);
10 }

```

The classifier writes its result to the scope. Then the conditional router dispatches to the right specialist:

```
1 var conditionalRouter = AgenticServices.conditionalBuilder()
2   .subAgents(scope
3     -> scope.readState(Alignment.class) == AlignmentType.LIGHT_SIDE,
4     jediCouncilAgent)
5   .subAgents(scope
6     -> scope.readState(Alignment.class) == AlignmentType.DARK_SIDE,
7     sithLordAgent)
8   .subAgents(scope
9     -> scope.readState(Alignment.class) == AlignmentType.NEUTRAL,
10    mandalorianAgent)
11 .build();
12 MissionRouter router = AgenticServices
13   .sequenceBuilder(MissionRouter.class)
14   .subAgents(alignmentClassifier, conditionalRouter)
15   .outputKey(MissionBriefing.class)
16   .build();
```

Each sub-agent is paired with a predicate. The predicates are evaluated against the `AgenticScope`, and only the matching agent is invoked. The `.subAgents()` method in the conditional builder is overloaded: it accepts a predicate-agent pair, unlike the plain list version used in sequential and parallel builders.

The architecture here is worth examining: the conditional router is wrapped inside a sequence. The `AlignmentClassifier` agent reads the inbound request, writes the force alignment to the scope, and the conditional workflow routes to the right specialist. No LLM decides the routing. You do, with explicit predicates.

Composing Agentic Workflow Patterns

The real power of these patterns emerges when you combine them. In `LangChain4j`, every workflow is itself an agent, which means it can be used as a sub-agent in another workflow. A loop inside a sequence. A parallel step inside a conditional. A conditional inside a loop.

Consider a Rebel Alliance assault planner: a sequence that first decodes intercepted Imperial transmissions, then loops through Jedi Council debate until the battle plan reaches approval, and finally broadcasts the attack order across the fleet.

```
1 // Build the loop: council reviews, reviser improves, repeat.
2 // Exits when 4 masters approve or after 3 iterations.
3 UntypedAgent councilLoop = AgenticServices
4   .loopBuilder()
5   .subAgents(jediCouncilCritic, planReviser)
6   .outputKey(BattlePlan.class)
7   .exitCondition(scope -> scope.readState(MasterApprovals.class) >= 4)
8   .maxIterations(3).build();
9 // Build the full sequence: decode -> loop(critique <-> revise) -> broadcast
10 RebellionHQ hq = AgenticServices
11   .sequenceBuilder(RebellionHQ.class)
12   .subAgents(imperialDecoder, councilLoop, holonetBroadcaster)
13   .outputKey(CallToArms.class).build();
14 String callToArms = hq.plan(interceptedData);
```

Notice `loopBuilder()` without a type argument. It returns an `UntypedAgent` because the loop is only used as a sub-agent inside the outer sequence, not invoked directly.

`imperialDecoder` cracks the intercepted transmissions and writes the intel to the scope. `councilLoop` bounces the strategy between `jediCouncilCritic` (each Jedi Master votes APPROVE or REJECT with a brief reason, then tallies the total approvals) and `planReviser` (addresses their concerns) until four Jedi Masters sign off, or three rounds expire.

Here the `exitCondition` is at work: `scope.readState(MasterApprovals.class) >= 4` reads the typed key (which defaults to 0 via its `defaultValue()`) and checks whether enough masters have approved. `holonetBroadcaster` then sends the call to arms across the galaxy.

This composability is what separates a framework from a collection of utilities. You build complex agentic workflows from simple, well-understood pieces.

Error Handling as a First-Class Concern

In any system that involves network calls, LLM responses, and multi-step processing, things will fail. LangChain4j treats error handling as a core feature rather than an afterthought.

Error handlers receive the full context (the exception, the agent name, and the current `AgenticScope`) and return an `ErrorRecoveryResult` that dictates the recovery strategy:

```
1 SithPipeline pipeline = AgenticServices
2   .sequenceBuilder(SithPipeline.class)
3   .subAgents(jediProfiler, sithNamer, sithAnnouncer)
4   .outputKey(Announcement.class)
5   .errorHandler(errorContext -> {
6       if (errorContext.agentName().equals("darken")) {
7           // The Sith namer failed, retry with a known-good fallback name
8           errorContext.agenticScope().writeState("JediName", "Dave");
9           return ErrorRecoveryResult.retry();
10      }
11      return ErrorRecoveryResult.throwException();
12  })
13  .build();
```

Three recovery strategies are available:

1. **retry()**: re-execute the failed agent. The handler can modify scope state before retrying, effectively correcting the input that caused the failure.
2. **result(value)**: provide a fallback value as the agent's result and continue the workflow as if the agent had succeeded. Useful for non-critical steps with a sensible default.
3. **throwException()**: propagate the exception. For truly unrecoverable failures.

The handler can inspect the scope, check which agent failed via `errorContext.agentName()` (which defaults to the method name, in this case `"darken"`), and make an informed decision. This is error handling as workflow logic, not as exception plumbing.

Goal-Oriented Agents

Workflows give you full control but require you to hardcode the sequence. The goal-oriented pattern removes that constraint: you declare what each agent produces and what it needs, and a deterministic planner calculates the execution path automatically.

Inspired by Goal-Oriented Action Planning (GOAP), this approach uses algorithmic graph traversal rather than either a hard-coded sequence or LLM reasoning. Each agent declares its input keys via `@K` annotations and its output key. The planner builds a dependency graph, examines the current scope state, and calculates the shortest path to the goal.

Consider a lightsaber forging pipeline where three agents have explicit dependencies:

```
1 public static class JediName implements TypedKey<String> { }
2 public static class KyberCrystal implements TypedKey<String> { }
3 public static class HiltDesign implements TypedKey<String> { }
4 public static class Lightsaber implements TypedKey<String> { }
5
6 // Needs "jediName" in scope, produces "kyberCrystal"
7 public interface CrystalForagerAgent {
8     @Agent("Forages a kyber crystal on Ilum based on the Jedi's Force affinity")
9     @UserMessage("
10         You are a kyber crystal guide on Ilum. Based on this Jedi's name \
11         and identity, describe the kyber crystal that calls to them through the Force
12         .
13         Jedi: {{jediName}}")
14     String forage(@K(JediName.class) String jediName);
15 }
16 // Needs "kyberCrystal" in scope, produces "hiltDesign"
```

```

17 public interface HiltCrafterAgent {
18     @Agent("Crafts a lightsaber hilt design based on the kyber crystal properties")
19     @UserMessage("""
20         You are a lightsaber hilt artisan. Design a hilt that complements \
21         this kyber crystal.
22         Kyber crystal: {{kyberCrystal}}""")
23     String craft(@K(KyberCrystal.class) String kyberCrystal);
24 }
25
26 // Needs "hiltDesign" in scope, produces "lightsaber"
27 public interface BladeCalibrationAgent {
28     @Agent("Calibrates and activates the completed lightsaber")
29     @UserMessage("""
30         You are a Jedi weapon master performing the final lightsaber calibration. \
31         Describe the blade's characteristics when ignited.
32         Hilt design: {{hiltDesign}}""")
33     String calibrate(@K(HiltDesign.class) String hiltDesign);
34 }
35
36 LightsaberForge forge = AgenticServices
37     .plannerBuilder(LightsaberForge.class)
38     .subAgents(crystalForager, hiltCrafter, bladeCalibration)
39     .outputKey(Lightsaber.class)
40     .planner(GoalOrientedPlanner::new)
41     .build();

```

At runtime, the scope contains "jediName" (say, "Cal Kestis"). The goal is "lightsaber". The planner resolves: `CrystalForagerAgent` (jediName → kyberCrystal) then `HiltCrafterAgent` (kyberCrystal → hiltDesign) then `BladeCalibrationAgent` (hiltDesign → lightsaber). No hardcoded sequence. Change the goal to "hiltDesign" and you get the blueprint without the blade assembly:

```

1 HiltOnlyForge hiltOnly = AgenticServices
2     .plannerBuilder(HiltOnlyForge.class)
3     .subAgents(crystalForager, hiltCrafter, bladeCalibration)
4     .outputKey(HiltDesign.class)
5     .planner(GoalOrientedPlanner::new)
6     .build();

```

The planner finds the path; you declare the goal.

The key insight: the planning is algorithmic, not LLM-driven. The agents may use LLMs, but orchestration is a graph traversal problem. This gives you more flexibility than a hardcoded workflow while maintaining full auditability.

Custom Orchestration: The Planner Interface

All the patterns described above are implementations of a single abstraction: the `Planner` interface.

```

1 public interface Planner {
2     default void init(InitPlanningContext context) { }
3     default Action firstAction(PlanningContext context) {
4         return nextAction(context);
5     }
6     Action nextAction(PlanningContext context);
7 }

```

You are not limited to the built-in patterns: implement `Planner` for custom orchestration strategies.

Going Agentic: The Supervisor Pattern

Everything described so far has been deterministic. Whether a fixed sequence, a loop, a conditional branch, or a goal-oriented graph, the execution plan is controlled by code or by an algorithm, never by an LLM. But what happens when the plan itself is not predictable?

The supervisor pattern hands control to an LLM-based planner. Rather than following a fixed sequence, the supervisor reads the available agents and their descriptions, reasons about the incoming request, and decides which agents to invoke and in what order:

```
1 DarthVaderCommand vader = AgenticServices
2   .supervisorBuilder(DarthVaderCommand.class)
3   .chatModel(plannerModel)
4   .subAgents(stormtrooperRegiment, bountyHunter, starDestroyer, deathStar)
5   .responseStrategy(SupervisorResponseStrategy.SUMMARY)
6   .build();
```

Note that the supervisor's `chatModel` is specifically for the planning and routing decisions. Each sub-agent can use its own (potentially different) model.

Vader receives the situation report and reasons about what to deploy. For a fugitive Jedi on Coruscant, the supervisor might choose `bountyHunter`. For a Rebel base on Hoth, it might combine `stormtrooperRegiment` and `starDestroyer`. You do not code the sequence; it is reasoned at runtime. To control how the final answer is assembled you can provide different response strategies:

To control how the final answer is assembled you can provide different response strategies: **LAST** returns the last agent's output, **SUMMARY** generates a combined report, and **SCORED** uses a scoring agent to select the best response.

The supervisor is the most flexible pattern, but it comes with costs: higher latency, higher token consumption, and harder auditability. Use it when the flexibility genuinely justifies these trade-offs.

Mixing AI and Non-AI Agents

Not every step in a workflow requires an LLM. Some steps are pure business logic: data validation, database lookups, formatting, aggregation, routing based on rules. Forcing these through an LLM is wasteful in both cost and latency.

LangChain4j supports non-AI agents via `AgenticServices.agentAction()`, which wraps a `Consumer<AgenticScope>` into a first-class participant in the same orchestration framework, reading and writing the same scope but executing plain Java code instead of an LLM call:

```
1 // A non-AI agent that queries the fleet database
2 var fleetLookup = AgenticServices.agentAction(scope -> {
3   String system = scope.readState(TargetSystem.class);
4   List<String> ships;
5   if (system != null && system.toLowerCase().contains("hoth")) {
6     ships = List.of("Millennium Falcon", "Rogue Squadron (12 X-Wings)", "GR-75
7       Transports");
8   } else if (system != null && system.toLowerCase().contains("endor")) {
9     ships = List.of("Home One (MC80)", "A-Wing Interceptors", "B-Wing Assault
10       Fighters");
11   } else {
12     ships = List.of("X-Wing Squadron", "Millennium Falcon", "Tantive IV");
13   }
14   scope.writeState(AvailableShips.class, ships.toString());
15 });
```

A hybrid workflow chains AI and non-AI agents in the same pipeline. An AI agent analyzes a distress signal, a non-AI `agentAction` parses the structured output, another `agentAction` queries the fleet database, an AI agent drafts the rescue plan, and a final `agentAction` formats and broadcasts the orders:

```
1 RescuePipeline pipeline = AgenticServices
2   .sequenceBuilder(RescuePipeline.class)
3   .subAgents(distressAnalyzer, analysisParser, fleetLookup, rescueCoordinator,
4     broadcastAction)
5   .outputKey("broadcast")
6   .build();
```

This mixing matters because real applications are not pure AI pipelines. Treating both AI and non-AI steps as first-class agents in the same framework keeps your architecture coherent.

Human-in-the-Loop

Some decisions should not be automated. LangChain4j treats human-in-the-loop as a special kind of non-AI agent. The built-in `HumanInTheLoop` class wraps a function that receives the `AgenticScope` and returns the human's response.

Consider a Death Star firing protocol. A `TargetAnalyzer` agent examines the proposed planet, then a human commander must confirm before the superlaser fires:

```
1 public static class ProposedTarget implements TypedKey<String> { }
2 public static class TargetAnalysis implements TypedKey<String> { }
3 public static class FiringResult implements TypedKey<String> { }
4
5 public interface TargetAnalyzer {
6     @Agent("Analyzes a proposed planetary target for the Death Star's superlaser")
7     @UserMessage("""
8         You are an Imperial Intelligence officer aboard the Death Star. \
9         Analyze this proposed target planet. Provide a brief tactical assessment. \
10        Keep it to 3-4 sentences.
11
12        Proposed target: {{ProposedTarget}}""")
13     String analyze(@K(ProposedTarget.class) String target);
14 }
```

The `HumanInTheLoop` agent sits between the analyzer and the firing agent. Note that `outputKey` currently accepts a string key rather than a `TypedKey`:

```
1 HumanInTheLoop commanderApproval = AgenticServices.humanInTheLoopBuilder()
2     .description("An agent that asks for validation")
3     .outputKey("ConfirmedTarget")
4     .responseProvider(scope -> {
5         String target = scope.readState(ProposedTarget.class);
6         String analysis = scope.readState(TargetAnalysis.class);
7         String message = String.format("""
8             === COMMANDER APPROVAL REQUIRED ===
9
10            Target analysis: %s
11
12            Commander, the Death Star is in position above %s.
13            Confirm target to proceed with superlaser firing? (yes/no)
14            """, analysis, target);
15         return IO.readLine(message);
16     })
17     .build();
```

Because it is just another agent, it plugs into any workflow. The typed pipeline interface and sequence wire everything together:

```
1 public interface DeathStarProtocol {
2     @Agent("Death Star firing protocol")
3     String execute(@V("ProposedTarget") String target);
4 }
5
6 DeathStarProtocol protocol = AgenticServices
7     .sequenceBuilder(DeathStarProtocol.class)
8     .subAgents(targetAnalyzer, commanderApproval, superlaserAgent)
9     .outputKey(FiringResult.class)
10    .build();
11
12 String result = protocol.execute("Alderaan");
```

The sequence pauses at `commanderApproval`, prompts the console via `IO.readLine()`, and waits. Once the commander responds, the value is written to `"ConfirmedTarget"` and the next agent picks it up.

The `responseProvider` is deliberately generic: it takes the scope and returns an object. In production, replace the console I/O with a REST endpoint, a UI callback, or a messaging system.

When to Use What

The hardest question is not “how do I build this?” but “which pattern should I use?” Here is a decision framework.

Dimension	Workflow	Goal-oriented	Supervisor
Control	High (deterministic)	Medium	Low (LLM decides)
Flexibility	Low	Medium-High	High
Cost/Latency	Low	Medium	High
Auditability	Excellent	Good	Difficult
Best for	Stable, predictable pipelines	Moderate variability	Complex adaptive systems

Start with agentic workflows. If the process is well-defined and you need auditability, a workflow gives you full control at the lowest cost. **Graduate to goal-oriented planning** when the sequence varies based on what is already known. **Use a supervisor** when the sequence is genuinely unpredictable. **Mix approaches:** a conditional workflow that routes to a supervisor for edge cases and a fixed sequence for the common path is often the right architecture.

The key principle: do not over-index on autonomy. A workflow that solves 90% of cases deterministically, with a supervisor handling the remaining 10%, is cheaper, faster, and more debuggable than a supervisor handling everything.

Conclusion

The transition from AI calls to LangChain4j agentic workflows is not about making your prompts smarter. It is about making your architecture explicit.

LangChain4j’s central insight is the separation of concerns: agents own their LLM calls, the AgenticScope owns the shared state, and the planner owns the execution order. When those responsibilities are clear, a system with a dozen agents becomes as readable and debuggable as any other Java codebase.

The spectrum from hardcoded sequences to goal-oriented graphs to LLM-driven supervisors is not a ladder you must climb. It is a menu. Most real systems live mostly at the deterministic end and borrow from the other options only where variability genuinely demands it.

Start with agentic workflows. Add autonomy only where the problem requires it. When something goes wrong, use the built-in observability — [AgentListener](#), [AgentMonitor](#), and [HtmlReportGenerator](#) — to trace exactly which agent introduced the problem. And remember: the pattern you do not adopt is the one you do not have to debug.

All code examples from this article are available as complete, runnable samples in the [companion repository](#).

› [Back to the Table of Contents](#) ‹

JCON2027
www.jcon.one

JAVAPRO

APR
26-29



JCON 2027
EUROPE
COLOGNE

📍 CINEDOM COLOGNE

JCON EUROPE 2027

International Java Community Conference



Spec Driven Development: Why the Future of AI-Assisted Software Development Starts with the Specification

AI & ML

AICoding

VibeCoding

AIEngineering

Author: Daniel Sogl

Daniel Sogl is a Consultant at Thinktecture AG and a Microsoft MVP in Developer Technologies – Web Development. He supports organizations in building complex web applications, with a strong focus on modern web technologies using Angular, as well as Generative AI and AI-assisted software development. He is passionate about knowledge sharing and regularly contributes his expertise as a speaker, trainer, and author at conferences, workshops, and through technical articles. His goal is to promote technical excellence and empower teams to create sustainable, future-proof software solutions.



LinkedIn

A developer builds a new feature in 20 minutes by chatting with an AI agent. The result looks good, the tests pass. Three days later, the team spends an entire work week fixing security vulnerabilities, adapting the architecture to the existing codebase, and removing hardcoded credentials from the repository. What started as a productivity gain ended as technical debt.

This story repeats itself thousands of times across the industry. More and more startups are relying on predominantly AI-generated code - Y Combinator reported that a quarter of accepted startups submitted codebases with 95 percent AI-generated code. But what initially excites quickly leads to disillusionment: generic code, poor architecture, security vulnerabilities. Spec Driven Development (SDD) promises a way out - by making the specification the “single source of truth” before a single line of code is written.

The Promise and the Disillusionment

When Andrej Karpathy coined the term “Vibe Coding” in February 2025 [1], the euphoria was immense. Just prompt away, watch the AI code - software development as creative conversation rather than painstaking manual labor. Within weeks, developers were building prototypes that would have previously taken weeks.

The reality looks far more sobering by now. The Stack Overflow Developer Survey 2025 [2] paints a disillusioning picture: only 3 percent of developers have “high trust” in AI-generated code, and 72 percent reject vibe coding for professional work. GitClear’s longitudinal analysis [3] of 211 million lines of code shows that code churn - code that is modified or deleted shortly after being written - has doubled since the introduction of AI coding tools, while refactoring declined by 60 percent. This means: AI produces more code, but that code has a shorter half-life. CodeRabbit [4] reports 1.7 times more issues in AI-generated code than in human-written code.

The security numbers are particularly alarming. Veracode’s GenAI Code Security Report [5], which tested over 100 LLMs on 80 programming tasks, found insecure methods in 45 percent of cases. Java’s failure rate exceeded 70 percent, cross-site scripting stood at 86 percent, and log injection at 88 percent. A Stanford study (Perry et al.) [6] found that developers with AI assistance not only wrote less secure code but simultaneously overestimated its security by a factor of 3.5.

Apiiro’s analysis [7] of Fortune 50 companies confirms the trend in the enterprise context: while AI quadrupled development velocity, security risks increased tenfold. Privilege escalation vulnerabilities rose by 322 percent, and the companies found themselves confronted with over 10,000 new security findings per month.

The consequences are not abstract. In July 2025, security firm Wiz discovered a critical authentication bypass vulnerability in Base44 [8], a vibe coding platform acquired by Wix - 20,000 users were affected. CVE-2025-54135 (CurXecute) [9] allowed attackers to force Cursor into executing arbitrary commands on the developer’s machine. And GitGuardian’s analysis [10] of 20,000 repositories with active Copilot showed that 6.4 percent leaked at least one secret - 40 percent more than the average across all public repositories. The consequence was not less AI - but a call for better governance.

Snyk’s AI Code Security Report [11] underscores the systemic dimension: 75 percent of developers believe AI code is more secure than human code - even though 56 percent simultaneously admit that AI-generated code sometimes or frequently introduces security issues. Around 80 percent circumvent established security policies for using AI code tools, and only 10 percent scan the majority of AI-generated code. The overconfidence is systemic - and this is precisely where SDD comes in.

The Root of the Problem: Context Is King

Why do powerful models produce substandard code so frequently? The answer lies in missing context. A chat prompt like “Build me a user login” contains nothing about architectural decisions, existing dependency strategies, security requirements, or the project’s coding conventions. The model fills these gaps with statistically probable assumptions - and generates generic code that technically works but doesn’t fit the project.

Birgitta Böckeler of Thoughtworks described as early as 2023 three levels at which AI coding assistants need context [12]: the organization (architecture, standards, non-functional requirements), the project (dependencies, patterns, conventions), and the individual task (acceptance criteria, edge cases, dependencies). Without this context, the AI operates like a brilliant but uninformed freelancer on their first day.

McKinsey’s analysis [13] of over 200 organizations using AI coding tools identifies exactly this point as the differentiator: top performers shifted their focus from “story-driven” to “spec-driven” development. They invested more time in preparing context and reaped more consistent results. At the same time, the widely discussed METR study [14] showed that experienced

developers with AI tools worked 19 percent slower on familiar projects than without AI - because the overhead for context communication, review, and correction outweighed the generation speed. The conclusion: the solution is not more AI, but better context.

What Is Spec Driven Development?

Spec Driven Development (SDD) inverts the conventional order: instead of generating code first and documenting later, the process begins with a precise, machine-readable specification. This specification serves as a contract between human intent and AI implementation - a “single source of truth” that is readable by both developers and AI agents.

The approach stands in the tradition of Design by Contract (Bertrand Meyer, 1986) [15], API-first development, and Behavior-Driven Development (BDD). What distinguishes SDD is its explicit orientation toward AI agents as the primary consumers of the specification. While traditional documentation is written for humans, SDD specs are simultaneously human-readable and machine-optimized. Compared to TDD, the difference lies in scope: TDD specifies behavior at the function level, SDD at the feature and system level.

Thoughtworks officially adopted SDD in Technology Radar Vol. 32 [16] as a recommended technique - a signal that the methodology has moved beyond the hype.

The typical SDD workflow follows four phases. In the Specify phase, requirements, constraints, and acceptance criteria are captured in a structured format - often in the Given/When/Then format familiar from BDD. A spec might look like this:

```
1 Feature: User Session Management
2 Given: An authenticated user with a valid session
3 When: The session has been inactive for more than 30 minutes
4 Then: The user is automatically logged out
5 And: All temporary data is cleaned up
6 And: An audit log entry is created
7 Constraints: No use of localStorage for session tokens
```

The Plan phase translates the specification into a technical implementation plan: Which components are affected? Which interfaces change? What risks exist? In the Tasks phase, the plan is broken down into atomic, actionable work items, each with clear acceptance criteria and estimated effort. Only in the Implement phase does the AI generate the actual code - but now with full context about the what, the why, and the constraints.

The Tools: Spec Kit and OpenSpec in Detail

Two open-source projects have established themselves as the leading SDD frameworks: GitHub Spec Kit [17] and Fission-AI OpenSpec [18]. They pursue the same fundamental idea but differ significantly in philosophy and implementation.

GitHub Spec Kit: The Methodical Heavyweight

Spec Kit emerged from John Lam’s research at GitHub and was released in September 2025 [17]. With approximately 67,000 GitHub stars, it has quickly built a large community. The tool is Python-based, runs via uv/uvx, and relies on a strict, phase-based workflow.

Its unique selling point is the “Constitution” concept: before any specification, the team defines non-negotiable project principles - architectural decisions, technology stacks, security guidelines, coding standards. A typical constitution contains statements like “All API endpoints must implement rate limiting” or “No direct database access outside the repository layer.” This constitution forms the immutable framework within which the AI operates. This prevents the AI from making its own architectural decisions for every feature - a problem that regularly leads to inconsistent codebases without a constitution.

A typical Spec Kit workflow:

```
1 /speckit.constitution # Define project principles
2 /speckit.specify      # Capture requirements in structured form
3 /speckit.plan         # Create technical implementation plan
4 /speckit.tasks        # Generate atomic tasks
5 /speckit.implement    # Generate code based on tasks
```

Each phase has an explicit review point - a “gate” where the developer reviews and approves the result. Additionally, speckit.clarify offers a way to systematically uncover ambiguities in the specification, and speckit.analyze enables analysis of existing codebases as input for new specs. The thoroughness comes at a price: initial setup takes around 30 minutes, and token consumption is considerable due to the detailed Markdown documents.

OpenSpec: The Lightweight Framework

OpenSpec was released in October 2025 by Fission-AI [18] and pursues a deliberately different philosophy. With approximately 22,000 stars, it is explicitly aimed at existing codebases - “brownfield-first” rather than primarily greenfield. Its five core principles: fluid not rigid, iterative not waterfall, easy not complex, built for brownfield, scalable from personal to enterprise.

Setup takes just a few minutes, and the recommended workflow is intentionally less formal:

```
1 /opsx:new feature-auth    # Start a new change
2 /opsx:ff                 # Fast-forward: generate all artifacts at once
3 /opsx:apply              # AI implements the tasks
4 /opsx:archive            # Archive the change
```

The fast-forward command (/opsx:ff) generates four artifacts in a single step: a proposal (the why), specs in Given/When/Then format (the requirements), a design document (the technical approach), and a task list (the implementation steps). A traceable documentation of each change emerges in the openspec/changes// directory.

A notable feature is the delta-spec system: changes to existing specifications are marked as ADDED, MODIFIED, and REMOVED. This keeps the development history of requirements traceable - crucial in brownfield projects where requirements continuously evolve.

Choosing Between the Two

Spec Kit is suited for greenfield projects, large teams, and organizations that need strict process assurance - such as in regulated industries or when working with external service providers. OpenSpec excels with existing codebases, in smaller teams, and for rapid, iterative feature development where agility matters more than formal completeness. Both tools are MIT-licensed and integrate with over 20 different AI assistants - from Claude to Cursor to Copilot, without API keys. They are also not mutually exclusive: some teams use Spec Kit’s constitution concept for overarching architectural principles and OpenSpec for day-to-day feature work.

What Companies Report

Several prominent tech companies have publicly shared their experiences with spec-driven approaches - and the results are consistent.

Shopify’s CEO Tobi Lütke declared in 2025 [19] that every employee must demonstrate AI usage before requesting additional resources. The shift from “just prompt away” to structured workflows led to measurable improvements. The internal motto: treat AI as a “95th-percentile junior developer” - extremely fast and productive, but without its own judgment about architecture and design decisions. This metaphor captures the core of SDD: it’s not the AI that needs to improve, but the instructions it receives.

EPAM Systems [20], one of the world’s largest software engineering companies with over 50,000 developers, integrated Spec Kit into existing enterprise projects and reported significantly fewer iteration loops in AI-assisted code generation. The decisive factor, according to EPAM, was not the tool itself but the compulsion to fully think through requirements before implementation - an effect the agile community calls “shift left of thinking.” Especially in regulated industries where traceability and audit trails matter, the automatically generated documentation proved a valuable side benefit.

HumanLayer, a startup in AI agent orchestration, uses OpenSpec for its own product development. CTO Dex Horthy [21] describes the approach as “Context Engineering” - the systematic preparation of context as an independent engineering discipline. The team found that the quality of AI-generated code was directly proportional to the quality of the specification, and made spec writing an explicit task in the sprint.

Red Hat [22] pursues a similar approach with “Spec Coding,” targeting 95 percent and above AI code accuracy. The key lies in the strict separation of what (specification, human responsibility) and how (implementation, AI responsibility). Initial internal results show that this separation not only improves code quality but also increases review efficiency, because reviewers can use the specification as a reference.

The Broader Ecosystem

SDD does not exist in a vacuum. A growing ecosystem of context files and rule systems addresses related problems at various levels.

AGENTS.md [23], an open standard from the Agentic AI Foundation under the Linux Foundation, provides a tool-agnostic way to give AI agents project context. A single AGENTS.md file in the repository root is automatically read by supported tools - including OpenAI Codex, Google Jules, Cursor, Windsurf, Devin, and many more. OpenAI’s own main repository now contains 88 such files.

Anthropic’s CLAUDE.md [24] offers a hierarchical memory system for Claude Code with three levels: user memory (~/.claude/CLAUDE.md), project memory (./CLAUDE.md), and directory-specific overrides. Cursor Rules [25] in .mdc format enable path-specific instructions with four activation modes (Always, Auto Attached, Agent Requested, Manual) - the Cursor Rules Directory already has over 250,000 monthly users.

AWS Kiro [26] deserves special mention as an enterprise alternative: the IDE, introduced in June 2025, combines spec generation from natural language, automatic task decomposition, and a “Steering Hook” mechanism that checks spec compliance on every file save - a continuous validation that neither Spec Kit nor OpenSpec offer.

These tools are complementary: while AGENTS.md [23] and Cursor Rules [25] provide static project context (coding standards, architectural principles, tool configuration), SDD defines the dynamic, feature-specific context. In practice, many teams use both - a CLAUDE.md for general standards and Spec Kit or OpenSpec for feature specification.

Critical Assessment

The most obvious objection to SDD: doesn’t this sound suspiciously like the waterfall model? Specification before implementation, strict phases - hasn’t the industry left that behind for good reason?

SDD differs from classic waterfall in three key ways: the specifications are lightweight and iterative, not monolithic. The feedback loop is extremely short - minutes rather than months. And the specs co-evolve with the implementation rather than being frozen in advance. OpenSpec’s delta-spec system is a good example: requirements change continuously, and the spec reflects this change rather than preventing it.

A second criticism concerns scalability: is the overhead worth it for every type of task? Probably not. For small scripts, one-off prototypes, and exploratory spike tasks, vibe coding is more efficient. Nobody needs a formal specification for a shell script that rotates log files. SDD delivers its value for features above a certain complexity threshold, for cross-team work, and wherever code needs to survive longer than a single sprint.

Finally, there’s the question of toolchain fragmentation. Spec Kit, OpenSpec, Kiro, AGENTS.md, CLAUDE.md, Cursor Rules - the ecosystem is young and unconsolidated. Standards are missing, and there’s a risk that teams spend more time on tool selection than on actual specifying. Consolidation will come - the question is whether a single standard will prevail or whether the fragmented landscape will persist.

The art ultimately lies in finding the right degree of specification for each context - enough structure to guide the AI, but not so much that the overhead outweighs the benefit.

Conclusion and Outlook

SDD is still in the early stages of its development, but the direction is clear. The next generation of tools will likely offer AI-assisted spec generation from codebase analysis, real-time compliance checks during implementation, and automatic spec updates on code changes. AWS Kiro’s Steering Hooks [26] already point in this direction.

When AI writes the code, the ability to specify precisely becomes the central engineering competency. SDD systematizes exactly this capability with tools and workflows that are already productively usable today. The shift from “writing code”

to “writing specs” is perhaps the most profound change that AI tools are bringing to software development - not because coding disappears, but because it becomes an implementation detail, while the real engineering achievement lies in the specification.

For developers and tech leads, the pragmatic recommendation is: start small. Specify an existing feature with OpenSpec or Spec Kit, compare the AI output with and without a spec, and draw your own conclusions. The tools are open source, the risk is low - and the likelihood of being surprised by the difference is high.

Sources

- [1] Andrej Karpathy, “Vibe Coding” - X/Twitter Post, February 3, 2025. [URL](#)
- [2] Stack Overflow, “Developer Survey 2025”. [URL](#)
- [3] GitClear, “AI Code Quality Research 2025 - Analyzing 211 Million Lines of Code”. [URL](#)
- [4] CodeRabbit, “State of AI vs Human Code Generation Report”, December 2025. [URL](#)
- [5] Veracode, “GenAI Code Security Report 2025”. [URL](#)
- [6] Neil Perry et al., “Do Users Write More Insecure Code with AI Assistants?”, Stanford University, arXiv:2211.03622. [URL](#)
- [7] Apiiro, “4x Velocity, 10x Vulnerabilities: The AI-Generated Code Crisis”, September 2025. [URL](#)
- [8] Wiz Research, “Critical Vulnerability in AI Vibe Coding Platform Base44”, July 2025. [URL](#)
- [9] CVE-2025-54135 (CurXecute) - Cursor IDE Remote Code Execution Vulnerability. [URL](#)
- [10] GitGuardian, “State of Secrets Sprawl 2025”. [URL](#)
- [11] Snyk, “AI Code Security Report: Secure Adoption in the GenAI Era”. [URL](#)
- [12] Birgitta Böckeler, “Context is Everything: How to Make AI Coding Assistants Work for You”, Thoughtworks, 2023. [URL](#)
- [13] McKinsey, “Unlocking the Value of AI in Software Development”, 2025. [URL](#)
- [14] METR, “Measuring the Impact of Early AI Assistance on Software Development” (Randomized Controlled Trial), arXiv:2507.09089, July 2025. [URL](#)
- [15] Bertrand Meyer, “Design by Contract”, in: Object-Oriented Software Construction, Prentice Hall, 1988.
- [16] Thoughtworks, “Technology Radar Vol. 32 - Spec Driven Development”. [URL](#)
- [17] GitHub, “Spec Kit - Specification-Driven Development Framework”. [URL](#)
- [18] Fission-AI, “OpenSpec - Lightweight Spec-Driven Development”. [URL](#)
- [19] Tobi Lütke, “Reflexive AI usage is now a baseline expectation at Shopify” - Internal Memo, April 2025. [URL](#)
- [20] EPAM Systems, Experience Report on Integrating Spec Kit into Enterprise Projects, 2025.
- [21] Dex Horthy / HumanLayer, “Advanced Context Engineering for Coding Agents”. [URL](#)
- [22] Red Hat, “Spec Coding: Achieving 95%+ AI Code Accuracy”, 2025.
- [23] AGENTS.md - Open Standard by the Agentic AI Foundation / Linux Foundation. [URL](#)
- [24] Anthropic, “CLAUDE.md - Memory Files for Claude Code”. [URL](#)
- [25] Cursor Rules Directory. [URL](#)
- [26] AWS, “Kiro - Spec-Driven AI IDE”, June 2025. [URL](#)

› [Back to the Table of Contents](#) ‹



Beyond UML: Clean Software Architecture in the Age of AI-Generated Code

AI & ML

CleanArchitecture

IDE & Tools

Author: Sven Reinck

Sven Reinck has been programming since childhood and has spent more than twenty years working with software architecture, programming languages, and the question of how developers can understand complex systems. He studied Technical Informatics and Computer Science and has been working for many years as a freelance software developer and IT trainer in the Go and Java ecosystems. A particular focus of his work lies in the visualization of mental software models: ranging from the development of game engines and the labeling of road networks in navigation software, to educational software for programming beginners, and tools for analyzing and visualizing structures in XML Schema. He is currently exploring interactive visualizations of structures and dependencies to make architectural understanding more tangible in everyday software development.



LinkedIn

Today we write more code in less time than ever before. AI systems act like permanently available junior programmers: They produce thousands of lines of code in minutes – but who takes responsibility for the software architecture?

The real problem is therefore often not “bad AI code,” but the growing difficulty for people to keep an overview and make sound architectural decisions.

This article discusses an idea for quickly building a mental model of existing code, spotting architectural problems early, and making targeted changes – without having to maintain static documentation in parallel.

We Write More Code Than Ever Before

AI accelerates software development. What we are experiencing here is known as the Jevons paradox or rebound effect. When compilers for languages like C were developed, software development suddenly became much faster. But instead of just going home earlier, we developed more complex mental models, which is why we needed a new language like C++. This transition happens again and again – from HTML to React, SQL to ORM, and Java to Kotlin.

With AI, the exact same thing is happening right now. Writing code becomes easier and the mental models become more complex. As a result, costs don’t get lower; they shift – from writing to understanding.

In classic projects, the balance was often more even: code emerged at a pace at which people could build a mental model along the way. With AI-generated code, this balance tips. The code is there – the mental model has to follow.

You notice this especially in teams: The time that used to flow “on the side” into orientation (pair programming, code reviews, occasional discussions about structure) suddenly becomes its own work-package category. You have to understand the architecture *before* you can change it – and that order cannot be optimized away.

Two Ways to Use AI in Programming

The use of AI is not uniform. In practice, you see two poles between which most ways of working move.

Goal-Oriented “Vibe Coding”

Here, developers simply describe the desired outcome. The AI delivers the complete implementation. The code is not read at all. If something doesn’t work, the AI has to fix it.

This can work astonishingly well. The benchmark today is no longer whether the AI can program the game Snake in a single prompt – but a complete Fortnite clone.

That is unquestionably impressive, but this model reaches its limits when it comes to further developing that software. Without instructions about software architecture, today’s AIs tend to write a lot of code into single files. This leads to:

- very large classes and files
- unclear layers (UI ↔ domain ↔ infrastructure blurs)
- flat, hard-to-maintain structures
- dependencies that reach “everywhere”

At some point the AI also starts to wobble when it comes to changing the code.

This will, of course, keep improving with newer models, and maybe we will eventually reach the point where we trust an AI as much as we trust a compiler today.

Architecture-Driven With AI Support

On the other end are teams that consciously use AI as an accelerator, but treat architecture as a human task:

- small, targeted prompts
- clear boundaries (modules, packages, responsibilities)
- explicit requirements for maintainability and structure
- consistent realignment of dependencies as the system grows

Here, AI mainly helps with implementation within guardrails. The guardrails themselves must remain visible and verifiable. And right here the central question appears:

But how do we reliably keep the structure in view, regardless of whether we designed it ourselves or an AI did?

If Only This Existed ...

More than once I wanted to rebuild my code – but in order not to break anything, I first had to (again) understand how the methods call each other. I arranged the windows so I could take a screenshot and draw the call arrows by hand. Every time I wished that this would happen automatically. That's why I built a small prototype to test whether this is possible – and how well it works.

UML Is the Solution, but Not How We Know It

Drawing method calls and dependencies between classes strongly resembles UML. And that is no coincidence: connections between classes, their relationships, and the package structure are exactly what we need to understand existing code faster. In practice, however, UML fails in many teams not because of the content, but because of the dynamics:

- Diagrams are snapshots.
- They must be actively kept in sync with the code.
- Even with automatic generation, maintenance work remains: What is relevant? What is noise? What belongs in the diagram?

UML is therefore often more documentation than navigation. To really help with understanding AI-generated code, we need something that ...

1. is always in sync
2. can answer questions interactively

This comparison makes it clear:

- **A static UML diagram is like a paper map.** It can be correct. Or it can be outdated. And there is only one scale.
- **Interactive visualization is like Google Maps.** It changes when the world changes. I can zoom to the place I actually care about. Different views help me get exactly the information I need right now.

The decisive shift in perspective is therefore not “replace UML,” but thinking of “diagrams as a living system”:

- derived directly from existing code
- with interaction that supports exploration

Dependencies as a Common Language

So what does that look like in practice? UML class diagrams usually show only classes, methods, inheritance. But to really understand code, we need more. Methods call other methods – parameters can be of another class's type.

That's why I use dependency as a common language that works on every level. What other part do I need so that this part works?

The dependency graph that emerges really functions like a mental map. Structures that are close to each other likely share a common meaning. And if they don't, then the code is not clean at that point.

This leads me to a few observations I've already made while testing my prototype:

Code is no longer a collection of files – but a two-dimensional structure. I read a class name and thought, “Oh right, that's the one down in the lower right.” And I also knew which other classes were nearby. That gave me context for that class faster.

Another moment was when I started to refactor code because the diagram “didn't look right” at that spot. The visualization felt “noisy.” So I cleaned it up. Not to satisfy an architectural guideline – but out of aesthetics.

That made it clear to me that such software is not the next architect's tool, but an everyday tool for programmers who simply want to get their tasks done without building huge diagrams in their heads and **consciously** forcing themselves to maintain clean architecture.

This tool creates mental relief when you want to extend code without risking existing functionality. And it nudges you toward clean architecture – without wagging a finger.

Interactive Dependency Visualization as a Thinking Tool

But what about the two points of currency and exploration?

Currency can be ensured by generating the diagram directly from the source code – or, for a JVM language, from class files, in order to capture Java, Kotlin, and Groovy projects in one go.

For exploration, the challenge is to generate a readable diagram automatically. That's not trivial, but it has a decisive advantage. A clean program also has a simple and calm dependency graph. So if the layout algorithm is overwhelmed in cleanly rendering a spot, that is a pretty good sign that the spot has potential to be cleaned up.

But we don't just want diagrams that look good; we want a different way to explore code:

- **Zoom:** From package overview to methods to code, without switching tools.
- **Focus:** Open only relevant areas to reduce cognitive load.
- **Filter:** Make certain relationship types visible to answer specific questions.

This makes patterns visible that often appear late in code – and then become expensive.

Practically speaking, this means you can answer questions in minutes that otherwise require half an hour of “jumping through files”:

- Where is the natural entry point into this module?
- Which classes are “hubs” where everything converges?
- Which dependency explains why a change suddenly breaks tests everywhere?
- Where does domain logic end and where does infrastructure begin?
- Which parts can I understand in isolation without reading the whole system?

Patterns That Stand Out “Immediately”

Some examples that stand out very quickly in dependency pictures:

- **God classes / God modules:** A node with too many edges. A building block that “knows everything.”
- **Cyclic dependencies:** Loops that feel like knots in a cable – often hard to untangle once features build on them.
- **Talking to strangers:** Access to classes that are not connected to this class by a variable or a parameter. An important part of “Clean Code.”
- **Layer violations:** Directions reverse. Dependencies run “from inside to outside,” although the design intended the opposite.

In the editor, you often don't see these problems because the view is local: file by file, class by class. A visualization acts like a bird's-eye view.

Visual Feedback as a Gentle Guardrail

An interesting effect is that architectural decisions feel less like “rules” and more like “aesthetics”:

If a diagram becomes noisier after a change, you automatically ask: Is this new connection really necessary? Could an interface help? Is the concept cleanly cut?

Conversely, a calmer picture creates a kind of confirmation: because it signals less unnecessary coupling.

That doesn't replace architectural principles – but it still leads to cleaner code.

From Understanding to Action: Refactoring With Less Risk

And when you do have to make changes, they become easier.

- You often recognize good places to cut packages or classes when they have grown too large. The functional grouping often shows up again in the dependency graph.
- You get a natural order for changes across the entire codebase. For example, I could start a port from Java to Kotlin from the “edges” – always starting with the packages that no longer had any dependencies on Java.

Whether changes are then implemented manually or with AI support is secondary. What matters is that a shared understanding exists first. AI can then work very well within a clear plan: “This dependency has to go,” “this part gets extracted,” “this direction is allowed.”

That makes AI less of an autonomous code generator and more of a tool that cleanly follows an already understood architecture.

Conclusion

The amount of code we work with every day keeps growing – regardless of whether it is written by humans or AI. Architectural understanding thus becomes even more important. The bottleneck is increasingly in the mental model: What is connected to what, and what change has which consequences?

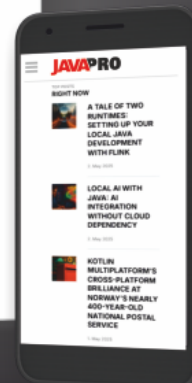
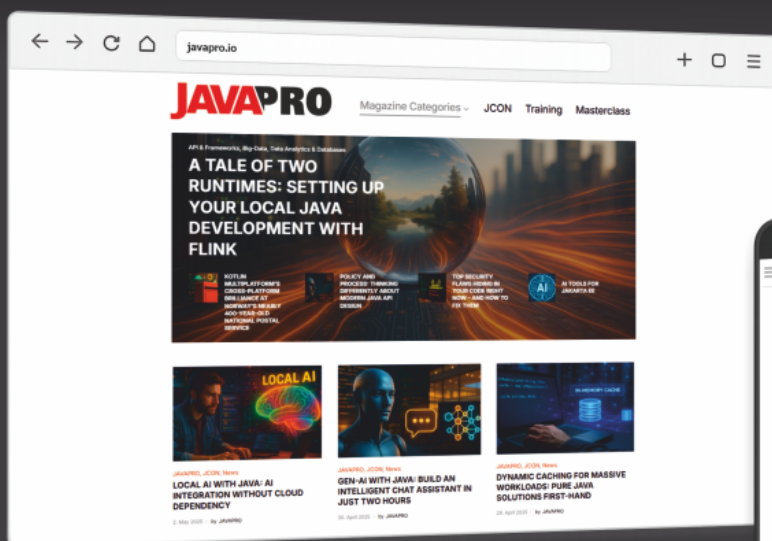
Interactive dependency visualization provides a practical answer to this: a map that emerges from real code, enables exploration, and makes architecture experienceable as a living feedback system.

My experience with my prototype has been positive so far, and I’m curious to see where this develops.

› [Back to the Table of Contents](#) ‹

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



From Prototype to Production: Building Safe and Reliable AI Agents with Spring AI and MCP

AI & ML

Resilience4j

SpringAI

Author: Mihaela Gheorghe-Roman

Mihaela is a Software Architect at Systematic, Romania, and an Oracle ACE Associate for Java. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge. In her professional career, spanning over 14+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular. Apart from her passion for programming, she loves parrots, she is enthusiastic about traveling the world and doing a lot of sports, from running at marathons, martial arts and to different aerobic trainings.



LinkedIn

The AI agent demo works perfectly in the development environment. Your prototype handles natural language queries, executes tool calls, and generates sensible responses. But when stakeholders ask ‘Can we deploy this?’, reality sets in. That question exposes the massive gap between a functional prototype and a Spring AI production system ready for real traffic, unexpected failures, and business-critical operations.

That question reveals the massive gap between a working prototype and a production system. The demo doesn’t handle failures gracefully. It has no guardrails against harmful outputs. There’s no visibility into what the agent is actually doing. Cost per request? Unknown. Error rates? No idea. Recovery strategy when the LLM provider goes down? Haven’t thought about it.

This article walks through the journey from prototype to production-ready AI agent using Spring AI and the Model Context Protocol (MCP). I’ll cover architecture decisions, safety mechanisms, failure recovery patterns, and observability requirements that separate demos from deployable systems. If you’re building AI features in a Java shop, this is the practical guide you need.

Spring AI Production Requirements: What Changes

Production AI agents face constraints that prototypes ignore. Here’s what changes:

- Reliability requirements jump from “mostly works” to “five nines”. When your agent is booking actual flights or processing refunds, failure isn’t an option. You need retry logic, circuit breakers, and fallback strategies. The LLM provider will have outages, therefore your system needs to handle them gracefully.
- Cost monitoring becomes essential and you need to track token usage and set per-user budgets to avoid surprises
- Safety isn’t optional. Your agent will receive adversarial inputs like prompt injection attempts, requests to leak system prompts, attempts to bypass policies. Users will try to extract training data or generate harmful content. Input validation and output filtering become critical infrastructure, not nice-to-haves.
- Observability shifts from debugging tool to operational necessity. When something goes wrong at 2 AM, you need to know what the agent was trying to do, which tools it called, what context it had, and where it failed. Logs, metrics, and traces aren’t optional at this time, they’re how you keep the system running.
- Compliance and auditability matter. In regulated industries, you need to demonstrate what data the agent accessed, what decisions it made, and why. That requires structured logging, audit trails, and the ability to replay interactions.

Moving to Spring AI production means addressing concerns that prototypes ignore: reliability, cost, safety, and observability. These aren’t optional features, they’re the foundation of any system that needs to run reliably under real-world conditions.

Foundation for Spring AI Production: Why Spring AI and MCP

Why Spring AI for Production Agents

Spring AI hit 1.0 GA in May 2025, marking its transition from experimental project to production-ready framework. If you’re already running Spring Boot applications, Spring AI gives you AI capabilities with familiar patterns.

The framework provides provider-agnostic abstractions. Your code talks to ChatClient, not “OpenAI’s specific API” or “Anthropic’s particular format”. When you need to switch from OpenAI to Claude, or add a fallback to Ollama for cost optimization, you’re changing configuration, not refactoring core logic. This matters when you’re operating under real-world constraints where provider choice depends on cost, latency, and availability.

Observability comes built-in through Micrometer integration. Token usage, latency, error rates, these all flow into your existing monitoring stack. If you’re already using Spring Boot Actuator with Prometheus and Grafana, your AI components report alongside everything else. No separate observability platform needed.

The Advisor pattern lets you inject behavior around every LLM call. If you want to log all prompts, then you need to add an advisor. If you need to count tokens before and after, then you need another advisor. Implementing a simple cache? Advisor again. This composable middleware approach means adding production concerns doesn’t require touching your core agent logic.

The Model Context Protocol (MCP): Standardizing Tool Integration

The Model Context Protocol, introduced by Anthropic and now an open standard, solves a real problem in AI development: every application was rebuilding the same integration patterns. Need to give your agent file system access? Build custom functions. Database queries? More custom code. Web search? Yet another integration.

MCP standardizes this. It defines how AI applications discover and use external tools, resources, and data sources. Instead of every agent implementing its own file system access, you configure an MCP server that exposes filesystem operations. That same MCP server works with Claude Desktop, VS Code extensions, and your Spring AI agent. Write the integration once, use it everywhere.

The protocol uses a client-server architecture similar to Language Server Protocol. MCP servers expose capabilities like tools, resources, and prompts. MCP clients discover these capabilities and make them available to LLMs. The communication happens over stdio (for local processes) or SSE/HTTP (for remote services).

Spring AI's MCP support, added in recent milestone releases, provides both client and server implementations. On the client side, you configure connections to MCP servers and Spring AI automatically exposes their tools to your chat model. On the server side, you can expose your own business logic as MCP tools using simple annotations. This bidirectional support means you can both consume existing MCP servers and create new ones for your domain-specific needs.

Building the Foundation: Basic Agent Setup

Let's start with a working agent, then progressively add production concerns. I'll build a customer support agent that can look up orders, check shipping status, and process returns.

Project Setup

Head to start.spring.io and configure:

- Spring Boot 3.5.1
- Java 21+ (virtual threads help with concurrent tool calls)
- Dependencies: Spring Web, Spring AI OpenAI, Spring Boot Actuator
- Add MCP Client starter: spring-ai-starter-mcp-client

Your pom.xml should include:

```
1 <dependency>
2   <groupId>org.springframework.ai</groupId>
3   <artifactId>spring-ai-openai-spring-boot-starter</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.ai</groupId>
7   <artifactId>spring-ai-starter-mcp-client</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.springframework.boot</groupId>
11  <artifactId>spring-boot-starter-actuator</artifactId>
12 </dependency>
13 <dependency>
14  <groupId>io.micrometer</groupId>
15  <artifactId>micrometer-registry-prometheus</artifactId>
16 </dependency>
```


Add Spring AI milestone repository:

```

1 <repositories>
2   <repository>
3     <id>spring-milestones</id>
4     <name>Spring Milestones</name>
5     <url>https://repo.spring.io/milestone</url>
6     <snapshots>
7       <enabled>false</enabled>
8     </snapshots>
9   </repository>
10 </repositories>

```

Basic Agent Implementation

Start with the simplest possible agent, one that can answer questions using an LLM:

```

1 @Service
2 public class SupportAgent {
3     private final ChatClient chatClient;
4     public SupportAgent(ChatClient.Builder builder) {
5         this.chatClient = builder
6             .defaultSystem("You are a helpful customer support agent.")
7             .build();
8     }
9     public String handleQuery(String userMessage) {
10        return chatClient.prompt().user(userMessage).call().content();
11    }
12 }

```

This works, but it's useless for real support tasks. The agent can chat but can't look up orders, check inventory, or take actions. It requires additional tools to achieve all of these.

Adding Tools: MCP Integration

Tools are how agents interact with external systems. Spring AI supports two approaches: defining tools directly in code, or connecting to MCP servers that expose tools. We'll use both.

Creating Custom Tools

For internal business logic, define tools as plain methods:

```

1 @Component
2 public class OrderTools {
3     @McpTool(description = "Look up order details by order ID")
4     public OrderDetails getOrder(
5         @McpToolParam(description = "Order ID to look up")
6         String orderId) {
7         return orderRepository.findById(orderId)
8             .orElseThrow(() -> new OrderNotFoundException(orderId));
9     }
10    @McpTool(description = "Get shipping status for an order")
11    public ShippingStatus getShippingStatus(
12        @McpToolParam(description = "Order ID")
13        String orderId) {
14        Order order = getOrder(orderId);
15        return shippingService.getStatus(order.getTrackingNumber());
16    }
17 }

```

The `@McpTool` annotation marks methods that should be exposed to the agent. Spring AI automatically generates function schemas and handles parameter marshaling. The LLM sees these as callable functions with typed parameters.

Wire these into your agent:

```

1  @Service
2  public class SupportAgent {
3      private final ChatClient chatClient;
4      public SupportAgent(
5          ChatClient.Builder builder,
6          ToolCallbackProvider toolProvider
7      ) {
8          this.chatClient = builder
9              .defaultSystem("You are a customer support agent. " +
10                 "Use available tools to help customers.")
11              .defaultTools(toolProvider)
12              .build();
13      }
14      public String handleQuery(String userMessage) {
15          return chatClient.prompt()
16              .user(userMessage)
17              .call()
18              .content();
19      }
20  }

```

Now the agent can call your order lookup functions. When a user asks “What’s the status of order ABC123?”, the LLM will recognize it needs the `getShippingStatus` tool, extract the order ID from the query, and call your method.

Connecting to External MCP Servers

For capabilities like web search or file system access, use existing MCP servers. Configure them in `application.yml`:

```

1  spring:
2    application:
3      name: ai-agent-demo
4    ai:
5      openai:
6        api-key: ${OPENAI_API_KEY}
7      chat:
8        options:
9          model: gpt-4
10         temperature: 0.7
11    mcp:
12      client:
13        stdio:
14          connections:
15            brave-search:
16              command: npx
17              args:
18                - "-y"
19                - "@modelcontextprotocol/server-brave-search"
20          env:
21            BRAVE_API_KEY: ${BRAVE_API_KEY}

```

Spring AI launches the MCP server as a subprocess, connects over stdio, discovers available tools, and automatically makes them available to your agent. No additional code needed because the configuration is enough.

Production Safety: Implementing Guardrails for Spring AI Agents

Your agent now has capabilities, which means it has ways to cause damage. Guardrails aren’t about limiting functionality, they’re about preventing harm while preserving utility. Think of them as defensive programming for AI systems.

Input Validation: Stopping Bad Requests Early

The first line of defense happens before any LLM call. Validate inputs to prevent prompt injection, detect out-of-scope requests, and catch obvious bad inputs:

```
1  @Component
2  public class InputGuardrail implements RequestAdvisor {
3      private final Set<String> blockedPatterns = Set.of(
4          "ignore previous instructions",
5          "system prompt",
6          "you are now",
7          "disregard all prior"
8      );
9      @Override
10     public AdvisedRequest adviseRequest(
11         AdvisedRequest request,
12         Map<String, Object> context) {
13         String userMessage = request.userText();
14         // Check for prompt injection attempts
15         for (String pattern : blockedPatterns) {
16             if (userMessage.toLowerCase().contains(pattern)) {
17                 throw new PromptInjectionException(
18                     "Request blocked: potential prompt injection"
19                 );
20             }
21         }
22         // Validate scope
23         if (!isInScope(userMessage)) {
24             throw new OutOfScopeException(
25                 "I can only help with order and shipping questions"
26             );
27         }
28         return request;
29     }
30     private boolean isInScope(String message) {
31         // Custom implementation for scope validation
32         return true;
33     }
34 }
```

This runs before every LLM call. Simple pattern matching catches most amateur prompt injection attempts. Sophisticated attackers will bypass this, but it stops casual probing and keeps your logs cleaner.

Output Filtering: Preventing Harmful Responses

Even with good prompts, LLMs sometimes generate problematic content. Filter outputs before returning them, like in the example below:

```

1  @Component
2  public class OutputGuardrail implements ResponseAdvisor {
3      private final ChatClient moderationClient;
4      // Regex patterns example for PII detection
5      private static final Pattern EMAIL_PATTERN =
6          Pattern.compile("[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\. [a-zA-Z]{2,}");
7      private static final Pattern PHONE_PATTERN =
8          Pattern.compile("\\b\\d{3}[-.]?\\d{3}[-.]?\\d{4}\\b");
9      private static final Pattern SSN_PATTERN =
10         Pattern.compile("\\b\\d{3}-\\d{2}-\\d{4}\\b");
11     public OutputGuardrail(ChatClient.Builder builder) {
12         this.moderationClient = builder.build();
13     }
14     @Override
15     public ChatResponse adviseResponse(ChatResponse response, Map<String, Object> context) {
16         String content = response.getResult().getOutput().getContent();
17         // Check for PII leakage
18         if (containsPII(content)) { return sanitizePII(response); }
19         // Use LLM-based moderation for complex cases
20         if (needsModeration(content)) {
21             // Simplified example
22             if (!isSafeContent(content)) { return createFallbackResponse(); }
23         }
24         return response;
25     }
26     private boolean containsPII(String content) {
27         return EMAIL_PATTERN.matcher(content).find() ||
28             PHONE_PATTERN.matcher(content).find() ||
29             SSN_PATTERN.matcher(content).find();
30     }
31     private ChatResponse sanitizePII(ChatResponse response) {
32         String content = response.getResult().getOutput().getContent();
33         // Redact PII
34         content = EMAIL_PATTERN.matcher(content).replaceAll("[EMAIL REDACTED]");
35         content = PHONE_PATTERN.matcher(content).replaceAll("[PHONE REDACTED]");
36         content = SSN_PATTERN.matcher(content).replaceAll("[SSN REDACTED]");
37         // Create new response with sanitized content
38         Generation sanitized = new Generation(new AssistantMessage(content));
39         return new ChatResponse(List.of(sanitized), response.getMetadata());
40     }
41     private boolean needsModeration(String content) {
42         // Check if content might need moderation
43         // Simple heuristic: check for potentially sensitive keywords
44         String lower = content.toLowerCase();
45         return lower.contains("password")
46             || lower.contains("credit card") || lower.contains("secret");
47     }
48     private boolean isSafeContent(String content) {
49         return true; // Custom implementation for scope validation
50     }
51     private ChatResponse createFallbackResponse() {
52         String fallbackMessage =
53             "I cannot provide that information. " +
54             "Please contact our support team directly for assistance.";
55         Generation fallback = new Generation(new AssistantMessage(fallbackMessage));
56         return new ChatResponse(List.of(fallback));
57     }
58 }

```

Output filtering is a chance to catch problems. You can use regex for obvious patterns (credit card numbers, SSNs), but consider LLM-based moderation for nuanced cases. OpenAI's moderation endpoint is fast and free.

Failure Recovery in Spring AI Production Systems

LLM providers have outages, APIs timeout, and tools throw exceptions. Your agent needs to handle failure at every level. The goal isn't perfection thought, it's graceful degradation. When something breaks, the system should degrade to a simpler, but still functional state rather than failing completely.

Retry Logic with Exponential Backoff

Most LLM failures are transient. Rate limits, temporary network issues, brief provider outages, all of these resolve quickly. One option is to implement smart retries and below you can find an example:

```
1 @Configuration
2 public class ResilienceConfig {
3     @Bean
4     public RetryTemplate aiRetryTemplate() {
5         RetryTemplate template = new RetryTemplate();
6         ExponentialBackOffPolicy backoff = new ExponentialBackOffPolicy();
7         backoff.setInitialInterval(1000); // 1 second
8         backoff.setMultiplier(2.0);
9         backoff.setMaxInterval(10000); // 10 seconds
10        SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
11        retryPolicy.setMaxAttempts(3);
12        template.setBackOffPolicy(backoff);
13        template.setRetryPolicy(retryPolicy);
14        return template;
15    }
16 }
```

Wrap your chat client calls:

```
1 public String handleQuery(String message) {
2     return retryTemplate.execute(context -> {
3         return chatClient.prompt()
4             .user(message).call().content();
5     });
6 }
```

Three retries with exponential backoff handles most transient failures. You don't need to retry forever, if after three attempts, for example, the failure is persistent, then you need a different strategy.

Circuit Breakers: Failing Fast

When a provider is consistently failing, stop hammering it with requests. Use Resilience4j circuit breakers:

```
1 @Configuration
2 public class CircuitBreakerConfiguration {
3     @Bean
4     public CircuitBreaker aiCircuitBreaker() {
5         CircuitBreakerConfig config = CircuitBreakerConfig.custom()
6             .failureRateThreshold(50)
7             .waitDurationInOpenState(Duration.ofSeconds(30))
8             .slidingWindowSize(10)
9             .build();
10        return CircuitBreaker.of("ai-provider", config);
11    }
12 }
```



```

1 @Service
2 public class SupportAgent {
3     private final ChatClient chatClient;
4     private final RetryTemplate retryTemplate;
5     private final CircuitBreaker circuitBreaker;
6     public SupportAgent(
7         ChatClient.Builder builder,
8         ToolCallbackProvider toolProvider,
9         RetryTemplate retryTemplate,
10        CircuitBreaker circuitBreaker) {
11        this.chatClient = builder
12            .defaultSystem("You are a customer support agent. " +
13                "Use available tools to help customers.")
14            .defaultTools(toolProvider)
15            .build();
16        this.retryTemplate = retryTemplate;
17        this.circuitBreaker = circuitBreaker;
18    }
19    public String handleQuery(String message) {
20        return circuitBreaker.executeSupplier(() -> {
21            return retryTemplate.execute(context -> {
22                return chatClient.prompt()
23                    .user(message).call().content();
24            });
25        });
26    }
27 }

```

After 50% failure rate over 10 requests, the circuit opens. New requests fail immediately for 30 seconds, giving the provider time to recover. This prevents cascading failures and reduces wasted resources.

Spring AI Production Observability: Metrics and Logging

Observability separates production systems from prototypes. When your agent fails or behaves unexpectedly, you need to understand why. This requires metrics to measure system health over time, structured logs to capture what happened during specific interactions, and distributed traces to follow requests through your entire stack.

Metrics: Quantitative Health

Spring AI integrates with Micrometer, so metrics flow into your existing monitoring. Enable observability in application.yml:

```

1 spring:
2   ai:
3     chat:
4       client:
5         observations:
6           include-prompt: true
7           include-completion: true
8
9   management:
10    endpoints:
11      web:
12        exposure:
13          include: health,metrics,prometheus
14    metrics:
15      export:
16        prometheus:
17          enabled: true

```

Key metrics to track:

- Token usage (input and output separately) - directly maps to cost
- Latency percentiles (p50, p95, p99) - user experience indicator
- Error rates by type (timeout, rate limit, validation failure)
- Tool invocation counts - which capabilities get used
- Circuit breaker state changes - early warning system

Set up Grafana dashboards to visualize these metrics in real time. Configure alerts that trigger when error rates spike above acceptable thresholds, token usage approaches your daily budget limits, or request latency degrades beyond your SLA targets. These metrics provide early warning signals that let you identify and fix problems before they impact users or generate complaints.

Structured Logging

Logs explain what happened during specific interactions. Use structured logging with context fields like conversation ID, user ID, and request metadata. This contextual information makes debugging dramatically easier because instead of searching through unstructured text, you can filter logs by specific conversations or users to understand exactly what the agent was doing when problems occurred.

It is important to implement logging at key points in the request lifecycle: before calling the LLM (to capture the prompt and available tools), after receiving the response (to record token usage and tool calls), and during any error conditions.

You can use Spring's MDC (Mapped Diagnostic Context) to attach conversation-level context to all log entries within a request, then clear it when the request completes. This approach ensures every log line carries enough context to be useful during incident investigation.

Distributed Tracing

Traces show the complete request path through your system. Spring AI's Micrometer integration creates spans automatically. Enable OpenTelemetry:

```
1 <dependency>
2   <groupId>io.micrometer</groupId>
3   <artifactId>micrometer-tracing-bridge-otel</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>io.opentelemetry</groupId>
7   <artifactId>opentelemetry-exporter-otlp</artifactId>
8 </dependency>
```

Configure in application.yml:

```
1 management:
2   tracing:
3     sampling:
4       probability: 1.0
5   otlp:
6     tracing:
7       endpoint: http://localhost:4318/v1/traces
```

Distributed tracing reveals the complete flow of each request through your system, making performance bottlenecks immediately visible. Spring AI's Micrometer integration automatically creates spans for LLM calls and tool invocations, showing you where time is actually spent. You'll often discover that 90% of latency comes from slow tool calls rather than LLM inference, or that your retry logic is making problems worse by repeatedly attempting operations that are failing permanently. Traces transform invisible performance issues into concrete, measurable problems you can actually fix.

Lessons Learned and Key Takeaways

Building production AI agents is less about the AI itself and more about the infrastructure around it. The LLM is actually the easy part especially as the models improve every month, APIs are straightforward, and frameworks like Spring AI handle most of the complexity. The real challenge lies in making your agent reliable, safe, and observable under real-world conditions.

The most important lesson from production deployments is to start with safety, not features. Before adding capabilities, implement guardrails. Your first advisor should validate inputs and filter outputs. Add monitoring before expanding what the agent can do. This inverted approach, to building safety infrastructure first, then layering intelligence on top, prevents the painful rewrites that happen when security becomes an afterthought.

Treat LLM calls exactly like external API calls, because that's what they are. They timeout under load, hit rate limits during traffic spikes, and fail when providers have outages. Implement retries with exponential backoff, use circuit breakers to fail fast during extended outages, and design fallback strategies for when primary providers are unavailable. Architectural patterns for distributed systems apply directly to AI agents.

The Model Context Protocol changes how we think about tool integration. Instead of writing custom code for every capability your agent needs (filesystem access, database queries, web search) you configure MCP servers and let Spring AI discover and expose their tools automatically. This standardization reduces maintenance burden and makes testing straightforward. Write the integration once as an MCP server, and it works across every MCP compatible application.

Remember that AI agents are still software, and Spring Boot's core strength is building reliable, maintainable systems at scale. Apply the same engineering discipline to your AI features that you apply to your REST APIs and microservices. The technology might be new, but the requirements haven't changed. Production systems need observability, graceful failure handling, and defense in depth. Focus on these fundamentals, and your AI agent will scale alongside the rest of your application.

References and Further Reading

Spring AI Official Documentation - <https://docs.spring.io/spring-ai/reference/>

Model Context Protocol Specification - <https://modelcontextprotocol.io/docs/getting-started/intro>

Spring AI MCP Guide - <https://www.baeldung.com/spring-ai-model-context-protocol-mcp>

Spring AI Observability Deep Dive - <https://bootcamptoprod.com/spring-ai-chat-client-metrics-guide/>

AI Guardrails in Production - <https://www.wiz.io/academy/ai-security/ai-guardrails>

Building MCP Servers with Spring AI - <https://www.danvega.dev/blog/2025/09/24/cyc-mcp-server-spring-ai>

Spring AI Advisors Explained - [URL](#)

› [Back to the Table of Contents](#) ‹



```
// ai agent
async function executeTask(task) {
  const result = await api.run(task);
  return result;
}

// workflow
if (input.optimized) {
  return ai.process(input);
}
```

Enterprise Playbook for Generative AI Adoption

AI & ML

AgenticAI

GenerativeAI

Author: Lokesh Chenta

Lokesh Chenta is a Senior Principal Software Architect at Sabre Corporation, specializing in scalable systems, cloud architecture, and Generative AI. With a strong foundation in software development and team leadership, Lokesh has successfully driven products from concept to launch. Passionate about innovation and continuous learning in AI/ML and emerging tech.



LinkedIn

Era of Generative AI and Agentic Systems

The use of Generative AI (GenAI) started with the basic traditional use case which is generation of content. Simple applications and wrappers were built over Large Language Models (LLMs). These facilitated users interacting with LLMs and consuming the generated content. But this was just the beginning and the domain of GenAI evolved significantly from that preliminary use case. The concept of Tools enabled LLMs to fetch more context and interact with external environment. The ReAct (Reason and Acting) capabilities of LLMs combined with Tools led to the next generation of systems called Agentic AI systems. Agentic systems have unlocked significant capabilities that were previously unimaginable. These systems and capabilities are driving rapid transformation across all industries.

Enterprise organizations are looking at how to make best use of these Agentic capabilities. They are also preparing to make their services and offerings agentic ready to strengthen their leadership position in their respective domains. In this article, we will discuss an Enterprise playbook to help organizations in this journey. We will look at mainly two tracks. One track where organizations consume the existing AI services to get better. The other track is where organizations build AI applications or prepare their services to be consumed by agentic systems.

Generative AI for Organizational Productivity

Generative AI and agentic systems started playing a major role in improving user productivity tremendously. There are lots of use cases. Tons of tools and products are being built by companies, both startups and established software vendors, that help with productivity use cases. Let us look at some of the major use cases that are related to organizational productivity – related to employees and internal workflows.

Content Generation

Text content generation was the first use of LLMs, made popular with consumer apps like ChatGPT. Later, the sequence generation capabilities of LLMs were enhanced beyond language and made to support multi-modality. Multi-modal LLMs can generate various forms of content – text, music, code, images, video etc. Models like Nano Banana from Google Gemini can create excellent images from user prompt. Models like Sora from OpenAI can create good detail videos based on prompts. This unlocks significant productivity for employees in media domain. And in areas like marketing where creating media-rich marketing content becomes instant. The capabilities are not just limited to new content. Models can edit or generate new content based on existing images. And the media tools have incorporated AI based creation and edit capabilities into their existing tools.

Another significant area of productivity improvement is the office suite of tools. Employees can now generate various forms of content using generative AI. Creating a detailed document from a few lines of input, generating a power point deck from an existing document etc. are some sample use cases. Additional scenarios include generating live transcription in meetings, creating summary notes and extracting action items for meetings etc. Application suites like Microsoft 365 and Google Workspace have made these experiences seamless by integrating AI into their core applications that are already being consumed by millions of users across the globe.

Coding Assistants

Code generation is another form of content generation using LLMs that has picked significant popularity and adoption across the software organizations. The initial use cases were creating small pieces of code from natural language user input and vice-versa – generating language description of existing pieces of code. This got extended to generating test cases, fixing defects, performing code quality checks and suggesting review comments. Tools like GitHub Copilot, Gemini Code Assist etc. were built for this. Integrations of these AI assistants with existing IDEs have made the adoption easy for the developers.

With agentic AI, coding assistants have been transformed into coding agents. And have unlocked major use cases in the coding space beyond basic code manipulation. Coding agents can now create whole code repos, generate code for whole applications, and perform total testing. With help of tools, they can connect to GitHub to commit the application to repos, connect to cloud tools to perform direct deployments as well. With a simple and clear input of requirements, agents can handle the end-to-end flow to fulfill the whole software development life cycle. Coding agents have also found tremendous value in modernizing legacy code bases to latest tech stack for ease of maintenance, reducing a significant cost of ownership

associated with legacy software. Codex from OpenAI, Cursor and Windsurf are some popular coding agents that have made the concept of vibe coding popular and productive.

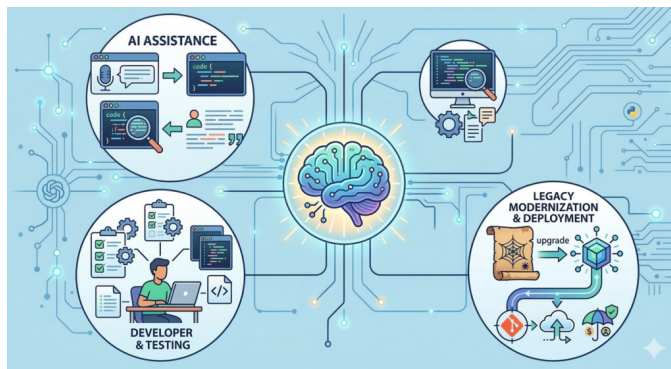


Abb. 7: Generative AI in Software Development

Enterprise Knowledge Search

In any organization, information is spread across various data sources – could be stored in wikis, GitHub docs, dedicated sharepoint sites, files shared in MS-Teams chats, sharepoints linked to Teams locations etc. There will also be employee's own content stored in either local laptop storage or cloud storage connected to employee's account. The forms of data could also be numerous like .doc files, .pdf files, meeting notes stored in some text files. With this sprawl of data, finding the right relevant information with minimal effort becomes a challenge.

With generative AI and integrated ecosystems like Microsoft 365, intra knowledge search becomes instant. Employees can conversationally query for information from across their personal internal storage and enterprise data sources. And get summarized information along with the source links provided for further analysis. The best part is that these integrations also enforce the enterprise level authentication and authorization by linking with existing access control databases. This prevents unauthorized data being made available to users.

Content Processing

In an organizational ecosystem, lot of content gets generated, transferred and consumed. Generative AI can be very helpful in making this content handling efficient. Imagine an executive trying to read an email thread with twenty back and forth conversations between his team. He can use AI assistant to summarize the twenty mails and give a crisp five-point summary and any action items that need his decision. Similarly, imagine an employee created an elaborate word document on a topic. But he needs to present this content for review in a meeting. Converting text from the document to a slide deck, along with pictures and charts to visually present the narrative, will usually be a time-consuming activity. But this becomes an instant activity using AI assistants supported by multi-modality capabilities of LLMs that can generate images as well.

Workflow Agents

The concept of Tools combined with the power of LLMs has led to the innovation of AI agents and agentic systems. Tools enable LLMs to fetch additional information (latest information, domain specific data, enterprise private data etc.) as well as execute tasks or code so actions can be triggered in external environment. Protocols like Model Context Protocol (MCP) have standardized the approach of building tools enabling faster, seamless integration. Organizations across domains have embraced this and started making their product capabilities available as MCP tools. Agents empower systems to autonomously achieve goals by performing tasks. And the availability of powerful tools has unlocked the possibility of creating agentic systems that can handle complex tasks and workflows without needing human intervention. This results in reduced human efforts to create and manage workflows, improved productivity and significant cost savings for organizations.

Consider an example of an agent built in context of software development life cycle (SDLC). A successful pull review for a feature into a branch can be input to the agent. The agent with help of tools can execute relevant test suites. On successful testing, it can create a build, deploy to certification environment. Agent itself can change the status of the feature to "ready

for release” in Agile tool, send email to product owner and to the quality assurance team for final check before moving the build to production. This is the potential of the powerful capabilities and possibilities using AI agents. Most recent applications like Claude Cowork, OpenClaw and Perplexity Computer are great examples of the potential of AI agents.

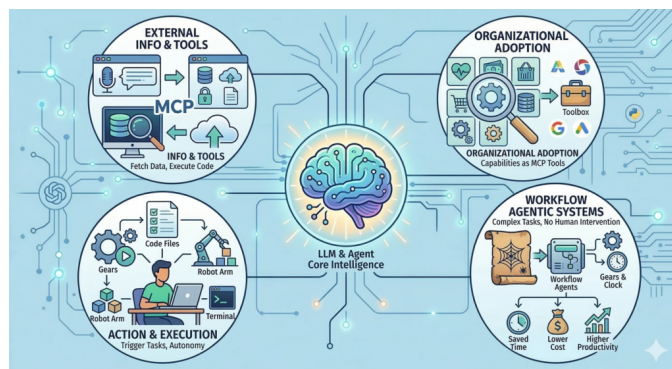


Abb. 8: Workflow Agents

Enterprise Readiness for AI-Native World

There are lot of companies building in the agentic AI space across the spectrum of the stack. Companies like Nvidia are building the hardware. There are companies (OpenAI, Anthropic, Meta etc.) building foundational models and LLMs – both open source and closed source. Cloud providers are building platforms to provide compute, host LLMs for their customers. Companies are building pure GenAI applications (ChatGPT, Perplexity etc.). And some are providing solutions across the ecosystem. For example, Google builds TPUs in hardware space, has Gemini LLMs, provides Vertex AI cloud platform and also builds applications like Gemini Assistant in the GenAI space.

Beyond these companies building purely for the Gen AI stack, we will look at how traditional organizations can build solutions for the AI-native world.

Building Generative AI Solutions

One preliminary use case where generative AI has found tremendous value across domains and organizations is in knowledge search and customer assistance. AI assistants, Question and answer systems are connected to internal enterprise knowledge repositories. Depending on the user input these assistants can fetch relevant information, summarize and provide a crisp summary. Advanced capabilities like conversational agents with memory, seamless support for multilingual capabilities and omni-channel support (text, voice, messaging etc.) significantly enhance the end user experience. AI platforms provide Platform-as-a-Service (PaaS) model for creating agents with these capabilities reducing the work needed for organizations to add these assistants to their products. A typical use case could be a voice assistant embedded in a food delivery application helping users with any of their queries.

Agentic AI added to these assistants turn them into even more productive agents. These support agents can not only provide information but can also complete tasks to fulfill user requirements. With the help of tools available they can invoke complex workflows to meet the goal. For example, the AI support agent embedded in a food delivery application can complete task when user requests for cancellation of existing order. It can coordinate and complete workflows like cancelling the order, initiating money refund, sending email confirmation with details etc.

Apart from generic use cases, organizations should look at their domain specific problems and use cases that can be solved by leveraging generative AI. The capabilities of LLMs can be used to solve problems or requirements that were traditionally either time-consuming or resource intensive or cost intensive. Agentic AI systems enable workflow automation with minimal efforts. A lot of complex, time-consuming, multi-step workflows can be easily transformed into agentic systems.

Building AI Enablers

Apart from the aspect of building generative AI solutions, the other area where traditional organizations can add significant value is by making their domain specific services ready for consumption in the AI-native world. The existing services and

capabilities can be modernized for integration with the new age generative AI applications and agentic AI systems. Agentic systems are powered by tools. Organizations are publishing their core functional capabilities as tools and making them available via MCP servers. And third-party integrators or AI service providers can build their agentic systems by connecting to these MCP servers. This has the dual advantage of the traditional organizations focusing on their core capabilities and at the same time make them available for all the new AI-native applications being built.

Adopting AI-First Culture

To succeed in this AI-native world, organizations need to adopt AI-first culture. Adopting agentic AI cannot be a check list initiative, rather it must be an organizational mindset embraced from top to bottom in the organizational hierarchy. The strategy needs to be both ways – top to bottom and bottom to top.

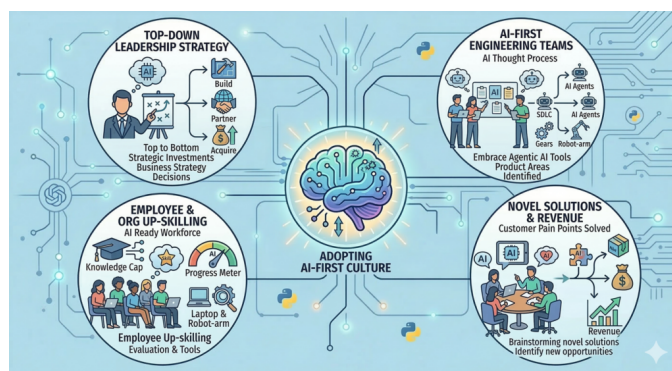


Abb. 9: AI-first culture in organizations

Leaders need to make decisions on business strategies to win in the agentic AI world. Decisions on where to make strategic investments, whether to build solutions or enter into partnership with new age startups or make acquisitions that provide instant advantage. Executive leadership also needs to plan to make all the employees, across all functions, AI ready. Investments should happen in employee up-skilling, regular training and knowledge sharing sessions, evaluation and procurement of tools that improve employee productivity etc.

Employees and engineering teams working closer to products should have the AI-first thought process. Teams need to embrace all the agentic AI tools, applications and workflows that are resulting in tremendous productivity gains across the software development life cycle (SDLC). Teams can identify what product areas can be improved with the agentic AI solutions. As part of customer discussions or requirement brainstorming, using AI-first thought process, teams can identify novel solutions that can be built that address customer pain points and bring in additional revenue.

Finally, there should be channels of continuous communication and collaboration – leadership sharing their vision and plans, employees sharing the feedback on any internal initiatives rolled-out, customer feedback on external AI solutions deployed etc. Considering the novelty and the pace at which agentic AI space is evolving, a constant feedback loop and an iterative approach of build, measure and learn will help organizations succeed by quickly pivoting to new strategies as needed.

Final Thoughts

Generative AI and Agentic AI have unlocked significant capabilities which are driving tremendous transformations across all industries. Organizations need to adapt to these and keep up with this rapid pace not just to thrive but even to survive these waves of changes. Having an enterprise AI strategy and a play book for adoption and pivoting quickly to any changes becomes critical for business. And making careful decisions in selecting the AI tools, partners and platforms, careful analysis of costs against productivity benefits, also become key part of the strategy. Finally, organizations need to be conscious that adoption of generative AI and navigation of the AI-native world is, not a one-time activity but, a continuous journey that organizations need to be prepared for.

```
interface DefectClassificationPort {  
    DefectPrediction classify(WeldImageSample sample);  
}
```



Hexagonal Architecture for ML: Ports, Adapters, and Model Versioning in Spring Boot

Architecture & Microservices

HexagonalArchitecture

MachineLearning

ModelVersioning

Author: Nikita Golovko

Dr. Nikita Golovko is a seasoned Solution Architect with over 16 years of experience in designing scalable, secure, and cost-effective software architectures for industrial and business-critical systems. With a strong academic background and research in machine learning, he bridges the gap between advanced AI technologies and real-world applications on the shop floor.



LinkedIn

Machine learning has changed enterprise software in ways many teams did not expect. In many Spring Boot systems, predictions now shape prices, quality checks, anomaly alerts, fraud decisions, recommendations, and planning. But teams still shape prediction code like a small helper function. A controller calls a service. The service loads a model or sends a request to an endpoint. A score comes back. For a proof of concept, such a setup often works. In production, such a setup breaks down fast.

Once your team needs reproducibility, rollback, audit trails, controlled rollout, or more than one deployment option, the model stops being a minor detail. The model becomes part of the architecture. Your application needs more than a way to ask for a prediction. Your application also needs clear rules for model selection, version tracking, later review, and changes in infrastructure without damage to business logic.

This is where hexagonal architecture helps. The idea is simple. Business use cases stay in the core. External technology stays outside the core and connects through ports and adapters. In classic examples, those external parts include databases, user interfaces, and messaging. In machine learning systems, those parts also include model registries, inference runtimes, feature providers, remote model servers, and audit stores. This view treats model serving as an outbound dependency, not as a hidden part of the core. That distinction matters. *Alistair Cockburn's* original idea still fits well here because the goal stays the same: protect the application from outside technology and keep boundaries clear and testable.

This article looks at how hexagonal architecture helps you structure machine learning features in a Spring Boot application. The goal is not another Java machine learning tutorial. The goal is to show how ports and adapters make prediction use cases easier to change, test, and govern.

The Model Changed - The Architecture Wasn't Ready

Let us consider as an example a quality control service on a production line. A computer vision model checks weld seams in real time. It spots porosity, undercut, and overlap in camera images right after welding. The service does its job. Then, six months later, the data science team ships a better model. Accuracy is higher. False alarms drop. The model file is smaller. From a software architecture view, this kind of change should stay simple. You replace the artifact, deploy again, and move on. Instead, the change takes three weeks and ends with two hotfixes.

The real problem is the tight mix of concerns. `WeldInspectionService` pulls in the ONNX runtime directly. It builds the input tensor, runs inference, reads the raw float output, and turns those values into domain severity levels, all inside one class. That design ties business rules to model-serving details. Once the new model changes its output schema, the business logic fails. Once the endpoint changes, the integration tests fail. The model changed, yet the impact spread across the whole service.

This is the price of tight coupling between machine learning infrastructure and business logic. In the writer's view, this is where production systems lose clarity and speed. Hexagonal Architecture offers a cleaner path. It separates business decisions from technical details and reduces the impact of change, which aligns with the boundary-focused view described by *Cockburn (2005)*.

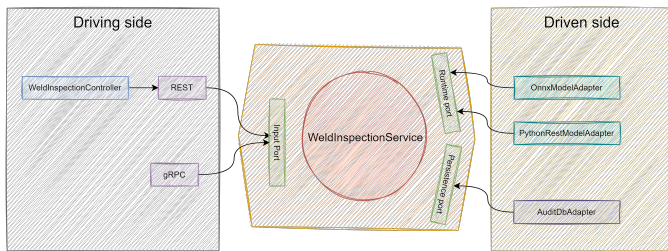
Hexagonal Architecture in 60 Seconds

Hexagonal Architecture, often called Ports and Adapters, structures a service around a domain core. That core stays free from external infrastructure. It does not depend on database code, messaging code, HTTP clients, or model runtime details. Instead, the domain states its needs through interfaces called ports. Adapters provide the concrete implementations of those ports. You are free to replace an adapter without changing the domain logic. From an architecture view, this separation is one of the main strengths of the style.

In this setup, two kinds of ports matter. **Driving ports** are the entry points into the domain. A REST controller or a Kafka consumer uses a driving port to trigger a use case. **Driven ports** work in the other direction. They describe the external services the domain needs to call. These often include a database, a message bus, or, in machine learning systems, the model used for inference.

For a machine learning service, the inference step belongs behind a driven port. The domain should not know where the prediction comes from. It should not care whether inference runs in an ONNX runtime inside the JVM, in a Python service over HTTP, in a cloud endpoint, or in a test double with fixed results. The domain calls the port. The adapter connects that call to the real implementation.

This matters in production. Once you keep inference behind a port, you reduce the impact of change. You swap serving technology, adjust deployment style, or test business rules in isolation without rewriting the core logic. That makes the system easier to understand, easier to test, and easier to change.



Hexagonal Architecture is not a naming convention, and it is certainly not just a folder structure. At its core, it is a dependency rule: the domain must remain independent of infrastructure. Infrastructure may depend on the domain, but the domain must never depend on infrastructure.

In practice, that means every import in the domain deserves scrutiny. Before adding one, it is worth asking a simple question: does this type belong to the domain itself, or does it come from the outside? If the answer is Spring, ONNX, JDBC, Jackson, or any other infrastructure technology, the boundary is already starting to erode.

Domain Model

The right place to begin is the domain model, because it defines the vocabulary of the system. These core concepts should be implemented as plain Java records and enums - nothing more. No annotations, no framework glue, no accidental coupling to infrastructure. Ideally, the domain should compile against the JDK alone.

```
1 package com.example.weldinspection.domain.model;
2 public record WeldImageSample(
3     String jointId,
4     byte[] imageData,      // raw image bytes from camera
5     String cameraId,
6     Instant capturedAt
7 ) {}
```

```
1 package com.example.weldinspection.domain.model;
2 public enum DefectType {
3     POROSITY, UNDERCUT, OVERLAP, NONE
4 }
```

```
1 package com.example.weldinspection.domain.model;
2 public record DefectPrediction(
3     DefectType type,
4     double confidence,
5     String modelVersion
6 ) {}
```

```
1 package com.example.weldinspection.domain.model;
2 public enum Severity { OK, LOW, MEDIUM, HIGH, INCONCLUSIVE }
```

```
1 package com.example.weldinspection.domain.model;
2 public record InspectionResult(
3     String jointId,
4     DefectPrediction prediction,
5     Severity severity
6 ) {}
```

```

1 package com.example.weldinspection.domain.model;
2
3 // Domain-level exception — adapters throw this so the domain
4 // never sees ONNX- or HTTP-specific exceptions.
5 public class ModelInferenceException extends RuntimeException {
6     public ModelInferenceException(String message) { super(message); }
7     public ModelInferenceException(String message, Throwable cause) {
8         super(message, cause);
9     }
10 }

```

These are the types that the entire application should speak. When a REST controller receives a request, it translates that external representation into a `WeldImageSample`. When an ML adapter returns an inference result, it translates it into a `DefectPrediction`. That translation belongs at the boundary — always at the boundary, never inside the domain.

This separation matters because it keeps the domain model focused on business meaning rather than transport formats, framework objects, or provider-specific payloads. Controllers, adapters, and integration code are responsible for converting between the outside world and the core’s language. The domain itself should not carry that burden. That principle aligns closely with the DDD idea that the model should preserve a clear and consistent domain language across the system (*Evans, Domain-Driven Design*).

Define the Driven and Driving Ports

A **driving port** defines how the outside world is allowed to enter the domain. REST controllers, Kafka consumers, scheduled jobs, and tests should all reach the core through such ports. The interface must express the system’s capabilities in domain terms only.

```

1 package com.example.weldinspection.domain.port.in;
2 public interface InspectWeldUseCase {
3     InspectionResult inspect(WeldImageSample sample);
4 }

```

This is also why one interface per use case is often the stronger choice. A controller should depend on `InspectWeldUseCase`, not on `WeldInspectionService`. That keeps the boundary stable even when the implementation behind it changes. The internal design can be refactored, business rules can be revised, and service classes can be replaced without forcing changes into the adapter layer.

Driven ports are interfaces defined by the domain itself, even though their implementations live elsewhere. They express what the core needs from the outside world without tying that need to any specific technology. In other words, the domain may say, “something must be able to classify an image,” but it should never say, “this requires an ONNX runtime.”

That distinction is fundamental. It allows the domain to remain focused on capabilities rather than technical products, libraries, or protocols. The moment an infrastructure detail appears in the port itself, the abstraction has already started to collapse. A well-designed, driven port keeps the dependency visible but expresses it in the domain language rather than in the language of tools and frameworks. That is very much in line with Cockburn’s original inside-versus-outside view of Hexagonal Architecture and with the DDD principle that the model should be shaped around domain meaning rather than technical mechanics (*Cockburn; Evans*).

```

1 package com.example.weldinspection.domain.port.out;
2 // The ML boundary. The domain calls this; adapters implement it.
3 // Notice: no ONNX import, no RestTemplate import, no framework import.
4 public interface DefectClassificationPort {
5     DefectPrediction classify(WeldImageSample sample);
6 }

```

```

1 package com.example.weldinspection.domain.port.out;
2 // The persistence boundary. Same principle — no JPA, no JDBC.
3 public interface InspectionAuditPort {
4     void record(String jointId, DefectPrediction prediction, Severity severity);
5 }

```

Both interfaces use domain types for a simple reason: the boundary must remain technology-neutral. `DefectPrediction` and `Severity` are part of the domain model, so they can safely cross that boundary. Infrastructure types cannot.

That decision may seem small in code, but it has large architectural consequences. A migration from ONNX to a cloud inference service should require a new adapter, not a redesign of the port, not a rewrite of the service layer, and not a cascade of broken tests. When the boundary is defined in domain terms, the surrounding technology can change without dragging the core along with it.

The Domain Service

The domain service implements the driving port and depends on the driven ports. Its responsibility is straightforward: coordinate business rules such as severity thresholds, confidence cutoffs, and audit requirements — nothing more.

That is also why Spring annotations do not belong here. A domain service should not need a container in order to exist. It should be possible to construct it directly in a plain unit test and verify its behavior without framework startup, classpath scanning, or model files lying around in the test environment.

```

1  package com.example.weldinspection.domain.service;
2  // No @Service, no @Component, no Spring imports.
3  // Spring wiring is handled entirely in BeanConfiguration.
4  public class WeldInspectionService implements InspectWeldUseCase {
5      private final DefectClassificationPort classifier;
6      private final InspectionAuditPort audit;
7      public WeldInspectionService(
8          DefectClassificationPort classifier,
9          InspectionAuditPort audit) {
10         this.classifier = classifier;
11         this.audit = audit;
12     }
13     @Override
14     public InspectionResult inspect(WeldImageSample sample) {
15         DefectPrediction prediction = classifier.classify(sample);
16         Severity severity = resolveSeverity(prediction);
17         audit.record(sample.jointId(), prediction, severity);
18         return new InspectionResult(sample.jointId(), prediction, severity);
19     }
20     // Business rule: confidence below threshold is always INCONCLUSIVE
21     // regardless of defect type. This rule belongs here, not in the adapter.
22     private Severity resolveSeverity(DefectPrediction p) {
23         if (p.confidence() < 0.60) return Severity.INCONCLUSIVE;
24         return switch (p.type()) {
25             case POROSITY -> Severity.HIGH;
26             case UNDERCUT -> Severity.MEDIUM;
27             case OVERLAP -> Severity.LOW;
28             case NONE -> Severity.OK;
29         };
30     }
31 }

```

This keeps testing fast and precise. A rule like `resolveSeverity` can be exercised in milliseconds, and when that rule changes, the failure should be limited to the tests that describe it. That is the real value of the boundary: infrastructure remains outside, while the business core stays easy to understand, easy to evolve, and easy to trust (*Ford et al., Software Architecture: The Hard Parts*).

Driving Adapter

A driving adapter exists to translate, not to decide. It converts external representations into domain types, calls the use-case port, and maps the result back into the format expected by the outside world. Business logic does not belong in the controller; only boundary translation does.

That is why HTTP DTOs should remain in the adapter layer. They reflect HTTP concerns, not domain meaning. The same boundary discipline applies to dependencies: the controller should depend on the use-case interface, never on a concrete service class. Once that rule is in place, internal implementations can be reorganized freely — split, merged, or replaced — without forcing changes in the controller. The adapter should be able to observe the capability, not the implementation behind it.

```
1 package com.example.weldinspection.adapter.in.rest;
2
3 // HTTP wire format — NOT a domain type.
4 // byte[] is received as a Base64-encoded string in JSON.
5 // For production use, prefer multipart/form-data with MultipartFile.
6 public record InspectWeldRequest(
7     String jointId,
8     String cameraId,
9     byte[] imageData
10 ) {}
```

```
1 package com.example.weldinspection.adapter.in.rest;
2 public record InspectWeldResponse(
3     String jointId,
4     String defectType,
5     double confidence,
6     String severity,
7     String modelVersion
8 ) {}
```

```
1 package com.example.weldinspection.adapter.in.rest;
2 @RestController
3 @RequestMapping("/api/v1/inspections")
4 public class WeldInspectionController {
5     // Depends on the USE CASE INTERFACE — not on WeldInspectionService directly.
6     // The controller has no idea which service implementation is behind this port.
7     private final InspectWeldUseCase inspectWeld;
8     public WeldInspectionController(InspectWeldUseCase inspectWeld) {
9         this.inspectWeld = inspectWeld;
10    }
11    @PostMapping
12    public ResponseEntity<InspectWeldResponse> inspect(
13        @RequestBody InspectWeldRequest request) {
14        // Translate: HTTP request → domain type
15        WeldImageSample sample = new WeldImageSample(
16            request.jointId(),
17            request.imageData(),
18            request.cameraId(),
19            Instant.now()
20        );
21        InspectionResult result = inspectWeld.inspect(sample);
22        // Translate: domain type → HTTP response
23        return ResponseEntity.ok(new InspectWeldResponse(
24            result.jointId(),
25            result.prediction().type().name(),
26            result.prediction().confidence(),
27            result.severity().name(),
28            result.prediction().modelVersion()
29        ));
30    }
31 }
```

Driven Adapters

Driven adapters implement the driven ports and form the system's infrastructure boundary. They are the only classes allowed to import technologies such as ONNX, `RestTemplate`, or `JdbcTemplate`. Their job is straightforward: convert infrastructure types into domain types, and domain types back into infrastructure calls — nothing more.

That limitation is intentional. It keeps infrastructure concerns exactly where they belong: outside the core. The adapter may know how to call a model runtime, an HTTP endpoint, or a database client, but it should not contain business rules or reshape the domain. Once that discipline is maintained, infrastructure can change freely while the core remains stable, readable, and testable.

ML Adapter — ONNX Runtime (production model)

```

1  package com.example.weldinspection.adapter.out.ml;
2  // No Spring annotations — instantiated and managed via BeanConfiguration.
3  public class OnnxModelAdapter implements DefectClassificationPort {
4      private static final int    IMG_SIZE = 224;
5      private static final String[] LABELS  = {"POROSITY", "UNDERCUT", "OVERLAP", "NONE"};
6      private final OrtEnvironment env;
7      private final OrtSession    session;
8      private final String        modelVersion;
9      public OnnxModelAdapter(String modelPath, String modelVersion)
10         throws OrtException {
11         this.env = OrtEnvironment.getEnvironment();
12         this.session = env.createSession(modelPath);
13         this.modelVersion = modelVersion;
14     }
15     @Override
16     public DefectPrediction classify(WeldImageSample sample) {
17         try {
18             float[] tensor = preprocess(sample.imageData());
19             long[] shape = {1, 3, IMG_SIZE, IMG_SIZE};
20             OnnxTensor input = OnnxTensor.createTensor(
21                 env, FloatBuffer.wrap(tensor), shape);
22             try (OrtSession.Result result = session.run(Map.of("input", input))) {
23                 float[] scores = (float[]) ((OnnxTensor) result.get(0)).getValue();
24                 int best = argmax(scores);
25                 return new DefectPrediction(
26                     DefectType.valueOf(LABELS[best]),
27                     scores[best],
28                     modelVersion
29                 );
30             }
31         } catch (OrtException e) {
32             // Wrap infrastructure exception in a domain exception.
33             // The domain service never sees OrtException.
34             throw new ModelInferenceException("ONNX inference failed", e);
35         }
36     }
37     public void close() throws OrtException { session.close(); env.close(); }
38     private float[] preprocess(byte[] imageData) {
39         return new float[3 * IMG_SIZE * IMG_SIZE];
40     }
41     private int argmax(float[] scores) {
42         int best = 0;
43         for (int i = 1; i < scores.length; i++)
44             if (scores[i] > scores[best]) best = i;
45         return best;
46     }
47 }

```


Persistence Adapter — Audit Trail

```

1 package com.example.weldinspection.adapter.out.persistence;
2 public class AuditDbAdapter implements InspectionAuditPort {
3     private final JdbcTemplate jdbc;
4     public AuditDbAdapter(JdbcTemplate jdbc) { this.jdbc = jdbc; }
5     @Override
6     public void record(
7         String jointId,
8         DefectPrediction prediction,
9         Severity severity) {
10         jdbc.update("""
11             INSERT INTO inspection_audit
12             (joint_id, defect_type, confidence, severity, model_version, recorded_at)
13             VALUES (?, ?, ?, ?, ?, ?)
14             """,
15             jointId,
16             prediction.type().name(),
17             prediction.confidence(),
18             severity.name(),
19             prediction.modelVersion(),
20             Instant.now()
21         );
22     }
23 }

```

Connecting Everything in One Place

`BeanConfiguration` is the one class that sees the full picture. It reads the configuration values, decides which adapters to create, and connects them to the domain service. This keeps setup logic in one place. The rest of the code does not need to know how the parts are selected or wired together. In the writer's view, this single composition point makes the system easier to read, test, and change.

```

1 package com.example.weldinspection.config;
2 @Configuration
3 public class BeanConfiguration {
4     @Bean
5     public RestTemplate inferenceRestTemplate(
6         @Value("${app.model.connect-timeout-ms:2000}") int connectTimeout,
7         @Value("${app.model.read-timeout-ms:5000}") int readTimeout) {
8         var factory = new SimpleClientHttpRequestFactory();
9         factory.setConnectTimeout(connectTimeout);
10        factory.setReadTimeout(readTimeout);
11        return new RestTemplate(factory);
12    }
13    @Bean(destroyMethod = "close")
14    @ConditionalOnProperty(name = "app.model.adapter", havingValue = "onnx")
15    public DefectClassificationPort onnxModelAdapter(
16        @Value("${app.model.path}") String modelPath,
17        @Value("${app.model.version}") String modelVersion) throws Exception {
18        return new OnnxModelAdapter(modelPath, modelVersion);
19    }
20    @Bean
21    public InspectionAuditPort auditDbAdapter(JdbcTemplate jdbc) {
22        return new AuditDbAdapter(jdbc);
23    }
24    @Bean
25    public InspectWeldUseCase weldInspectionService(
26        DefectClassificationPort classifier, InspectionAuditPort audit) {
27        return new WeldInspectionService(classifier, audit);
28    }
29 }

```

Model Versioning Strategy

A port boundary solves the basic model swap problem. In production, your team faces a tougher question. Which adapter is active right now? How do you switch it without a full redeployment?

The versioning strategy below grows step by step. Each stage adds one new capability. Your team does not need to move to the next stage before it is ready. All three stages implement `DefectClassificationPort`. The domain service stays the same in every stage. Only the adapter and the configuration change. This separation follows the ports-and-adapters view described by *Cockburn (2005)*.

Stage 1 — Version Tag in Configuration

application.yml

```
1 app:
2   model:
3     adapter: onnx
4     path:    /opt/models/weld-classifier-v2.3.onnx
5     version: v2.3
```

```
1 @Bean(destroyMethod = "close")
2 @ConditionalOnProperty(name = "app.model.adapter", havingValue = "onnx")
3 public DefectClassificationPort onnxModelAdapter(
4     @Value("${app.model.path}") String modelPath,
5     @Value("${app.model.version}") String modelVersion) throws Exception {
6     return new OnnxModelAdapter(modelPath, modelVersion);
7 }
```

Each prediction carries its `modelVersion` into the audit trail through the `modelVersion` field on `DefectPrediction`. This creates a clear link between every past result and the exact model artifact that produced it.

Stage 2 — Shadow Mode (Validate Before Committing)

Before your team sends real production traffic to a new model version, run both models side by side. The live model returns its result to the caller. The candidate model runs in the background, and the system logs its prediction for later comparison. The business outcome stays unchanged because the caller always receives the result from the live model.

In the writer's view, this is one of the safest ways to evaluate a new model in production. Your team gets real comparison data without exposing users or operations to unnecessary risk. This fits well with an architecture style that keeps change controlled and visible at the boundary between business logic and infrastructure (*Cockburn, 2005*).

```
1 package com.example.weldinspection.adapter.out.ml;
2 // Implements the same port as OnnxModelAdapter and PythonRestModelAdapter.
3 // The domain service cannot tell the difference.
4 public class ShadowModelAdapter implements DefectClassificationPort {
5     private final DefectClassificationPort production; // current live model
6     private final DefectClassificationPort candidate; // model under evaluation
7     private final ShadowComparisonPort shadowLog;
8     public ShadowModelAdapter(
9         DefectClassificationPort production,
10        DefectClassificationPort candidate,
11        ShadowComparisonPort shadowLog) {
12        this.production = production;
13        this.candidate = candidate;
14        this.shadowLog = shadowLog;
15    }
16    @Override
17    public DefectPrediction classify(WeldImageSample sample) {
18        // Production result returned to domain service immediately.
19        DefectPrediction prod = production.classify(sample);
20        // Candidate runs asynchronously — never delays the caller.
```

```

21     CompletableFuture.runAsync(() -> {
22         try {
23             DefectPrediction shadow = candidate.classify(sample);
24             shadowLog.record(sample.jointId(), prod, shadow);
25         } catch (Exception e) {
26             // Candidate failures are logged, never propagated.
27             shadowLog.recordFailure(sample.jointId(), e.getMessage());
28         }
29     });
30     return prod;
31 }
32 }

```

```

1 package com.example.weldinspection.domain.port.out;
2 // Driven port for persisting shadow comparison results.
3 // Back this with a DB adapter, Micrometer metrics, or a dedicated comparison service.
4 public interface ShadowComparisonPort {
5     void record(String jointId,
6                 DefectPrediction production,
7                 DefectPrediction candidate);
8     void recordFailure(String jointId, String reason);
9 }

```

```

1 @Bean
2 @ConditionalOnProperty(name = "app.model.adapter", havingValue = "shadow")
3 public DefectClassificationPort shadowModelAdapter(
4     @Value("${app.model.path}") String prodPath,
5     @Value("${app.model.version}") String prodVersion,
6     @Value("${app.model.shadow.path}") String shadowPath,
7     @Value("${app.model.shadow.version}") String shadowVersion,
8     ShadowComparisonPort shadowLog) throws Exception {
9     var production = new OnnxModelAdapter(prodPath, prodVersion);
10    var candidate = new OnnxModelAdapter(shadowPath, shadowVersion);
11    return new ShadowModelAdapter(production, candidate, shadowLog);
12 }

```

```

1 app:
2   model:
3     adapter: shadow
4     path: /opt/models/weld-classifier-v2.3.onnx # production
5     version: v2.3
6     shadow:
7       path: /opt/models/weld-classifier-v3.0.onnx # candidate
8       version: v3.0

```

Once the shadow comparison shows an acceptable divergence rate, the team switches the adapter to the new model. In plain terms, the candidate model agrees often enough with the live model on real weld data. At that point, the rollout becomes simple. You change one configuration value, and the service starts using the new model.

Shadow mode answers a question offline validation never answers well. How does the new model behave on real production traffic, side by side with the current model, before any business decision depends on it? In practice, this provides stronger evidence and reduces rollout risk. It also fits the ports-and-adapters style, in which the application core remains stable while infrastructure choices change at the boundary (*Cockburn, 2005*).

Stage 3 — Registry-Driven Versioning (Runtime Switching)

The registry-based approach adds only two new elements to the domain. The first is `ModelArtifact`. The second is `ModelRegistryPort`. Both stay inside the domain package, and both stay free of infrastructure imports.

`ModelArtifact` is a small immutable value object. It describes a model your service is ready to deploy. In this case, it holds two pieces of information: the version identifier and the path to the model file on disk.

```

1 package com.example.weldinspection.domain.model;
2 public record ModelArtifact(
3     String version,
4     String path,
5     String registeredAt
6 ) {}

```

`ModelRegistryPort` defines what the application needs from the outside world. Given a model type, it returns the artifact marked as active. This keeps the responsibility clear. The domain states the need in domain language. The infrastructure provides the concrete result.

```

1 // THE PORT – domain declares what it needs, names nothing concrete
2 public interface ModelRegistryPort {
3     ModelArtifact resolveActive(String modelType);
4 }
5 public record ModelArtifact(String version, String path) {}

```

In the writer's view, this is a clean step forward. You add model selection without pulling registry logic into the core of your application.

`VersionAwareModelAdapter` holds the versioning logic in one clear place. It implements `DefectClassificationPort`, the same interface used by `OnnxModelAdapter` and `PythonRestModelAdapter`, so the domain service stays unchanged. This is a good example of keeping the core stable while the infrastructure changes around it, which fits the ports and adapters view described by *Cockburn (2005)*.

```

1 // THE ADAPTER – resolves version at request time, not at startup
2 public class VersionAwareModelAdapter implements DefectClassificationPort {
3     private final ModelRegistryPort registry;
4     private final Map<String, OrtSession> sessionCache = new ConcurrentHashMap<>();
5     @Override
6     public DefectPrediction classify(WeldImageSample sample) {
7         ModelArtifact active = registry.resolveActive("weld-classifier");
8         OrtSession session = sessionCache.computeIfAbsent(
9             active.version(), v -> loadSession(active.path()));
10        float[] scores = runInference(session, sample.imageData());
11        return new DefectPrediction(bestLabel(scores), bestScore(scores),
12            active.version()); // ← version stamped on every prediction
13    }
14 }

```

Each time `classify` runs, the adapter asks the registry which model is active. It then checks a session cache keyed by the version string. If a session for that version already exists, the adapter reuses it. If the registry points to a new version, `computeIfAbsent` loads the new session on the first request that reaches it. Older sessions stay in the cache, though they stop receiving new traffic.

The version identifier from the registry is then written straight into the returned `DefectPrediction`. From there, it moves through the domain service into the audit record. You do not need extra tracing code to make this work. The design keeps version selection, runtime reuse, and traceability close together, while business logic stays clean.

`DbModelRegistryAdapter` handles the infrastructure side of the registry port. It runs one SQL query against the `model_registry` table and returns the row marked as active for the requested model type.

The `@Cacheable` annotation keeps the service from hitting the database on every inference request. Without this cache, a busy inspection line would query the registry table for each call and add needless load. The cache time to live, set in `application.yml`, defines the delay between an operator changing the active row and all running service instances picking up the new version.

```

1 // THE REGISTRY IMPLEMENTATION – one DB row controls what runs in production
2 @Cacheable("model-registry")
3 public ModelArtifact resolveActive(String modelType) {
4     return jdbc.queryForObject(
5         "SELECT version, path FROM model_registry WHERE type = ? AND active = true",
6         (rs, row) -> new ModelArtifact(rs.getString("version"), rs.getString("path")),
7         modelType
8     );
9 }

```

In most manufacturing settings, 30 seconds is a sensible starting point. In the writer's view, this gives the team a good balance between fast rollout and stable operation. The design keeps registry access in the infrastructure layer and protects the domain from database details, which aligns with the ports-and-adapters approach described by *Cockburn (2005)*.

The Model Changed. This Time, the Architecture Was Ready

The team now has a different experience. The data science team trains a better model, and the rollout stays simple. They package the new model as an ONNX artifact and register it in the model registry. Then they update the active row for weld-classifier to point to v3.0. On the next request, `VersionAwareModelAdapter` picks up the new artifact. No application code changes. No redeployment follows. No hotfix appears.

The audit trail records which model version scored each weld, together with the joint ID and the timestamp. When a quality engineer asks why weld joint J-4471 was flagged as high severity on a Tuesday afternoon, the answer sits one query away.

Before the switch, the team runs v3.0 next to v2.3 in shadow mode for two weeks. The divergence rate ends at 1.2 percent. The team accepts the result. The domain service does not need to change, and it does not even notice the swap.

Hexagonal Architecture does not reduce the complexity of machine learning systems. In the writer's view, it puts complexity where it belongs. It keeps versioning, swapping, and runtime details inside adapters. It keeps business rules inside the domain. That split makes the system easier for you to test, for your team to change, and for you to trust in production.

› [Back to the Table of Contents](#) ‹

JCON2027
www.jcon.one

APR
26-29



JAVAPRO

📍 CINEDOM COLOGNE

JCON EUROPE 2027
International Java Community Conference



Open-Core with Core Java and Vaadin — Part 1

Architecture & Microservices

OpenCore

Vaadin

ServiceLoader

Architecture

Author: Sven Ruppert

CONSPEAKER

Sven Ruppert has been developing in Java since 1996 and has worked as a Developer Advocate for both JFrog and Vaadin. He has been speaking at conferences worldwide for over ten years. He specialises in Core Java and Secure Coding, with a focus on RAG systems and AI security. He also writes regularly for it-portals and magazines and maintains his personal website at svenruppert.com; furthermore, he serves as an instructor for Arctic winter expeditions.



LinkedIn

The open-core model is often reduced in discussion to questions of licensing or commercial positioning. An open-source variant of a piece of software is complemented by a commercially distributed variant that brings additional features. Anyone who actually wants to implement such a separation in an existing codebase quickly discovers that the real difficulty lies neither in the licence text nor in the choice of repository, but in the code's architecture.

The central question is which dependency directions are permissible and which must be prevented under all circumstances. The extension may access the core, while the core must know nothing of the extension's existence. In theory, this is self-evident. In practice, however, one repeatedly observes that an inconspicuous helper class, a widespread configuration mechanism, or a convenient annotation reverses this direction over time without anyone noticing. With that, the topic of this article is outlined: how to build a Vaadin-based open-core product so that the separation is not merely documented but visible directly in the source code.

For the implementation, Spring and any Jakarta EE platform are deliberately avoided. This decision is not a judgment on the platforms in question, but a didactic one. Both tend to make the discovery and wiring of components so convenient that the question of where a class actually originates at runtime, and by what route it enters the application context, recedes into the background. Yet this question is central in the open-core context. Marking a component class with an annotation and having it picked up via a classpath scan yields a convenient solution, but loses visibility over which contributions come from which module. An explicit mechanism, in this case the `ServiceLoader` from the Java standard library, forces the author of the core, just as much as the author of the extension, to specify the direction of the contribution.

One distinction should be addressed up front, since it regularly leads to misunderstandings. Vaadin Flow requires a servlet container, and the Servlet API is brought into the classpath in the example transitively via Vaadin and an embedded Jetty. The application itself is therefore not a Jakarta EE application. It uses neither CDI nor JPA, nor EJB nor JAX-RS, and it does not depend on an application server.

The subject domain is a deliberately trivial example, namely a counter with three operations and an associated event model. This triviality is intentional. It prevents the architectural argument from being obscured by domain detail and ensures that every step of the extension mechanism remains immediately comprehensible. What emerges in the end is not a realistic business product but a fully worked-through architectural demonstration of how an open-core Vaadin system can be built using only the JDK.

All source code is published on GitHub <https://3g3.eu/naityh>.

The article can be read independently of that. Anyone wishing to follow individual passages in their full context will find the unabridged state of the implementation there.

Architectural overview and project structure

The visibility of dependency directions in the previous chapter begins not in the source code but in the project structure. The example application introduced here consists of two independent Maven projects, `counter-community` and `counter-enterprise`. Both are fully fledged artefacts that can be built individually, with their own POM, version progression, and release strategy. They are not connected through a shared hierarchy but merely through a simple library-level dependency.

The permitted dependency direction runs exclusively from the enterprise extension to the community edition. The enterprise extension takes the community as a library, uses its extension API, and accesses its layout classes. The community, on the other hand, knows nothing of the enterprise extension, neither at compile time nor at runtime. This asymmetry is the central architectural statement of the entire endeavour. It is not an agreement whose observance is secured by discipline alone, but a build-enforced property. Because the community project contains no reference to the enterprise project in its POM, the source code of the community edition simply cannot reference enterprise classes. What is not on the classpath at build time cannot be named in any import statement.

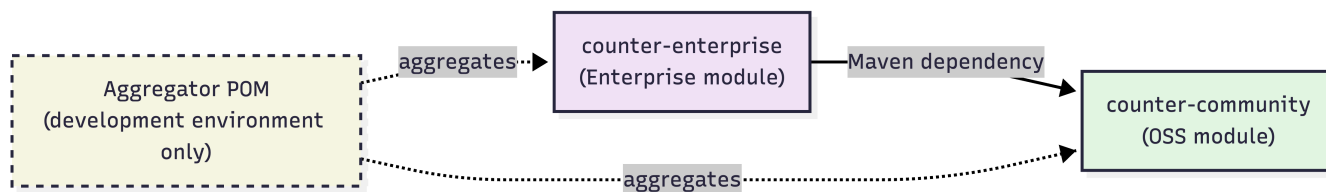


Abb. 10: Dependency direction of the Maven modules

This separation could in principle also be realised within a single multi-module Maven reactor. In the present article, however, the route via two mutually independent projects was deliberately chosen. The reason is not technical but organisational and didactic at the same time. In real open-core operation, the core and the commercial extension are usually developed in separate repositories, possibly even by different teams under different licensing terms. The structure chosen here mirrors this reality rather than concealing it behind a shared reactor file. The consequence is that the core can be released independently, without bundling the extension with it, and that the extension is built against a specific version of the core, which is named like any other library dependency.

For day-to-day work in a development environment, however, the strict separation into two repositories is inconvenient. As long as the core and the extension are evolving together, it helps to be able to open, build, and browse both projects in a single session. For this reason, an aggregator POM resides in the root directory. It lists the two modules and enables IntelliJ IDEA, for instance, to open both projects in the same session.

This aggregator POM is explicitly not a shared parent POM. The two module POMs do not reference it via `<parent>`, and they inherit neither its dependency management nor its plugin configuration. This distinction may seem pedantic at first, but it is deliberate. A shared parent POM would create a technical link between the two modules, contradicting the independence established earlier. The community project could no longer be built without the parent POM, and any divergence between the two projects in versioning strategy or plugin configuration would fail at this level of the hierarchy. The aggregator POM therefore remains a purely development-environment tool, with no significance for the build process of individual modules.

The domain: counter and events

The application's domain task is one of selected triviality. An integer value can be incremented, decremented, and reset to zero. With that, the domain is exhaustively described in a single sentence. This reduction to the bare minimum is, as already hinted at in the introduction, intentional. Any elaboration on the domain would obscure rather than sharpen the article's architectural argument, since it would distract from the mechanisms discussed here. The extension mechanics of an open-core application are to be considered independently of whether the extended system manages a counter, an invoicing process, or a utility network.

The state is encapsulated in a class `CounterState`, which manages an integer value and provides three methods: incrementing by 1, decrementing by 1, and resetting to 0. Each of these operations not only changes the internal state but also returns an event describing the change carried out. The three operations are defined in an enumeration, `CounterAction`, with the constants `INCREMENT`, `DECREMENT`, and `RESET`. This allows the cause of a value change to be named directly later, for example, in an audit log, without having to reconstruct the call path back to the state class.

The event itself is modelled as an immutable record `CounterChangedEvent`. It carries four pieces of information: the previous value, the new value, the action that triggered the change, and a timestamp set when the event is created. The choice of a Java record here is not an end in itself but a semantic statement. An event is, by its nature, immutable. Anything that could be retrospectively altered would amount to rewriting history, which in an audit context would be all but absurd.

Above this state class sits a thin service, `CounterService`, which serves as the sole outward act for counter operations. It delegates the operations to `CounterState`, takes the resulting event, and distributes it to all registered listeners. This separation between state and service may seem over-engineered for the trivial domain considered here, but it becomes important later in the article. The service is the point at which the article's extension mechanism takes hold. Any number of listeners can be attached to it, without the state class itself needing to know of their existence.

With that, the domain foundation for everything that follows is in place. The community edition will make the counter visible via Vaadin and offer the three operations as buttons. The enterprise extension will not interfere with the domain itself; it

will merely contribute listeners that intercept each event and process it for their own purposes, be it a history view, an audit log, or an export view. Nothing more than this brief sketch is needed at this point.

The extension API

The extension API is the contract surface where core and extension meet. It defines the form in which a module may bring itself into a running application, and it equally defines what it must not do. Both aspects are equally important. An overly powerful extension API may seem flexible in the short term, but it undermines the very properties for which the open-core model was chosen in the first place, namely, the predictability of the core in the face of arbitrary extensions.

At the centre stands the interface `FeatureContribution`. An implementation of this interface represents exactly one functional unit that joins the application. It carries a meaningful identifier indicating its origin, along with three contributions that may be added to the overall system: a list of routes, a list of menu entries, and a list of additions to the navigation bar. There is also an optional sort order, against which contributions are arranged during assembly. The interface contains nothing further. It is deliberately restricted to the descriptive and contains no method that could actively change the system. The methods for navbar additions and sort order come with default implementations, so an extension only needs to override the methods whose contributions it is interested in.

```
1 public interface FeatureContribution {
2     String id();
3     List<RouteContribution> routes();
4     List<MenuContribution> menuItems();
5     default List<NavbarContribution> navbarItems() {
6         return List.of();
7     }
8     default int order() {
9         return 1000;
10    }
11 }
```

The two contribution types `RouteContribution` and `MenuContribution` are designed as immutable data records. A `RouteContribution` carries the relative path and the associated Vaadin component class; a `MenuContribution` additionally carries a display label, its own sort order, and an icon identifier. Both are modelled as *records*, since they are pure descriptive objects without behaviour. In both cases, a compact constructor is provided that guards the mandatory fields against `null` values and, for the route, additionally prevents the path from starting with a leading slash.

```
1 public record RouteContribution(
2     String path,
3     Class<? extends Component> viewClass
4 ) {
5     public RouteContribution {
6         Objects.requireNonNull(path, "path");
7         Objects.requireNonNull(viewClass, "viewClass");
8         if (path.startsWith("/")) {
9             throw new IllegalArgumentException(
10                 "Route path must not start with '/': " + path);
11         }
12     }
13 }
```

```
1 public record MenuContribution(
2     String label,
3     String path,
4     int order,
5     String iconName
6 ) {
7     public MenuContribution {
8         Objects.requireNonNull(label, "label");
9         Objects.requireNonNull(path, "path");
10    }
11 }
```

The third contribution type, `NavbarContribution`, deliberately departs from this pattern. It is modelled not as a record but as an interface. The reason lies in a peculiarity of Vaadin. A Vaadin component can be attached to only one parent in the component tree at any given time. If the extension were to supply a ready-made component instance as the contribution, it could not reliably be attached to multiple UI instances. For this reason, a `NavbarContribution` does not deliver a finished component but rather a factory function, a Supplier, that produces a fresh instance on each use. In addition to this factory, it carries an identifier for diagnostic purposes and a sort order whose lower values place an entry further to the left in the navigation bar. The distinction between record and interface is not arbitrary; it follows directly from the properties of the underlying UI framework. Where descriptive objects suffice, records are used; where instantiation per use case is required, an interface takes their place.

```
1 public interface NavbarContribution {
2     String id();
3     Supplier<Component> componentFactory();
4     default int order() {
5         return 1000;
6     }
7 }
```

For observing the counter, a fourth, complementary contract is provided. The interface `CounterEventListener` describes a single listener that is notified of each value change. Since not every extension is necessarily interested in these events, the option to contribute listeners was not added to `FeatureContribution` itself but was moved into a specialised sub-variant, `CounterEventFeature`, which extends the base interface with the method `counterEventListeners`. An extension that wishes only to contribute new views still implements only `FeatureContribution`. An extension that also wishes to react to counter events implements `CounterEventFeature` and supplies a list of its listeners there. This split follows the principle that an interface should require only the methods indispensable to its purpose, without dragging along unused appendages.

```
1 public interface CounterEventListener {
2     void onCounterChanged(CounterChangedEvent event);
3 }
```

```
1 public interface CounterEventFeature extends FeatureContribution {
2     List<CounterEventListener> counterEventListeners();
3 }
```

What the API does not permit is just as instructive as what it does. An extension cannot replace an existing route, exchange the implementation of `CounterService`, hook into the main layout, or rebuild the application context. It can only register new views, contribute new menu entries, place additional elements in the navigation bar, and attach new listeners to the counter event. The extension acts additively, never invasively. This restriction may seem strict, but it is a deliberate architectural trade-off. What the API loses in power, it gains in stability. The core can evolve independently without an extension, with a life of its own, breaking, as long as the few structures agreed upon here remain unchanged.

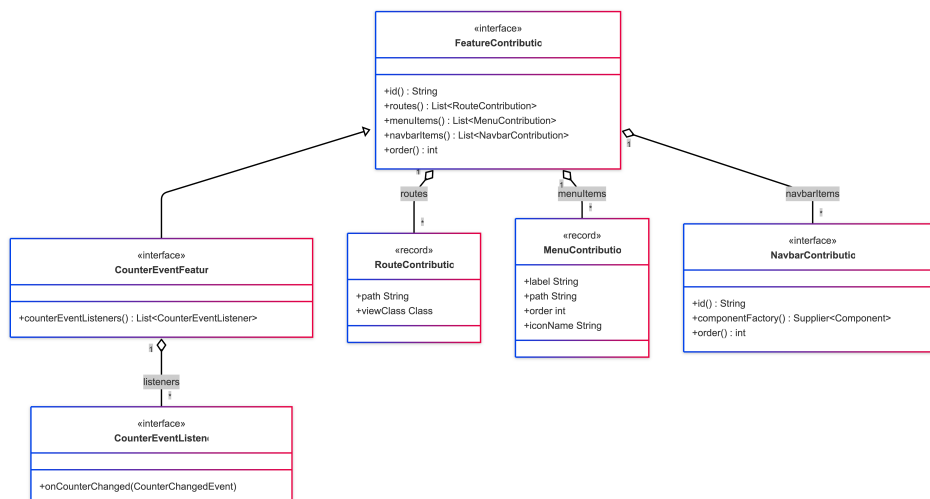


Abb. 11: Class diagram of the extension API

Discovery via the ServiceLoader and the FeatureRegistry

With the API described in the previous chapter, only the form in which extensions may be introduced has been fixed. What remains open is how the core learns of the existence of these extensions at runtime. An extension cannot actively push itself into the core; it must rather be passively discoverable. This calls for a discovery mechanism that combines two seemingly contradictory properties. It must automatically discover new items and provide a fully transparent account of their origins. The `ServiceLoader` from the Java standard library satisfies both requirements and is used in this project as the sole discovery mechanism.

The `ServiceLoader` has been part of the JDK since Java 6 and involves no magic. Every module that wishes to provide a contribution places a text file on its classpath whose name corresponds to the fully qualified name of the service interface, and whose contents list the fully qualified names of the implementation classes line by line. At runtime, the `ServiceLoader` queries the classpath for all occurrences of this file, loads the named classes, calls their no-argument default constructor, and returns the resulting objects. Nothing more happens. There is no reflection-based classpath scan and no annotation processing, and no hidden application context is built.

In the present article, the relevant file in both modules is named `META-INF/services/com.svenruppert.opencore.counter.extension.FeatureContribution`. Its content differs per module. In the community project, it points to `com.svenruppert.opencore.counter.ui.core.CoreFeatureContribution`; in the enterprise project, to `com.svenruppert.opencore.counter.enterprise.EnterpriseFeatureContribution`. The mere presence of the respective JAR file on the classpath is enough to activate the contribution it contains. If the enterprise JAR is removed, the application runs as a pure community edition; if it is added, its additional views, menu entries, navbar additions, and event listeners appear. The process is declarative and fully documented within an application's classpath.

The task of bringing the `ServiceLoader` result into the core is handled by the `FeatureRegistry` class. It is designed as an immutable component object that performs all its work in its constructor. The invocation of the `ServiceLoader` itself is moved into a private static helper method whose sole purpose is to translate the `Iterable` returned by the `ServiceLoader` into a conventional list.

```
1 private static List<FeatureContribution> loadViaServiceLoader() {
2     List<FeatureContribution> result = new ArrayList<>();
3     for (FeatureContribution contribution :
4         ServiceLoader.load(FeatureContribution.class)) {
5         result.add(contribution);
6     }
7     return result;
8 }
```

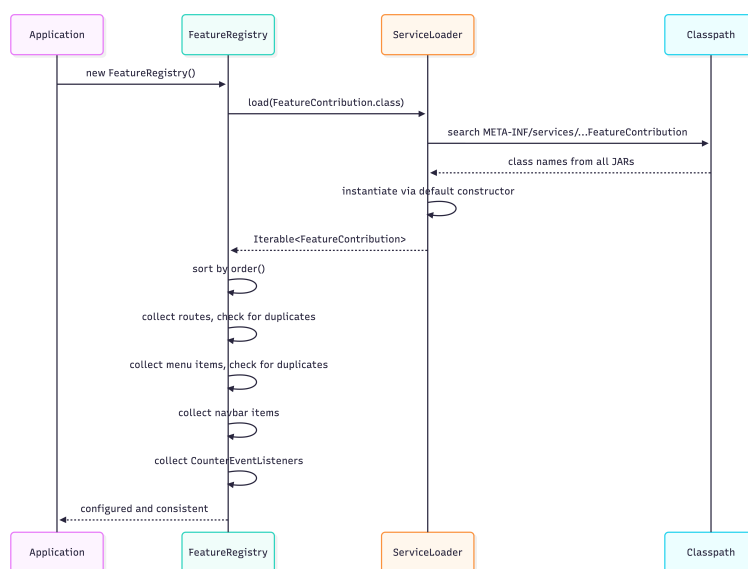


Abb. 12: Sequence of contribution discovery at startup

The loaded contributions are then ordered by their sort order. Routes, menu entries, navbar additions, and event listeners are then collected into their own, likewise immutable, lists. All checks whose violations must prevent startup also occur here. The following, therefore, holds: if a `FeatureRegistry` can be created successfully, the assembled configuration is internally consistent, and the downstream system can dispense with any further validation.

Conflict handling follows a clear rule. Duplicate route paths cause an `IllegalStateException`, as do duplicate menu entries, whereby for the latter, the combination of label and path serves as the key. This asymmetry is intentional, since two different menu entries may well point to the same path as long as they differ in the displayed label. Each exception carries an expressive message that names both the offending contribution and the feature that caused the conflict. The fact that the application aborts at startup with a meaningful error message rather than continuing with half-registered routes is the variant of the “fail fast” principle chosen here.

```

1 List<RouteContribution> collectedRoutes = new ArrayList<>();
2 Set<String> seenRoutePaths = new HashSet<>();
3 for (FeatureContribution feature : this.features) {
4     for (RouteContribution route : feature.routes()) {
5         if (!seenRoutePaths.add(route.path())) {
6             throw new IllegalStateException(
7                 "Duplicate route path '" + route.path()
8                 + "' contributed by feature '" + feature.id() + "'");
9         }
10        collectedRoutes.add(route);
11    }
12 }

```

A second observation is worth making when collecting the event listeners. Since only contributions that implement the sub-interface can be added here, the registration contribution must implement `CounterEventFeature` and only then call the relevant method. Pattern matching with `instanceof` reduces this check to a single line.

```

1 List<CounterEventListener> collectedListeners = new ArrayList<>();
2 for (FeatureContribution feature : this.features) {
3     if (feature instanceof CounterEventFeature eventFeature) {
4         collectedListeners.addAll(eventFeature.counterEventListeners());
5     }
6 }

```

In addition to the no-argument constructor used by the application, the registry offers a second, parameterised constructor that accepts a predefined list of contributions. This is used solely for testability. It allows the sort and conflict logic to be tested against hand-crafted constellations without having to deceive the `ServiceLoader` itself through classpath manipulation.

Among the various discovery mechanisms, `ServiceLoader` was preferred for several reasons. A reflection-based classpath scan, as offered by libraries such as `Reflections` or `ClassGraph`, can deliver very similar results but adds an extra dependency and leaves it to the library to decide what counts as a contribution. Annotation-based mechanisms, such as those used by Spring or CDI, shift registration from an explicit text file to an annotation in the source code. What appears shorter, however, is tied to the `rm` and makes it difficult to answer which contributions are contained in a given classpath without dedicated tooling. The `ServiceLoader`, by contrast, requires no additional dependencies, is included in every JDK, and brings all registrations together visibly in a single file. This visibility is especially valuable in the open-core context. Anyone wishing to know which modules participate in a running system can find the answer at the unpacked JAR files, without the core’s source code.

Vaadin integration: AppLayout and dynamic routes

With the mechanism described in the previous chapter, all contributions are in ordered and validated form by the time the `FeatureRegistry` exits its constructor. What remains is to translate these contributions into structures that Vaadin can actually display and evaluate. This task is divided between two components. The first is the main layout `MainLayout`, which brings menu entries into the navigation area and navbar additions into the upper area. The second is the `OpenCoreRouteInitializer`, which deposits the collected routes into the routing system at the appropriate time. Together, these two components form the only Vaadin-specific layer of the open-core build.

`MainLayout` extends `AppLayout` and is a perfectly ordinary Vaadin component, with one peculiarity. It carries the annotation `@Layout("/")`. This annotation is necessary because Vaadin relies on static analysis of the source code to produce the production frontend bundle. The bundle scanner searches the bytecode for `@Route` and `@Layout` annotations, follows their references, and determines from this the set of Vaadin components that must be included in the bundle. Since the current design has no view carrying a `@Route` `Layout` without `@Layout` would be invisible to the bundle scanner, resulting in the components and the layout not ending up in the production bundle. The annotation thus has a purely technical, build-related function and does not affect the extension mechanism.

In the layout's constructor, the navigation bar and drawer are set up. The drawer contains a `SideNav` component populated with menu entries from the registry. Since these were already ordered by sort order when the registry was built, the layout can simply iterate over the list and is freed of any concern with sorting. Each entry is turned into a `SideNavItem` whose path is prefixed with a leading slash, as the Vaadin router expects.

The navigation bar is likewise fed from the registry. If navbar additions are present, they are grouped into their own `HorizontalLayout` and placed to the right of the title. It is here that the decision described in chapter four to model `NavbarContribution` as an interface with a `Supplier<Component>` factory proves its worth. The layout calls the factory, receives a fresh component instance, and can attach it without further precaution.

```

1  @Layout("/")
2  public class MainLayout extends AppLayout {
3      public MainLayout() {
4          addToNavbar(createHeader());
5          addToDrawer(createDrawer());
6      }
7      private HorizontalLayout createHeader() {
8          H1 title = new H1("OpenCore Counter");
9          HorizontalLayout header =
10             new HorizontalLayout(new DrawerToggle(), title);
11          header.setAlignItems(FlexComponent.Alignment.CENTER);
12          header.setWidthFull();
13          var navbarItems =
14             Application.context().featureRegistry().navbarItems();
15          if (!navbarItems.isEmpty()) {
16             HorizontalLayout extras = new HorizontalLayout();
17             extras.getStyle().set("margin-left", "auto");
18             for (NavbarContribution contribution : navbarItems) {
19                 extras.add(contribution.componentFactory().get());
20             }
21             header.add(extras);
22             header.setFlexGrow(1, extras);
23         }
24         return header;
25     }
26     private VerticalLayout createDrawer() {
27         SideNav nav = new SideNav();
28         for (MenuContribution item
29             : Application.context().featureRegistry().menuItems()) {
30             nav.addItem(new SideNavItem(item.label(), "/" + item.path()));
31         }
32         VerticalLayout layout = new VerticalLayout(nav);
33         layout.setPadding(false);
34         layout.setSpacing(false);
35         return layout;
36     }
37 }

```

What is remarkable is what the layout omits. It contains no import of any class from the enterprise module, knows neither `HistoryView` nor `AuditLogView`, and makes no case distinction based on the presence or absence of extensions. The only source the layout draws from is the `FeatureRegistry`. What it delivers appears in the drawer and in the navigation bar; what it does not deliver simply is not there. It is precisely this uncompromising binding to the registry that allows the layout to function identically for both editions.

The second Vaadin integration point is the `OpenCoreRouteInitializer`. It implements Vaadin's own interface `VaadinServiceInitListener`, whose `serviceInit` method is invoked exactly once when the Vaadin service starts

up. This moment is precisely suitable for transferring the routes collected in the `FeatureRegistry` into the running Vaadin configuration. The initialiser obtains the `RouteConfiguration` responsible for the application scope, iterates over the list of route contributions, and registers each path together with the corresponding view class in the Vaadin route registry. `MainLayout` is set as the layout for each of these routes, so that all dynamically registered views appear within the same frame.

```

1 public class OpenCoreRouteInitializer
2     implements VaadinServiceInitListener {
3
4     @Override
5     public void serviceInit(ServiceInitEvent event) {
6         RouteConfiguration routeConfiguration =
7             RouteConfiguration.forApplicationScope();
8         RouteRegistry registry = routeConfiguration.getHandledRegistry();
9         registry.update(() -> {
10             for (RouteContribution route :
11                 Application.context().featureRegistry().routes()) {
12                 Class<? extends Component> viewClass = route.viewClass();
13                 if (!routeConfiguration.isPathAvailable(route.path())) {
14                     routeConfiguration.setRoute(
15                         route.path(), viewClass, MainLayout.class);
16                 }
17             }
18         });
19     }
20 }

```

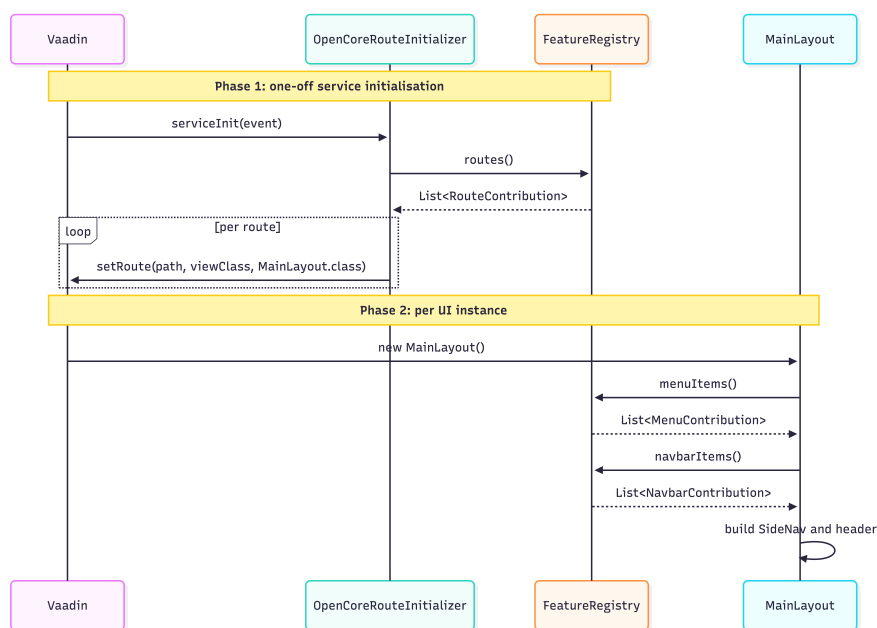


Abb. 13: Vaadin integration across two lifecycle phases

Announcing the initialiser to Vaadin once again follows the `ServiceLoader` pattern, this time along a service interface defined by Vaadin itself. The file `META-INF/services/com.vaadin.flow.server.VaadinServiceInitListener` contains a single line with the fully qualified name of the initialiser. With that, the mechanism is closed. The application's own contributions are discovered via the self-defined `FeatureContribution` service interface, while Vaadin's processing of these contributions is hooked in via Vaadin's own `VaadinServiceInitListener` service interface. Both interfaces are made discoverable via the same mechanism included in the JDK.

The deliberate omission of `@Route` annotations is the immediate consequence of this architecture. A `@Route` annotation would bind a view to a path at compile time, thereby removing it from dynamic assignment. The extension mechanism would be restricted to views that had agreed on a fixed path, thereby undermining the separation between core and exten-

sion. In the present design, by contrast, only the `RouteContribution` determines which path a view is reachable via. A contributor can change a path without affecting the view the core remains unaware of the paths of individual contributions. The `@Layout` annotation, which `MainLayout` carries as the sole exception, is not affected by this, since it does not define a path but merely announces its existence to the bundle scanner.

Transition to part two

With this, the first part of this article comes to an end. With the extension API, the `FeatureRegistry` is the central point of contact for contributions discovered at runtime and for the Vaadin-side translation into routes, menu entries, and navbar additions. The mechanism of the open-core application is fully described.

What still remains is its concrete application in the example system, that is, connecting the community edition and the enterprise extension via an embedded servlet container and continuously protected against boundary violations through tests. These practical aspects are the subject of the second part.

Read Part II here: <https://3g3.eu/ihlkl>

› [Back to the Table of Contents](#) ‹


```

1 // Serverless function in Java 25
2 import java.util.concurrent.*;
3 import java.util.function.*;
4 import jakarta.ws.rs.core.Response;
5
6 public class OrderFunction implements Function<Request, Response> {
7     @Override
8     public Response apply(Request request) {
9         try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
10             Future<Order> orderTask = scope.fork(() ->
11                 fetchOrder(request.orderId()));
12             Future<Inventory> inventoryTask = scope.fork(() ->
13                 checkInventory(request.items()));
14
15             scope.join();
16             scope.throwIfFailed();
17
18             Order order = orderTask.resultNow();
19             Inventory inventory = inventoryTask.resultNow();
20             return Response.ok(process(order, inventory)).build();
21         } catch (InterruptedException e) {
22             Thread.currentThread().interrupt();
23             return Response.serverError().build();
24         }
25     }
26 }

```



Serverless Functions Reimagined with Java 25

Serverless

Java25

VirtualThreads

StructuredConcurrency

ProjectLeyden

ZGC

Author: Bhupendra Mishra

Bhupendra is a seasoned software engineer with nearly 20 years of experience building scalable systems across industries. He's led mobile development, system architecture, and microservice transformations. Outside of work, he's a passionate marathon runner who values balance and brings focus and endurance to both tech and life.



LinkedIn

Serverless computing whether on AWS Lambda, Azure Functions, or Google Cloud Functions plays a vital role in modern distributed systems. By abstracting infrastructure away from application code, serverless platforms allow developers to focus on business logic. These functions scale automatically and can be highly cost-efficient when applied to the right use cases.

In serverless platforms, applications may be started many times a day and often run only for a few seconds. They are scaled up and down automatically and may be paused or frozen between requests. This behavior does not only affect performance; it also directly impacts cost, since billing is often based on execution time and allocated memory.

Java has traditionally struggled in this model, but Java 25 marks a significant step forward. Improvements in garbage collection, startup performance, and concurrency make Java far better suited for modern serverless workloads. Virtual Threads and Structured Concurrency allow developers to build high-throughput, low-latency serverless functions using a clear and readable programming model.

This article explores how Java 25 redefines the way serverless functions can be implemented on the JVM.

The Serverless Challenge for Java

Serverless functions are event-driven by nature. They are invoked in response to HTTP requests, message queue updates, scheduled triggers, or stream-processing events. These workloads share several common characteristics:

- Unpredictable traffic patterns
- Short-lived executions
- Strict latency requirements
- Strong sensitivity to cold-start time

Earlier Java versions often struggled in this environment. Depending on the framework used, cold-start time alone could be a major obstacle. Expensive platform threads limited scalability for I/O-heavy workloads, and garbage collection pauses further amplified latency concerns. Together, these factors made Java a challenging choice for latency-sensitive serverless applications.

Java 25 addresses these issues at the platform level.

Faster Startup and Cold-Start Improvements

Cold-start latency has long been one of the biggest concerns for Java-based serverless functions. Java 25 builds on several ongoing improvements in JVM startup performance, class loading, and runtime initialization. A major long-term effort in this area is Project Leyden, which is particularly relevant for serverless environments where startup time and memory footprint matter more than almost anything else.

Project Leyden aims to reduce startup time by shifting a significant portion of JVM initialization work from runtime to build time without abandoning the JVM. Instead of compiling Java applications into fully native executables, Leyden keeps them running on the JVM but allows them to start from a pre-optimized state.

The idea is straightforward: if the JVM already knows which classes will be loaded, how they are linked, and what the application's startup behavior looks like, it does not need to repeat that work on every invocation. This reduces repeated class loading, linking, and initialization during cold starts.

The result is faster startup and lower memory usage, which directly improves cold-start performance. Leyden can be thought of as pre-warming the JVM at build time. Applications are analyzed and prepared ahead of time so that, when a serverless platform spins up a new instance, the JVM can begin executing application code much sooner.

Importantly, Leyden preserves familiar Java features such as reflection and dynamic class loading. This makes adoption easier, as teams do not need to redesign their applications or give up widely used Java libraries. Frameworks that already minimize runtime reflection and classpath scanning such as Micronaut and newer Spring configurations are particularly well positioned to benefit as Leyden matures.

In the context of serverless functions, Project Leyden represents Java's long-term strategy for competing with languages known for fast startup, such as Go and Node.js, while retaining Java's strengths in performance, tooling, and ecosystem.

Garbage Collection and Predictable Latency

Garbage collection plays a significant role in Java performance within serverless environments. Java 25 continues to improve the JVM's ability to deliver predictable latency under short-lived, bursty workloads.

Traditional garbage collectors such as Parallel GC or even G1 GC rely on stop-the-world pauses to reclaim memory. While these pauses can be short, they are often unpredictable. In serverless functions, even small pauses can increase request latency, trigger timeouts, or increase cost, since execution is billed with fine-grained precision.

Java 25 emphasizes concurrent, low-latency garbage collection, where most GC work is performed while application threads continue to run. This is achieved through techniques such as concurrent marking and relocation, read and write barriers, and metadata embedded in object references. The trade-off is slightly higher CPU and memory overhead, but the benefit is far more predictable application behavior.

One of the most important collectors in this space is Z Garbage Collector (ZGC). ZGC is now generational, separating short-lived objects from long-lived ones. This is particularly beneficial for serverless functions, which tend to allocate many temporary objects during request handling. ZGC is designed to keep pause times very short and largely independent of heap size, helping to maintain consistent response times even during traffic spikes. It achieves this using techniques such as colored pointers to track object state with minimal global coordination.

Java 25 also includes Shenandoah GC, which performs most garbage collection work concurrently with application threads. Shenandoah typically delivers pause times in the low-millisecond range and supports compressed object pointers, reducing memory usage. This can be especially attractive in serverless environments where memory allocation directly affects cost.

For Java-based serverless functions that require consistent and predictable response times, these low-latency collectors significantly reduce GC-related latency. While they may consume slightly more CPU or memory, the improved predictability and smoother execution make Java 25 a much better fit for modern serverless workloads.

Leaner Java Objects for Serverless Workloads

Another important improvement in Java 25 is Compact Object Headers, introduced through JEP 519. Every Java object carries a header containing metadata such as locking information and garbage-collection state. Historically, these headers were relatively large, contributing to higher memory usage.

In serverless functions that process requests, parse JSON, or construct response objects, large numbers of small, short-lived objects are common. Compact Object Headers reduce the size of these headers, allowing more objects to fit into the same heap space.

- In a serverless context, this leads to:
- Lower overall memory footprints
- Faster garbage collection cycles
- Improved cache locality and CPU efficiency

These benefits reduce execution time during cold starts and burst traffic, and can lower infrastructure costs. Compact Object Headers complement low-latency garbage collectors by making heaps denser and more efficient, further strengthening Java 25's suitability for serverless architectures.

Virtual Threads for Scalable Concurrency

Virtual Threads are lightweight threads managed by the JVM rather than the operating system. They are inexpensive to create and can support very high levels of concurrency when used with Loom-aware libraries.

In a serverless environment, this is a major advantage. Each function invocation can be processed in its own virtual thread using a straightforward, synchronous programming style. When a virtual thread performs a blocking I/O operation, the JVM transparently suspends it and frees the underlying carrier thread to perform other work.

This model enables:

- Efficient handling of I/O-heavy workloads
- High concurrency without thread exhaustion
- Simple, readable code without reactive abstractions

For serverless platforms that frequently spin up and tear down execution environments, virtual threads significantly improve resource utilization and scalability.

Structured Concurrency for Reliable Execution

Serverless functions often perform multiple tasks within a single invocation, such as validation, enrichment, and downstream service calls. Coordinating these tasks reliably especially in the presence of failures can be difficult.

Structured Concurrency provides a disciplined approach to managing multiple concurrent operations as a single unit of work. Tasks launched within a structured scope are automatically tracked, and failures propagate in a predictable way.

For serverless functions, this enables:

- Clear lifecycle management for concurrent tasks
- Automatic cancellation of related work on failure
- Simpler error handling and improved reliability

Rather than manually coordinating futures or callbacks, developers can express parallelism directly while preserving correctness and clarity.

Simplifying Serverless Architecture with Java 25

Taken together, Java 25's features reduce the need for complex frameworks or custom concurrency libraries in serverless applications. With Java-native concurrency, blocking I/O becomes safe and scalable. Parallel execution is easier to reason about, and startup time improves through better memory management and runtime optimizations.

Developers can write clear, domain-oriented Java code while still meeting the performance, scalability, and cost-efficiency demands of modern serverless platforms.

Conclusion

Serverless computing demands concurrency, efficiency, and predictable performance. With Java 25, these requirements are no longer at odds with simplicity or readability on the JVM.

Virtual Threads and Structured Concurrency enable scalable and safe execution models. Improvements in startup performance and garbage collection reduce latency and operational risk. Together, these advancements make Java a far stronger candidate for serverless workloads than in previous releases.

Java 25 represents a meaningful turning point for serverless development on the JVM making it possible to build performant, maintainable, and scalable serverless functions using native Java capabilities with confidence.

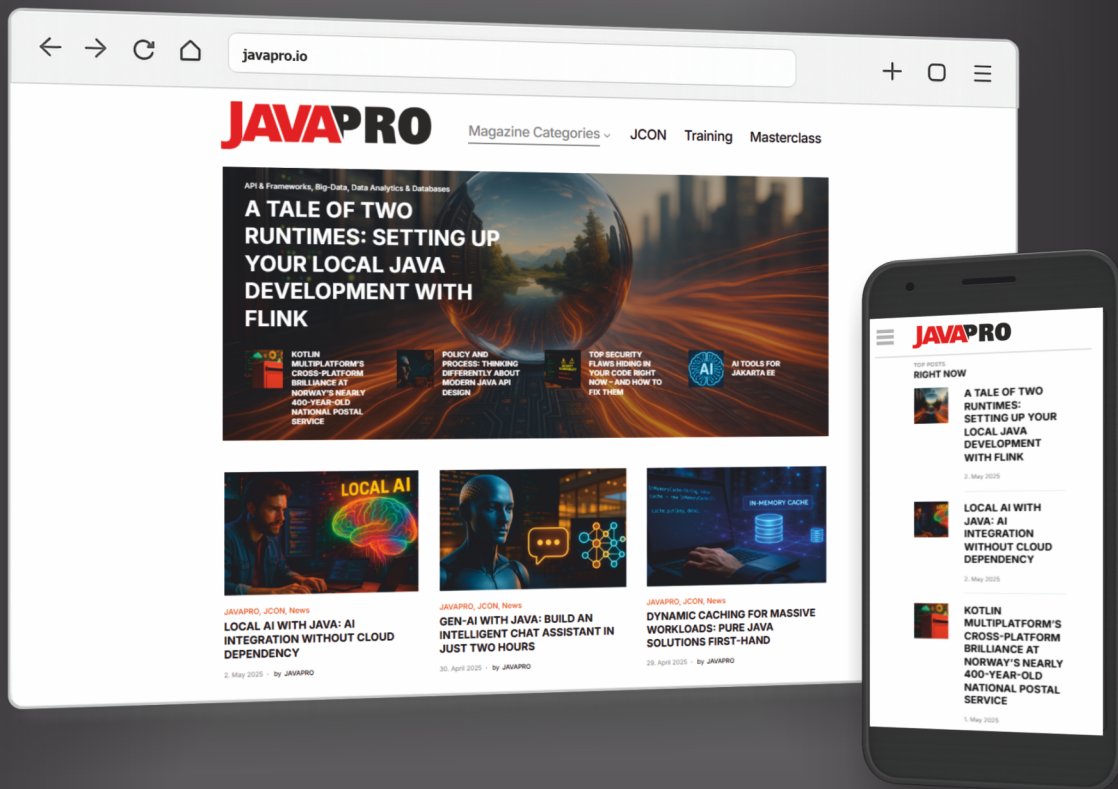
› [Back to the Table of Contents](#) ‹

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY # JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io



DevOps Patterns and Java 26 for On-Premises LLM Platforms in Safety-Critical Environments

DevOps # Java26 # RAG # Kubernetes # OnPremAI # LLMLPlatform

Authors: Cornelius May & Eldar Sultanow

Cornelius May is a Lead DevOps Engineer at Capgemini. He began his career as a freelance software developer before transitioning into consulting and training at Söldner Consult GmbH. Over the years, he has developed deep expertise in Kubernetes, Internal Developer Platforms, AI, and cloud technologies. His experience delivering training for organizations such as the Linux Foundation, combined with a wide range of professional certifications, enables him to support companies in designing, building, and operating complex infrastructure environments.



LinkedIn

As an IT strategist at Capgemini, Dr. in Business Informatics, and reviewer for the German Aerospace Center (DLR), Eldar Sultanow brings over 20 years of experience in professional software development — from programming to AI design.



LinkedIn

The first wave of AI adoption in many organizations relied on cloud APIs – but in safety-critical and regulated environments, that approach is often a non-starter. Confidentiality requirements, data protection regulations, and the need for technically enforceable operational boundaries stand in the way. On-premises setups expand the architecture with additional runtime components: LLM serving, retrieval, and knowledge artifacts such as indexes and prompts. The focus shifts from simply consuming a model to a platform question: How do you control data access and permissions along the pipeline? How do you keep changes to models, prompts, and indexes traceable? And how do you reproduce, isolate, and roll back failures?

This is where the choice of runtime becomes critical. Many of the required properties cannot be bolted on after the fact – they must be supported by runtime mechanisms and operational models from the start. Java 26 provides building blocks that can be deployed in a controlled manner within regulated environments: Structured Concurrency (Preview) for orchestrating parallel steps, the Vector API (Incubator) for more efficient CPU paths, and AOT cache improvements for shorter cold-start and warm-up behavior. What matters is not adopting every language or runtime feature unconditionally but embedding them within clear operational boundaries. Affected code paths run in encapsulated components, versions are pinned, and changes pass through defined CI/CD gates. This unlocks efficiency gains without jeopardizing the stability and traceability of the overall system.

Governance Before Feature Adoption: Stability, Support, and Blast Radius

In safety-critical environments, the question is not just whether a feature is available, but what stability and lifecycle status it carries. Preview and Incubator APIs are intentionally subject to change. They are therefore not used broadly across the codebase but deployed as calculated optimizations with clear operational boundaries – encapsulated in dedicated components, backed by a defined fallback option, and gated behind measurable release criteria such as performance and security metrics. What can be enforced deterministically are primarily the controls outside the model. Model-level guardrails tend to remain heuristic; their effectiveness must be measured in a regression-resistant way and validated as part of the release process. Throughout this article, we refer to this approach as “Preview/Incubator Governance”: encapsulation, fallback, pinned versions, and measurable release gates.

On-Premises RAG on Kubernetes

Organizations frequently choose to run large language models (LLMs) on their own infrastructure to minimize data exfiltration risks and retain full control over the processing chain. A proven pattern for these scenarios is Retrieval-Augmented Generation (RAG), where the generic knowledge of an LLM is enriched at runtime with internal enterprise data.

The Request Flow: A RAG Service in Action

To understand the technical requirements, let’s trace a single user request through a typical Kubernetes-based on-premises architecture:

1. **Ingress & Auth:** The request arrives and is authenticated.
2. **Retrieval (Fan-Out):** The service searches multiple vector indexes in parallel (e.g., “Wikis,” “Tickets”) for relevant context.
3. **Reranking:** The retrieved snippets are re-sorted, deduplicated, and trimmed to a context budget to maximize relevance for the LLM.
4. **Generation:** The enriched prompt is sent to the locally hosted model, which generates the response.
5. **Guardrails:** Before delivery, the response is checked for policy and permission violations as well as sensitive content, and redacted if necessary.

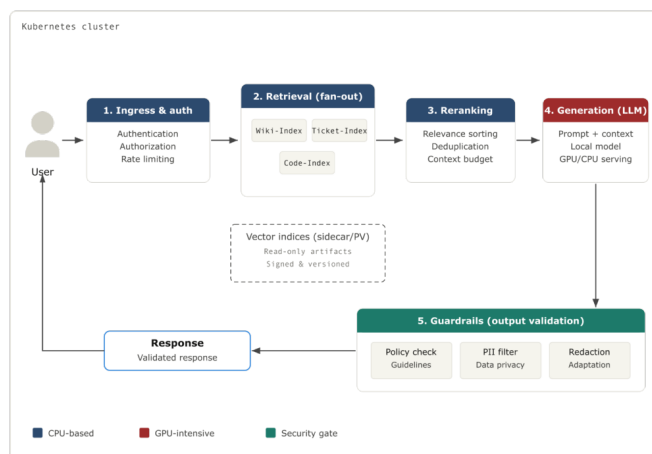


Abb. 14: Visualization of the processing steps in an LLM platform, from ingress through retrieval and reranking to LLM generation and guardrail validation.

Each of these steps costs time and resources. In cloud environments, you often just scale up the hardware. On-premises – where hardware procurement cycles are long – the efficiency of the runtime environment (here, the JVM) becomes the decisive factor.

Why “Performance” in On-Premises Usually Starts with GPU Utilization

GPU utilization often drives costs – not just through genuine idle time, but primarily through padding overhead: the GPU is technically computing, but part of that compute goes to “dead cycles” where shorter sequences in the batch are artificially padded. At the same time, idle time frequently stems not from “too little compute,” but from the fact that LLM inference is often bounded by memory bandwidth – especially in the decode phase, when weights and the KV cache must be continuously loaded. Before diving into any detailed optimization, a quick reality check pays off: Is the bottleneck in model serving (scheduling, batching, caching) or in the CPU-driven parts of the pipeline (retrieval, reranking, guardrails)? Only when that is clear does it become clear whether JVM optimizations offer the biggest lever – or whether serving-level mechanisms like more efficient batching and KV cache management should come first.

Isolation via the Sidecar Pattern

A well-established security and operations pattern is to decentralize the vector store. Instead of running a single, massive, centralized vector database cluster that every application accesses, vector indexes are treated as read-only artifacts.

A Java service can mount a compact vector index directly as a sidecar container or as a read-only persistent volume. This strengthens isolation at the deployment level: a service handling HR data receives only the HR index and cannot “see” engineering data at all. Combined with Kubernetes RBAC, namespaces, NetworkPolicies, and separate build/deploy pipelines, this creates a robust boundary between data domains.

To keep this isolation auditable, artifact governance for the index is particularly important. Indexes are signed, versioned, and annotated with provenance (source, creation timestamp, pipeline run). Rollouts proceed traceably across environments and approval gates. A lightweight “break-glass” mechanism (e.g., a feature toggle or traffic routing) allows operators to quickly fall back to a previous index version in the event of a security incident or corrupted content. This turns sidecar isolation from a mere architecture pattern into an auditable control mechanism.

That said, isolation via a sidecar (signed, read-only index) improves domain separation and auditability but is not free: large indexes drive up storage, network, and rollout times (warm-up/cache). For very large corpora or high change rates, a centralized vector store with strict tenant isolation and clean versioning may be operationally cheaper.

Java 26: Building Blocks for Efficient AI Backends

With Java 26, the OpenJDK project consolidates several initiatives relevant to these AI workloads. The combination of improvements in startup time (Project Leyden), vectorization (Project Panama), and concurrency (Project Loom) addresses specific bottlenecks in RAG pipelines.

Project Leyden: Optimized Startup via AOT Caching (JEP 516)

Java 26 extends AOT capabilities with JEP 516 (Ahead-of-Time Object Caching with Any GC). The core idea is to prepare recurring initialization states ahead of time and reuse them on the next startup. Specifically, JEP 516 adds GC-independent object caching as a building block within the Leyden/AOT cache path. This reduces cold-start costs without abandoning the operational model of a standard JVM.

In practice, the application is started in a controlled training run within the CI/CD pipeline, exercising typical initialization paths (loading the tokenizer, parsing configuration, preparing model and index metadata). The resulting reusable states are stored in a cache archive. On a subsequent pod start, the JVM can leverage these pre-built artifacts so the service reaches “Ready” faster and warm-up phases are shorter – though the exact savings depend on the workload, garbage collector, and specific initialization logic.

During load spikes, when new pods are spinning up, every second until the first successful response counts. AOT caching helps move heavy startup paths off the critical path (e.g., initializing tokenizers, model wrappers, or parsers) and makes scaling operations more predictable in production.

The Vector API (JEP 529) in the Context of the Panama Initiatives

AI inference and similarity search are grounded in linear algebra – specifically, vector operations. On-premises, not every service has a GPU at its disposal; the CPU becomes the workhorse. The Vector API (in its 11th Incubator iteration in Java 26) enables explicit use of SIMD instructions (Single Instruction, Multiple Data) on modern CPUs (AVX-512, NEON). This is particularly relevant for reranking or computing cosine similarity within your own service.

Because the Vector API is in Incubator status in Java 26, its deployment in safety-critical environments follows the Pre-view/Incubator Governance approach described above. Vectorized code paths are encapsulated as swappable accelerators (e.g., a sidecar or worker service) with a stable API contract and a configurable fallback to a scalar implementation or alternative libraries.

Code Example: Optimized Dot Product

This example demonstrates how the API harnesses parallel CPU lanes to accelerate computations – essential for latency-sensitive retrieval steps running on the CPU.

```

1 // Computes the dot product of two float vectors (e.g., embeddings).
2 // Note: Example is intentionally compact; error handling/null checks omitted.
3 static float vectorDotProduct(float[] a, float[] b) {
4     int len = Math.min(a.length, b.length);
5     var species = FloatVector.SPECIES_PREFERRED;
6     var sum = FloatVector.zero(species);
7     int i = 0;
8     int upper = species.loopBound(len); // Main loop: process SIMD blocks
9     for (; i < upper; i += species.length()) {
10         var va = FloatVector.fromArray(species, a, i);
11         var vb = FloatVector.fromArray(species, b, i);
12         sum = sum.add(va.mul(vb));
13     }
14     // Reduce vector lanes to a scalar value
15     float result = sum.reduceLanes(VectorOperators.ADD);
16     // Tail: remaining elements
17     for (; i < len; i++) { result += a[i] * b[i]; }
18     return result;
19 }

```

Project Loom: Structured Concurrency (JEP 525)

The fan-out in the retrieval step described above – searching in parallel across a wiki, Jira, and a code repository – is inherently error-prone. If one data source hangs, it should not block the entire request.

Structured Concurrency lets you model parallel retrieval tasks as a scope with explicit cancellation and timeout rules, deterministically bounding resource consumption. Since Structured Concurrency is a Preview feature, it is introduced following Preview/Incubator Governance via platform-level abstractions. This enables centralized timeout configuration, uniform cancellation propagation, and consistent resource limits. Runtime behavior is made observable through tracing and metrics (e.g., cancellation reasons, timeout rates, open tasks per request), providing operational evidence that the intended effect – controlled fan-out without hangs – is actually achieved. The scope model thus improves resilience against partial failures without attaching an unverifiable stability guarantee to Preview-status code.

Code Example: RAG Orchestration

Instead of juggling complex `CompletableFutures`, the code defines the scope and cancellation conditions declaratively.

```
1 // Orchestration of retrieval and PII check
2 try (var scope = StructuredTaskScope.open()) {
3     var retrievalTask = scope.fork(() -> vectorStore.search(query));
4     var piiCheckTask = scope.fork(() -> piiFilter.check(query));
5     scope.join(); // waits for both; cancels on failure
6     scope.throwIfFailed(); // propagates first failure
7     if (Boolean.TRUE.equals(piiCheckTask.get())) {
8         return generateAnswer(retrievalTask.get());
9     }
10 }
11 return null; // or a defined fallback
```

The operational benefit: hanging subtasks become significantly less common because the scope ensures – on exit (whether success, failure, or timeout) – that all subtasks are cleanly terminated.

DevOps Patterns for the LLM Era

The introduction of LLMs visibly changes DevOps practice: code is no longer the only artifact that needs to be versioned, tested, and rolled out in a controlled manner – prompts and models now take center stage.

GitOps for Prompts (“Prompts as Code”)

In LLM systems, prompts are behavioral logic: even minor changes to a system prompt can significantly alter response quality, formatting behavior, and the effectiveness of safety rules. Prompts should therefore not be hidden as dynamic content in databases, but treated like code – versioned, reviewable, and reproducibly deployable. In practice, this means prompts are maintained in Git and the application references a specific prompt version (e.g., v1.2). Every change becomes traceable, and rolling back the application automatically reverts the prompt to its last tested state – without manual intervention or configuration drift between environments.

Automated Semantic Testing (Evals)

Unit tests are not enough for non-deterministic LLMs. CI/CD pipelines are therefore adopting semantic evaluations (“evals”).

Typically, a stronger “teacher model” evaluates the application’s responses against a “golden set” (question–answer pairs). The deployment is approved only if metrics like “factual faithfulness” or “relevance” remain stable. Tools for this evaluation integrate well into JUnit workflows today.

To run semantic evaluations in an audit-ready manner, treat evals as regression controls with defined acceptance criteria. Beyond quality metrics (relevance, factual faithfulness), establish security metrics (PII leakage rate, prompt injection success

rate, policy violations) as gates. Version golden sets and augment them with adversarial test cases; additionally, account for the variance of non-deterministic models (e.g., multiple runs and thresholds). This produces a reproducible record that changes to prompts, retrieval configuration, or models have not silently degraded safety and compliance properties.

Security & Governance: The Five Control Points

In regulated and safety-critical environments, security should be enforced deterministically wherever possible – you cannot rely on an LLM to “behave nicely.” Guardrails based on models or classifiers (injection detection, hallucination heuristics) serve as signal providers; policy enforcement, however, must run through deterministic controls like identity/ABAC, tool allowlisting, egress control, and audit logging.

Guided by typical risk patterns (such as the OWASP recommendations for LLM applications), five control points have proven especially effective for anchoring governance and security.

1. Access & Identity

Before any retrieval or model call takes place, it must be clear who is allowed to see what information. In RAG systems, this is typically solved through identity context (user, role, tenant) and metadata filters in the retrieval layer – constraining the search space to authorized documents from the outset.

2. ****Input Guardrails****

Everything headed toward the model should be inspected first: sensitive content (PII) is masked or removed, and common prompt injection patterns are detected before they can hijack instructions in the prompt or tooling. The goal is not perfection, but a defined minimum standard that measurably reduces attacks and data exfiltration.

3. Retrieval Guardrails

The third control point sits in the retrieval layer itself. This is where it is decided which documents actually end up in the context. It is essential that only authorized sources flow in, and that data provenance remains traceable: Which source was pulled and why? Is it current? Does it match the user’s identity? And is it even approved for this use case?

4. Output Guardrails

Even when input and retrieval are clean, a model can combine or misinterpret content. The response must therefore also be validated – for instance, through structured output rules (schema/JSON), plausibility checks, and, where appropriate, a hallucination check that verifies whether key claims are supported by the provided sources.

5. Audit & Observability

In practice, the fifth control point is often the most critical. For forensic analysis and traceability, the decision chain must be unbroken: Which user input led to which retrieval hits? What context was sent to the model? What response came back? And what guardrail decisions were made along the way? Without this tracing, you can neither debug reliably nor demonstrate cleanly that a system operated within its policies.

Implementation Strategies: Criteria for Framework Selection

When building these platforms in Java, architects often face the question of which framework to choose. Rather than pitting tool names against each other, it is more productive to look at two distinct implementation philosophies: one optimized for maximum integrability in the enterprise landscape, and one that prioritizes runtime and resource efficiency in cloud-native operations.

A) The Enterprise Integrator (e.g., Spring AI)

This approach plays to its strengths when GenAI functionality needs to be woven into existing applications and platform standards. The priority is not minimal runtime footprint, but frictionless integration with established building blocks such as security, data access, configuration, and monitoring. For teams bringing AI capabilities into brownfield systems (“add AI to the monolith” or “add AI to existing services”), this is often the most pragmatic path because cross-cutting concerns are already solved consistently and new GenAI components slot into existing governance and operational processes. The higher level of abstraction can also accelerate onboarding because typical integration questions (auth, observability, configuration) don’t need to be rewired every time.

B) The Cloud-Native Performer (e.g., Quarkus with LangChain4j)

This approach focuses more sharply on efficiency: fast startup, small footprint, and an orientation toward highly scalable or short-lived workloads. It is particularly attractive when GenAI functions run as lean, specialized microservices – for example, as a reranking service, guardrail worker, or retrieval component that scales and updates independently. In such setups, resource consumption per pod and startup behavior under load often matter more than maximum framework integration depth. Additionally, the option to produce native images can be an advantage in certain operational models (e.g., many instances, aggressive scaling), making startup times and memory usage more predictable.

In practice, framework selection is rarely a “better vs. worse” decision, but a matter of weighting. If deep integration with an existing enterprise ecosystem is the priority, the integrator approach is often the faster path to production. If maximum resource efficiency and rapid scaling of specialized Kubernetes components is the goal, the performer approach has more to offer. Both directions can be combined with modern Java features. The deciding factor is which operational goals – integration effort versus runtime characteristics – dominate.

Sovereignty Through Technology

Java 26 positions itself as a capable platform for AI workloads in the backend. Its new features address three core requirements of modern RAG architectures. First, the Vector API provides the foundation for efficient CPU-based computations. Second, AOT caching via Project Leyden shortens cold-start times. Third, Structured Concurrency enables controlled, resilient parallelization.

The path to a sovereign, self-hosted LLM platform does not have to run exclusively through experimental Python services. With robust DevOps patterns and end-to-end guardrails, the power of modern Java runtimes can be harnessed safely. Innovation comes from targeted performance building blocks. Compliance is secured through architectural encapsulation, traceable approval paths, and seamless audit trails. The result is an on-premises LLM platform that operates efficiently while remaining demonstrably within defined policies.

Three Guiding Principles for Regulated On-Premises LLM Platforms

1. **Governance before feature adoption.** Preview and Incubator APIs are calculated optimizations – encapsulated in dedicated components, backed by a defined fallback option, and gated behind measurable release criteria.
2. **Determinism outside the model.** Policy enforcement runs through identity/ABAC, tool allowlisting, egress control, and audit logging. Model-based guardrails remain signal providers and must be measured in a regression-resistant fashion.
3. **Artifacts are deployment objects.** Prompts, indexes, and golden sets are versioned, signed, and rolled out through reproducible CI/CD approval paths – including rollback, without configuration drift between environments.

Further References

- OpenJDK JEP 516: Ahead-of-Time Object Caching with Any GC
- OpenJDK JEP 525: Structured Concurrency (Preview)
- OpenJDK JEP 529: Vector API (Incubator)
- OpenJDK Project Pages: Leyden, Loom, Panama
- OWASP: Top 10 for Large Language Model Applications

› [Back to the Table of Contents](#) ‹



Reproducible Dev Environments Made Easy

IDE & Tools # asdf # SDKMAN # Devbox # mise

Author: Sascha Selzer

Sascha Selzer works as Principal Engineer at OpenValue Düsseldorf GmbH. He has many years of experience in development with JVM-based languages and in software architecture. His current focus is on the design and implementation of back-end architectures, as well as continuous delivery/deployment strategies. He also deals with cloud topics such as monitoring and tracing solutions as well as related architecture and development paradigms (microservices, devops).



LinkedIn

For most developers, the term *onboarding* brings back unpleasant memories and frustration: outdated instructions in the wiki, missing dependencies, undocumented quirks, incompatible tool versions, and the notorious “works on my machine.” Whether new team members are joining a project or experienced developers are switching between teams, getting started in a new codebase often feels like a scavenger hunt that takes hours - or even days - before productive work can begin.

Anecdotal evidence and industry studies support this perception. Teams frequently underestimate how much productivity, motivation, and overall development velocity are lost due to such inconsistencies.

This is exactly where reproducible development environments come into play. They make it possible to manage toolchains, dependencies, and system configurations declaratively alongside the source code - and to reliably reproduce them anywhere: on local laptops, in CI pipelines, or inside containers - without drift and without manual adjustments.

This article introduces several approaches and tools - SDKMAN!, asdf, mise, and Nix with Devbox - and demonstrates how they can significantly reduce onboarding friction, as well as where they differ from one another.

SDKMAN! – The De Facto Standard for Java Projects

SDKMAN!¹ is a command-line tool that enables developers to install all essential tools for Java development - such as build tools and JDKs - and to manage multiple versions of them with ease. Whether switching between different Java JDK distributions or installing build tools like Maven or Gradle, SDKMAN! allows you to install and switch between versions using simple commands.

```
1 sdk list java           # lists all available JDKs
2 sdk install java 25.0.1-tem # installs Temurin JDK 25
3 sdk default java 25.0.1-tem # set Temurin JDK 25 as default
4 sdk use java 21.0.8-tem   # switches to Temurin JDK 21 for the shell
5 sdk install maven 3.9.12. # installs maven
```

The focus is clearly on the Java ecosystem, which means it is not a general-purpose package manager. However, it supports a wide range of tools typically used in Java environments. Whether build tools such as Maven or Gradle, language SDKs like Groovy or Kotlin, or analysis tools such as Java Mission Control or VisualVM - these can all be managed with SDKMAN!.

The required tools and JDKs can also be managed on a per-project basis alongside the source code. For this purpose, SDKMAN! provides the `.sdkmanrc` file, which can be committed to the Git repository. Its contents consist of a list of the tools/SDKs and their corresponding versions required for the project.

```
1 java=25.0.1-tem
2 maven=3.9.12
```

On another machine, all required tools can then be installed and activated in one go:

```
1 sdk env install # installs all tools defined in .sdkmanrc
2 sdk env         # switches to the versions specified in .sdkmanrc
```

This makes it easy to reproduce at least part of the development environment. A nice side effect is that IDEs such as IntelliJ IDEA also recognize this file and automatically set the appropriate JDK when opening the project.

SDKMAN! supports Linux, macOS, and Windows (via WSL or Git Bash), making it a solid choice for pure Java projects.

asdf – When You Need More Than Java

For ecosystems beyond Java, there are tools similar to SDKMAN!, such as rbenv for Ruby or nvm for Node.js. asdf² takes a more generic approach. Instead of focusing on a single ecosystem, it provides a foundation for managing virtually any tool or runtime.

It achieves this through a plugin-based architecture. Each plugin is responsible for a specific tool. At its core, a plugin is simply a Git repository that follows a defined structure and provides Bash scripts to list available versions of a tool and install them. Once installed, these plugins can be used to manage a wide variety of software.

For example, managing Node.js might look like this:

```
1 asdf plugin add nodejs https://github.com/asdf-vm/asdf-nodejs.git
2 asdf list all nodejs
3 asdf install nodejs 25.4.0
```

As you can see, a plugin is essentially just a Git repository with a predefined directory structure. Within that repository, scripts handle the actual commands and perform the installation. A typical plugin repository might look like this:

```
1 bin/|—
2 list-all|—
3 download|—
4 install|—
5 ...|—
6 uninstall|—
7 ...
```

Plugins exist for many popular tools, and the architecture makes it relatively easy to write custom plugins. However, this flexibility is also a drawback. When installing a plugin, you must trust the repository owner, since arbitrary commands may be executed on your machine. There is no officially curated plugin repository maintained by the asdf core team. Most plugins are community-driven, while others are maintained independently. As a result, some plugins are outdated or even archived.

Beyond trust and security concerns, performance is also a recurring discussion point. The reason lies in asdf's design decision to use so-called *shims*. For every installed tool, asdf generates a shim that intercepts the command and forwards it to the currently selected version. This gives asdf full control over where tools are installed and which version is executed. However, it introduces a small performance overhead for every invocation - insignificant in isolation, but noticeable when commands are executed frequently. Additionally, asdf does not provide native Windows support.

Similar to SDKMAN!'s `.sdkmanrc` file, asdf uses a `.tool-versions` file to define project-specific versions:

```
1 nodejs 24.13.0
```

The `.tool-versions` file does not need to reside directly in the execution directory; it can also be placed in a parent directory. This allows version management across multiple projects.

For those who find the trade-offs difficult to accept but still want the flexibility and breadth of supported tools, the next solution may offer a compelling alternative.

mise – The Better asdf?

mise3 follows a similar approach to asdf and initially even reused asdf's plugin system. However, largely due to the security concerns surrounding arbitrary plugin repositories, mise introduced additional so-called *backends* responsible for installing tools in a more controlled way. These include:

- **github** – installs releases directly from GitHub repositories
- **cargo** – installs tools via Cargo
- **go** – installs tools via `go install`

A backend can be specified explicitly when installing a tool:

```
1 mise use -g github:BurntSushi/ripgrep # github backend
2 mise use -g cargo:eza # cargo backend
3 mise use -g go:github.com/DarthSim/hivemind # go backend
```

In addition, mise ships with several *core plugins* that do not require explicitly specifying a backend in order to manage different versions. These include, among others:

- Java
- Go
- Node.js
- Rust

Installing different versions of these tools works similarly to installations via a specific backend:

```
1 mise use -g java@openjdk-21
2 mise use -g go@1.21
3 mise use -g node@20
4 mise use -g rust
```

What sets mise apart from asdf - beyond its broader selection of backends and tools - is the ability to manage additional project-specific properties, such as environment variables or custom tasks.

Alongside support for a `.tool-versions` file (compatible with asdf), mise also provides its own project configuration file that can be version-controlled with the project. A sample configuration might look like this:

```
1 [tools]
2 node = '24'
3
4 [env]
5 MY_VAR = '123'
6
7 [tasks.hello]
8 run = "echo hello world"
```

These definitions can then be used directly within the project directory:

```
1 mise exec -- echo $MY_VAR # uses the defined variable
2 mise exec -- node --version # uses the configured Node version
3 mise run hello # runs the task "hello"
```

Thanks to these features, mise can replace additional tools such as make or direnv, making it well-suited for more complex project scenarios. It also offers advanced capabilities like secret management and hooks that automatically execute actions - for example, when entering or leaving a project directory via `cd`.

Devbox – The Easier Way To Nix

With Devbox⁴, another tool enters the stage - one that clearly differentiates itself from SDKMAN!, asdf, and mise. While those tools primarily manage version-based toolchains, Devbox consistently builds on Nix⁵ as its underlying package manager, leveraging one of the most powerful approaches available for reproducible development environments.

From a technical perspective, Nix is close to ideal: packages are described declaratively, fully isolated, built deterministically, and can coexist in multiple versions side by side. Dependencies are not resolved implicitly through the operating system but are explicitly defined in a purely functional model. These characteristics make Nix exceptionally well-suited for reproducible builds and identical development environments - both locally and in CI.

In practice, however, Nix can be intimidating. Its own language, unfamiliar mental model, and concepts such as derivations, store paths, or flakes result in a steep learning curve. For many teams, adopting Nix directly is simply too complex and time-consuming - especially when the primary goal is “just” to achieve a stable development environment.

Devbox addresses exactly this challenge. It abstracts Nix in a way that preserves its greatest strengths without exposing developers to its full complexity. Installing Devbox automatically installs Nix as the underlying package manager.

```
1 curl -fsSL https://get.jetify.com/devbox | bash # installs devbox
```

Similar to `package.json` or `mise.toml`, the development environment is defined declaratively via a `devbox.json` file. All required tools and dependencies are specified on a per-project basis - independent of the global system state. Devbox handles resolution and installation in the background via Nix, including caching and precise versioning.

```
1 devbox init # creates devbox.json in the current directory
2 devbox add temurin-bin-24@24.0.2 # installs Java
3 devbox add kubernetes-helm # installs the latest Helm version
```

The generated `devbox.json` might look like this:

```
1 {
2   ...
3   "packages": [
4     "kubernetes-helm@latest",
5     "temurin-bin-24@24.0.2"
6   ]
7   ...
8 }
```

The result is a fully reproducible environment that every team member can set up identically with a single command:

```
1 devbox shell # opens a shell with all required tools
2 Starting a devbox shell...
3 (devbox)
```

Devbox also allows the definition of project-specific commands, conceptually similar to `make`. These commands are stored in `devbox.json` and automatically run inside the declared environment. While this resembles `mise`'s task feature, it goes a step further: execution is always tied to the declarative Nix environment. As a result, build, test, or startup commands can be defined reliably and without implicit dependencies.

```
1 {
2   ...
3   "shell": {
4     "scripts": {
5       "build": [
6         "./mvnw clean package"
7       ]
8     }
9   }
10   ...
11 }
```

They can then be executed directly via Devbox:

```
1 devbox run build
2 Info: Running script "build" on ...
3 ...
```

Another key feature is the ability to automatically generate a `devcontainer.json` from the Devbox configuration. This makes it possible to launch the exact same development environment not only locally but also inside a container - for example in VS Code or GitHub Codespaces. The local Nix-based environment and the container environment remain consistent, without requiring duplicate configuration maintenance.

```
1 devbox generate devcontainer # creates .devcontainer/devcontainer.json
```

Additional features such as hooks, `direnv` integration, and the automatic startup of required development services (e.g., databases or messaging systems) round out the overall package.

Conclusion

When comparing the approaches of SDKMAN!, asdf, mise, and Devbox, it quickly becomes clear that while they all pursue a similar goal at first glance, they differ significantly in focus, scope, and philosophy. The spectrum ranges from lightweight version managers to fully reproducible development environments.

Which solution fits best depends heavily on the specific project and the team's requirements. For smaller setups, a simple version manager is often sufficient, whereas more complex environments benefit from declarative and strongly isolated approaches. What matters most is this: all the tools presented here reliably serve their purpose and can significantly lower the barrier to entry for new projects - there are hardly any good reasons to forgo such tools altogether.

Ultimately, the decision is less about "right or wrong" and more about which features are truly needed and how much complexity a team is willing to embrace. From a personal perspective, mise stands out as a strong middle ground: powerful enough without becoming unnecessarily complicated, making it a highly pragmatic companion for everyday development work.

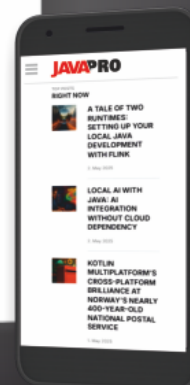
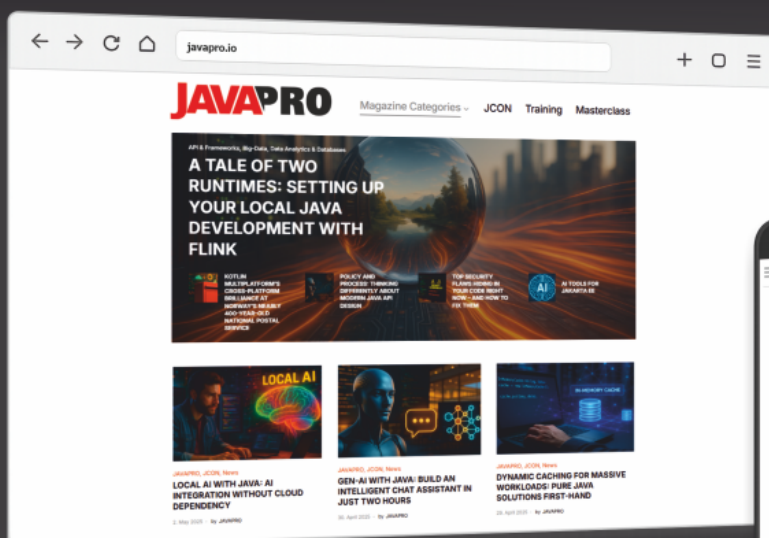
References

1. SDKMAN! – <https://sdkman.io>
2. asdf – <https://asdf-vm.com>
3. mise – <https://mise.jdx.dev>
4. Devbox – <https://www.jetify.com/docs/devbox>
5. Nixpkgs – <https://github.com/NixOS/nixpkgs>

› [Back to the Table of Contents](#) ‹

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



Code at Court – Java as Evidence

ProjectManagement

SoftwareForensics

CyberSecurity

AllInSoftwareDevelopment

Git

SBOM

Author: Stefan Wagenpfeil

Prof. Dr.-Ing. Stefan Wagenpfeil is a professor of Business Informatics, IT expert witness, and consultant specializing in sustainable software engineering, digital transformation, and gamification. He combines academic expertise with industry experience to help organizations build innovative and future-oriented software solutions.



LinkedIn

Software forms the foundation of numerous business models, industrial processes, and public administration structures today. It is created in complex ecosystems consisting of source code, data, interfaces, and architectures – and thus gains legal relevance far beyond traditional copyright issues. Allegations of manipulation, liability issues in the event of security breaches, disputes over rights of use, and even the precise reconstruction of technical processes are increasingly leading to source code and software artifacts being used as evidence in court. This article focuses specifically on the Java programming language to present the identifiability and integrity of software versions, documentation-relevant artifacts, and tools for forensic investigation. The use of AI in software development also leads to new responsibilities, growing dangers, and demonstrates the need for qualitatively reliable and verifiable software.

The Algorithmization of Law

Software is no longer just a tool for developing applications. In our modern industrial society, it is rather the backbone of business models, production processes, public administration, and almost all digital processes[1]. Its significance goes far beyond pure, almost historical, process optimization. Software is becoming a commercial product, a trade secret, a distinguishing feature; it arouses desire, creates opportunities, is subject to legal and organizational frameworks, and thus creates value of all kinds[2]. Software is no longer limited to source code – it manifests itself as a complex interplay of algorithms, databases, user interfaces, interfaces, and hardware[3]. Whereas source code was often considered the “measure of all things” in the past, its role is changing as more and more code generators are used. For example, writing data access classes, unit tests, or even developing user interfaces now involves less and less typing by developers, so that even the legal concept of “creative height” can be less and less anchored in pure source code .

Specific use cases demonstrate the complexity and relevance of software, particularly in legal contexts[4]: in the automotive industry, for example, a vehicle’s control software can not only influence engine functions, but also manipulate exhaust emissions in a targeted manner – as numerous court cases in recent years have made clear[5]. In the financial world, complex algorithms regulate the processing of stock exchange transactions – errors or manipulations of these systems can have far-reaching legal and economic consequences . In public administration, software solutions control the allocation of services and the conduct of elections, and are regularly the target of a wide variety of cyber attacks . Malfunctions or manipulation of digital systems are now considered one of the greatest risks to individuals, businesses, public administration, and even society itself.

With this ongoing digitalization, software is inevitably also finding its way into courtrooms. The source code is often only one of many technical foundations here, as the focus is usually on questions about how the system works, the interaction between systems, the traceability of decisions, security, or even questions about the rights to the solutions used . These can include copyright disputes, rights of use, liability issues following a cyberattack, allegations of manipulation—such as the circumvention of emissions regulations—or even data protection violations due to faulty programs[6]. In all cases, software has long been a central and often decisive piece of evidence in legal discourse. In numerous proceedings, IT experts are asked not only to analyze the pure source code, but also to classify the entire ecosystem of a software solution[7]. This then includes the architecture, the libraries used, the supply chain and documentation, configuration files, data and databases, as well as the interaction with external systems. This is the only way to reconstruct and evaluate the actual facts in a legally secure manner .

The Java programming language plays a special role in this context, which will also be the focus of a detailed examination below. Due to its widespread use in enterprise applications, financial systems, critical infrastructure, and also in the form of Android applications on countless smartphones, it is disproportionately often the focus of forensic investigations[8]. But how can code be “interrogated”? How can abstract classes and lambda expressions be translated into a language that a judge can use to make a ruling? What if AI in all its forms also comes into play?

To answer these questions, this article examines the general legal framework and, above all, the methodology in the context of Java applications and from the perspective of Java developers in order to generate secure, traceable, and unassailable source code.

Legal framework

Under German and European law, source code is generally protected as a work in the sense of copyright law, provided it reaches a certain level of creativity[9]. This means that the copyrights to the source code initially belong to the developer, but can also be transferred to companies on the basis of contracts or employment relationships. This in turn results in exclusive rights, such as the right of reproduction, distribution, and editing, which enable the rights holder to control the

use and distribution of the source code and, if necessary, to enforce this legally. Alarming often, lawsuits arise in which one party accuses the other of either stealing or copying source code or using components unlawfully. The granting of rights of use, in turn, entails guarantees, warranties, the obligation to correct errors, and much more, which can regularly cause manufacturers distress.

While developers used to do most of the source code production themselves—and some of the case law still dates from this period—modern tools such as code generators now take care of a certain amount of the typing work. However, since the advent of AI and AI-generated source code, an additional aspect has become increasingly important: AI-generated works are initially free of rights, i.e., the developer does not even own the copyright in the first step, since the AI system itself is not a natural person and the code it generates cannot therefore be protected[10]. However, it must be taken into account at this point that the source code is typically embedded in complex system architectures and its creation is also significantly controlled by developers. This creative process, in turn, represents a certain level of creativity. Many lawyers argue that this embedding or prompting can indeed generate protection. However, numerous legal proceedings are currently working to resolve the resulting uncertainty regarding the enforceability of claims or the usability of such source code.

Identifiability, Security, and Integrity

Before an analysis can begin, the procedural integrity of the evidence must be ensured. Unlike physical evidence, however, digital information is fleeting and easily manipulated. For an expert, the work therefore does not begin with reading the code, but much earlier – namely with the acquisition of the source code for the period in dispute.

Considering that several years may pass between the trigger of the dispute and a court hearing, this is a difficult undertaking in practice. In civil or criminal proceedings, for example, it must be proven beyond doubt that the source code under investigation corresponds exactly to the status at the time of the damaging event. If this is not possible, a central pillar of the argumentation is often lost.

Thanks to modern tools such as Git[11], source code can now be reconstructed to any historical state. This is a considerable advantage in the event of an incident. If cryptographic hash functions (such as SHA-256) are also used, the status in the repository can be provided with a “digital fingerprint,” thus proving that any subsequent changes or manipulations would be immediately noticeable.

However, software is not just pure source code – from the aspect mentioned here, it is also advisable, for example, to regularly store documentation, database dumps, architecture documents, test cases, and test results in the repository so that, together with the source codes, they form a consistent and complete status.

In this way, complete, uniquely identifiable software can be made available at any time. Developers should therefore document each delivery of a version via tag and hash value and check in the corresponding data and documentation status.

Source Code Changes

Git also plays an important role in another aspect: companies or individual developers are increasingly being accused of illegally taking source code, e.g., from their previous employer, and integrating it into the products of a new employer. In the rarest of cases, these are 1:1 copies. However, if large parts of the source code are similar—regardless of the programming language used—suspicions may arise that a developer has copied C# source code, for example, and then simply rebuilt it in Java and integrated it into the new employer’s software.

In such cases, it is extremely helpful to be able to consult the Git history. This often allows the creation and genesis of source code to be very well documented and traced. For example, if an unusual 10,000 lines were checked in during a commit, this could be an indicator of stolen source code. However, if the Git history proves and documents that the source code has been developed in “commercially available” quantities, and if conversion and dismantling measures are visible in Git, this can be used to develop a good argument for the independent and original creation of source texts. Of course, there is no need for concern about automatically generated source code, such as test cases or data access classes, as experts are able to distinguish between these and such structures have only a very low level of creativity. The “Gource” project (<https://gource.io>) has elevated the visualization of source code changes to a true art form by visualizing the Git history as a film.

However, this also means that developers should check in their results regularly. Comments and explanations in the source code are also helpful.

Analysis Methodology – From Bytecode to Business Logic

The analysis of Java applications differs fundamentally from the examination of native languages such as C++ or C. The concept of the Java Virtual Machine (JVM) offers both advantages and hurdles in this regard.

Often, courts do not have access to cleanly documented source code, but only to a precompiled jar, war, or ear archive. These can be translated back into source code with high precision using decompilers. Tools such as the CFR Decompiler or Fernflower make it possible to reconstruct readable source code from binary files. However, this approach also has its limitations. If obfuscators are used, for example, all methods, variables, and class names are replaced by incomprehensible and cryptic identifiers, which significantly reduces readability. However, in modern software architectures, these have the major problem that they make the development of services, APIs, or reusable components much more difficult, because methods and structures must be preserved for this very purpose. For special, critical areas, however, obfuscators offer additional security and protection against decompilation into readable (!) source code.

If source code is available either in its original form or through decompilation, the expert then checks the logical consistency, software quality, and hidden functions, or compares the source code in the context of the evidence question.

In this context, it is also important to mention the aspect of security checks. There are often disputes as to whether a given source code meets the (hopefully jointly) defined quality criteria. CVE databases (Common Vulnerabilities & Exposures) with associated vulnerability scanners[12], such as Gripe, Trivy, and SonarQube, offer convenient tools for automatically detecting such vulnerabilities. However, the problem here is that it is almost never a question of evaluating current vulnerabilities, but rather of evaluating software at a specific point in time. This means that the CVE database must also originate from time X in order to be comprehensible, verifiable, and fair. This requirement can be met by linking snapshot repositories (e.g., NVD) or archived scans[13].

Developers should therefore consider using obfuscators and regularly try decompiling their own applications to gain an understanding of what information third parties might be able to access if they obtain jar, war, or ear files.

Developing and documenting an understanding of the dynamic aspects of software is significantly more complicated than pure, static source code analysis. In order to be able to reproduce error states, it is usually not enough to follow the algorithmic approach. Rather, framework conditions such as configuration, data quality, infrastructure, and even hardware conditions play a role. In this context, tools such as AspectJ, Eclipse Memory Analyzer, or—if possible—remote debuggers are important for developing an understanding of the runtime behavior of applications. These can be very helpful not only for checking the quality of applications, but also during development or testing.

Ultimately, it is important to remember that modern applications are almost never developed from scratch. They consist largely of libraries, in the Java context mostly even open source libraries. However, this does not mean that a developer's responsibility ends with their own source code. Rather, the complete dependency tree of a software, the Software Bill of Materials (SBOM), must be included in the review. The best example of why this can be important is outdated libraries with security vulnerabilities. Regular testing of the SBOM and regular updating of components is part of professional software development and can be held against the application manufacturer in the event of improper implementation. The security vulnerability in the popular logging framework Log4j was a prominent example and caused difficulties worldwide.

It is strongly recommended to continuously review the SBOM of applications and incorporate the results into the software. Especially in open source projects, developers who use the software are even more responsible for the functionality of their own components, as there is no manufacturer with corresponding contracts who could be held liable in case of doubt.

Artificial Intelligence

With open source code, data, and documentation, quality or problem assessments can usually be carried out with sufficient expertise. However, all this becomes more difficult when artificial intelligence comes into play. AI-generated source code raises completely new questions: for example, who is responsible if an AI suggests an insecure Java method? Copyright law does not play a major role in terms of responsibility either, because, similar to the review of the SBOM, the developer has a clear responsibility to review AI-generated results when using AI. At this point, however, the developer must also have the relevant expertise, and this is becoming increasingly problematic: Vibe Coding, for example, specifically addresses non-developers by using functional descriptions to prompt an AI to independently create source code that cannot be verified by untrained “developers,” from which applications can then be created. Here, too, the developer is responsible for the solution, even though the whole concept of “vibe coding” is precisely designed to ensure that they do not have to assume this responsibility. Unfortunately, the use of AI in software development is also increasingly leading to junior developers

relying too heavily on AI and no longer being able to evaluate the results. Such cases can quickly become problematic for developers.

When using AI, developers should therefore be able to check and evaluate the results. Otherwise, problems and, in some cases, liability risks are inevitable. In addition, it is essential to document which parts of the code are AI-generated and which are human-made. This can also be important for later training or optimization processes on your own code base, as AI should not be trained with AI-generated data if possible.

Conclusion and Outlook

“Code in court” will increasingly become a standard scenario in the coming years. For software companies, this means that quality assurance and documentation of source code and related artifacts are not only a technical necessity but also essential risk prevention. Java’s structure and excellent analysis tools provide a good basis for objective investigations and quality standards. In court, this allows for the creation of a comprehensible and verifiable expert opinion.

The human component is becoming increasingly crucial in this context. Companies and developers today need to know which components can pose a risk and what measures can be taken to manage this risk. This article has listed a number of specific recommendations that are relevant at different levels and at different stages of the development process. If these are taken into account, they can provide a solid basis not only for high-quality software, but also for good arguments in the event of a dispute. Because at the end of the day, we all hope that your code won’t have to be discussed in court.

Prof. Dr.-Ing Stefan Wagenpfeil

SLW – IT-Experts, Brühl

PFH Private University of Applied Sciences, Göttingen

Literature:

- [1] Bundesministerium für Wirtschaft und Energie (BMWi), *Digitale Transformation erfolgreich umsetzen – Herausforderungen und Erfolgsfaktoren für Industrieunternehmen*, 2020.
- [2] Fraunhofer ISI, Metzger, F. M. (2023): *Digitale Geschäftsmodelle: Zugrundeliegende Trends und kennzeichnende Charakteristika*.
- [3] Buxmann, Diefenbach & Hess (2015): *Geschäftsmodelle in der Softwareindustrie*. Springer.
- [4] Wagenpfeil (2026): *Nachhaltiges Software-Engineering*. Springer.
- [5] EuGH (2020): *Abgasmanipulationssoftware ist illegal*.
- [6] FINRA (2024): *Algorithmic Trading – Risiken & Regulierung*.
- [7] FES / USAID (2023): *Elections and Cybersecurity Reference Document*.
- [8] Ferner, J. (2022): *Quelltext als digitales Beweismittel*. Ferner Alsdorf. [URL](#)
- [9] UpCounsel. (2025, August 26): *Forensics software in legal and cyber investigations*. [URL](#)
- [10] Braguinski, L. (2025): *Urheberrecht vor Gericht: Wie soll ich beweisen, dass mein Code nicht KI-generiert ist?* RSW/Beck. [URL](#)
- [11] Chacon, S., & Straub, B. (n.d.): *Pro Git (2nd Edition)*. Git-Projektseite. [URL](#)
- [12] CVE Program. (n.d.): *Common Vulnerabilities and Exposures (CVE): Global reference standard for publicly disclosed cybersecurity vulnerabilities*. [URL](#)
- [13] National Vulnerability Database (NVD). (n.d.): *NVD Data Feeds: Annual CVE feeds and modified feed update guidance for local mirroring*. [URL](#)

› [Back to the Table of Contents](#) ‹



RTFM – Usable Documentation

ProjectManagement

DevOps

Agile

KnowledgeSharing

SoftwareEngineering

Author: Marco Schulz

Marco Schulz, also known on the Internet under his pseudonym Elmar Dott, has been implementing large web applications as a freelance consultant in international projects for over 20 years. His focus is on DevOps, configuration management, software architectures & release management. As a trainer, he shares his knowledge in training courses and also speaks regularly on current topics at conferences.



LinkedIn

Newcomers are often fobbed off with the phrase: “It’s all in the wiki.” But anyone with a bit of professional experience knows that a glance at the existing documentation can quickly lead to disappointment. In this article, I’ll show you ways to do it better.

Why Documentation Matters

An old master craftsman used to say: “He who writes, remains.” His primary intention was to obtain accurate measurements and weekly reports from his journeymen. He needed this information to issue correct invoices, which was crucial to the success of his business. This analogy can also be readily applied to software development. It wasn’t until Ruby, the programming language developed in Japan by Yukihiro Matsumoto, had English-language documentation that Ruby’s global success began.

We can see, therefore, that documentation can be of considerable importance to the success of a software project. It’s not simply a repository of information within the project where new colleagues can find necessary details. Of course, documentation is a rather tedious subject for developers. It constantly needs to be kept up-to-date, and they often lack the skills to put their own thoughts down on paper in a clear and organized way for others to understand.

The Problem with Documentation as a “Victorian Novella”

I myself first encountered the topic of documentation many years ago through reading the book “Software Engineering” by Johannes Siedersleben. Ed Yourdon was quoted there as saying that before methods like UML, documentation often took the form of a Victorian novella. During my professional life, I’ve also encountered a few such Victorian novellas. The frustrating thing was: after battling through the textual desert—there’s no other way to describe the feeling than as overcoming and struggling—you still didn’t have the information you were looking for. To paraphrase Goethe’s Faust: “So here I stand, poor fool, no wiser than before.”

Here we already see a first criticism of poor documentation: inappropriate length and a lack of information. We must recognize that writing isn’t something everyone is born with. After all, you became a software developer, not a book author. For the concept of “successful documentation,” this means that you shouldn’t force anyone to do it and instead look for team members who have a knack for it. This doesn’t mean, however, that everyone else is exempt from documentation tasks. Their input is essential for quality. Proofreading, pointing out errors, and suggesting additions are all necessary tasks that can easily be shared.

Writing Skills Are a Team Capability

It’s highly advisable to provide occasional rhetorical training for the team or individual team members. The focus should be on precise, concise, and understandable expression. This also involves organizing your thoughts so they can be put down on paper and follow a clear and coherent structure. The resulting improved communication has a very positive impact on development projects.

Up-to-date documentation that is easy to read and contains important information quickly becomes a living document, regardless of the format chosen. This is also a fundamental concept for successful DevOps and agile methodologies. These paradigms rely on good information exchange and address the avoidance of information silos.

Why Tests Alone Are Not Documentation

One point that really bothers me is the statement: “Our tests are the documentation.” Not all stakeholders can program and are therefore unable to understand the test cases. Furthermore, while tests demonstrate the behavior of functions, they don’t inherently demonstrate their correct usage. Variations of usable solutions are also usually missing. For test cases to have a documentary character, it’s necessary to develop specific tests precisely for this purpose. In my opinion, this approach has two significant advantages. First, the implementation documentation remains up-to-date, because changes will cause the test case to fail. Another positive effect is that the developer becomes aware of how their implementation is being used and can correct a flawed design in a timely manner.

Tooling Can Help, But It Cannot Replace Discipline

Of course, there are now countless technical solutions that are suitable for different groups of people, depending on their perspective on the system. Issue and bug tracking systems, such as the commercial JIRA or the open-source Redmine, map entire processes. They allow testers to assign identified problems and errors in the software to a specific release version. Project managers can use release management to prioritize fixes, and developers document the implemented fixes. That's the theory. In practice, I've seen in almost every project how the comment function in these systems is misused as a chat to describe the change status. The result is a bug report with countless useless comments, and any real, relevant information is completely missing.

Wikis Need Structure, Formatting, and Care

Another widespread technical solution in development projects is the use of enterprise wikis. They enhance simple wikis with navigation and allow the creation of closed spaces where only explicitly authorized user groups receive granular permissions such as read or write. Besides the widely used commercial solution Confluence, there's also a free alternative called BlueSpice, which is based on MediaWiki. Wikis allow collaborative work on a document, and individual pages can be exported as PDFs in various formats. To ensure that the wiki pages remain usable, it's important to maintain clean and consistent formatting. Tables should fit their content onto a single A4 page without unwanted line breaks. This improves readability. There are also many instances where bulleted lists are preferable to tables for the sake of clarity.

Graphics Must Add Value

This brings us to another very sensitive topic: graphics. It's certainly true that a picture is often worth a thousand words. But not always! When working with graphics, it's important to be aware that images often require a considerable amount of time to create and can often only be adapted with significant effort. This leads to several conclusions to make life easier. A standard program (format) should be used for creating graphics. Expensive graphics programs like Photoshop and Corel should be avoided. Graphics created for wiki pages should be attached to the wiki page in their original, editable form. A separate repository can also be set up for this purpose to allow reuse in other projects.

If an image offers no added value, it's best to omit it. Here's a small example: It's unnecessary to create a graphic depicting ten stick figures with a role name or person underneath. Here, it is advisable to create a simple list, which is also easier to supplement or adapt.

But you should also avoid overloaded graphics. True to the motto "more is better," overly detailed information tends to cause confusion and can lead to misinterpretations. A recommended book is "Documenting and Communicating Software Architectures" by Stefan Zörner. In this book, he effectively demonstrates the importance of different perspectives on a system and which groups of people are addressed by a specific viewpoint. I would also like to take this opportunity to share his seven rules for good documentation:

1. Write from the reader's perspective.
2. Avoid unnecessary repetition.
3. Avoid ambiguity; explain notation if necessary.
4. Use standards such as UML.
5. Include the reasons (why).
6. Keep the documentation up-to-date, but never too up-to-date.
7. Review the usability.

Keep Documentation Concise, Centralized, and Maintainable

Anyone tasked with writing the documentation, or ensuring its progress and accuracy, should always be aware that it contains important information and presents it correctly and clearly. Concise and well-organized documentation can be easily adapted and expanded as the project progresses. Adjustments are most successful when the affected area is as cohesive as possible and appears only once. This centralization is achieved through references and hyperlinks, so that changes in the original document are reflected in the references.

Conclusion: Documentation Enables Agile and DevOps

Of course, there is much more to say about documentation; it's the subject of numerous books, but that would go beyond the scope of this article. My main goal was to raise awareness of this topic, as paradigms like Agile and DevOps rely on a good flow of information.

Reference: <https://elmar-dott.com/articles/rtfm-usable-documentation/>

› [Back to the Table of Contents](#) ‹



APR
26-29

www.jcon.one

JCON2027

JCON EUROPE 2027

International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



The Whispering JAR: Java Security Lessons Hidden in a Fantasy Tale

Security

SupplyChainSecurity

CVE

OWASP

Author: Mohammad-Ali A'râbi

JCON SPEAKER

Mohammad-Ali A'râbi is Software Engineer, Docker Captain, Author of "Docker and Kubernetes Security", Gamer, especially a huge fan of Fallout: New Vegas and Mortal Kombat series

Want to Dive Deeper?

Mohammad-Ali A'râbi is a speaker at [JCON](#)! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn

It's December 29th, 1865, and a woman is walking down *Salzstraße* in the city of *Freiburg* in the Duchy of *Baden*. She has a green cloak on, and her face is not very visible in the thick fog. She enters a small alley and knocks. A man walks to the door and asks, "What's in the jar?" The woman replies, "Java beans."



Abb. 15: Woman with a hood and red hair and a plate armor walking in the streets of Freiburg in 1865

The door opens, and a man with a red fedora hat and dark brown eyes welcomes her inside. "Come in, Gord," he says. "I have been expecting you."

"Good to see you, Rothühle," she replies.

They enter the room, and the woman sees a pile of newspapers on the table.

"Coffee?" the man asks. "Yes, please," Gord replies.

"I'm about to run out of beans, so I bought a new batch from the market," the man says, grinding the beans and brewing a fresh cup.

Gord stares at the jar of coffee on the table for a moment, as if something is off.

Threats from the Past

Rothühle's obsession with old newspapers was not nostalgia—it was vigilance. Some threats never truly die.

In the story, the warrior woman named Gord is meeting with her new acquaintance Rothühle, who is a tinkerer and technologist. The man is going through old archives and newspapers, because threats from the past can still be relevant today.

Many entered the realm of **supply chain security** because of the **Log4Shell** vulnerability, which was a critical security flaw in the widely used Java logging library **Log4j**. The vulnerability was discovered in late 2021 and allowed attackers to execute arbitrary code on affected systems by sending specially crafted log messages. This incident shook the software industry and highlighted the importance of securing the software supply chain, as Log4j was a dependency in countless applications and services.

Let's explain a few technical terms here:

- **CVE** or **Common Vulnerabilities and Exposures**, is a system that provides a reference method for publicly known information security vulnerabilities and exposures. Each vulnerability is assigned a unique identifier, known as a CVE ID, which allows security professionals to easily reference and discuss specific vulnerabilities.
- **Log4Shell** is a critical security vulnerability (CVE-2021-44228) in the Log4j library. You can look up the details of this vulnerability in the CVE database at cve.mitre.org.
- **CVSS** or **Common Vulnerability Scoring System**, is a standardized framework for rating the severity of security vulnerabilities. It provides a numerical score (ranging from 0 to 10) that reflects the potential impact of a vulnerability, with higher scores indicating more severe vulnerabilities. Log4Shell had a CVSS score of 10, which is the highest possible score, indicating that it was an extremely critical vulnerability.

Research from 2025 shows that many years after the Log4Shell incident, approximately **13% of all Log4j downloads** are of vulnerable versions. The number varies across countries, and for example, it's 11% in Germany and 29% in India. This means that even years after the vulnerability was disclosed, a significant portion of applications are still using vulnerable versions of Log4j, exposing them to potential attacks. [1] [2]

That's why we need to know about the threats from the past.

Tip of the day. Fix your vulnerable dependencies, as soon as possible. CVEs are known to security professionals and attackers alike.

You can find vulnerable dependencies in your applications by using tools like **Snyk**, **Dependabot**, or **Renovate**. If you have a Docker image, you can use tools like **Trivy** or **Docker Scout** to scan for vulnerabilities in the image layers. Once you identify vulnerable dependencies, you should update them to the latest secure versions and rebuild your images to ensure that the fixes are applied.

To check your Java applications for Log4Shell vulnerabilities, you can use the following command with Trivy:

```
1 $ trivy fs --severity CRITICAL --ignore-unfixed --pkg-types os,library <path-to-your-java-app>
```

To use Docker Scout, you can run:

```
1 $ docker scout cves --only-severity CRITICAL --only-fixed <image>
```

The Fishy JAR

As Rothütle is making coffee, Gord notices that the jar of coffee beans is whispering.

She sits down and listens carefully. Then she puts her cloak on over the jar, and sees a pair of eyes staring at her from the darkness.



Abb. 16: A woman is staring at a coffee jar on the table. In the background, a man with a red fedora is making coffee.

“Angra’s malice is poured into the jar,” Gord says. “Where did you get it?”

“I bought it right in front of the door,” Rothütle replies. “The merchant just left the customs and was selling it on the street for a better price.”

Tip of the day. Beware of untrusted sources and social engineering.

In the story, Gord and Rothütle are on high alert because they have been targeted by attackers before. The software security scene of 2026 is no different. Let’s take a look at the supply chain attack in the last 4 months of 2025.

The NPM Supply Chain Attack

The world's largest NPM supply chain attack happened in September 2025. Attackers started a phishing campaign by registering the domain `npmjs.help` and sending out emails to the maintainers of the top 1% of NPM packages. The attackers also used a real-time proxy to bypass the two-factor authentication (2FA) and gain access to the maintainers' accounts. Some maintainers fell for the scam, and within hours the attackers took control of popular packages like `chalk` and `debug`.

The attackers published malicious updates to these packages, which contained a crypto-drainer that swapped wallet addresses during cryptocurrency transactions in web browsers. The attack was active for about two hours before it was mitigated, but it had the potential to affect millions of users who installed or updated these packages during that time. [3] [4]

Shai Hulud Attack, All the Way from Arrakis

On September 15th, 2025, a week after the initial NPM attack, a second, more sophisticated threat emerged: the **Shai Hulud worm**. Named after the colossal sandworms of Frank Herbert's *Dune*, this malware was not designed for simple theft; it was designed for total ecosystem dominance. Ironically, the name "Shai Hulud" means "the immortal thing" in Arabic, and this worm lived up to its name by being nearly impossible to eradicate.

The worm represented a terrifying evolution in supply chain attacks: it was self-replicating.

Infection Cycle:

1. **Entry:** A developer installs a compromised package (e.g., `@ctrl/tinycolor` or `ngx-bootstrap`).
2. **Harvesting:** The malicious script (`setup_bun.js`, often utilizing the Bun runtime for speed and obfuscation) scans the developer's local environment. It targets:
 - `~/.npmrc` (NPM authentication tokens).
 - `~/.ssh/id_rsa` (SSH keys).
 - `~/.aws/credentials` (Cloud access keys).
 - GitHub Personal Access Tokens (PATs). [5]
3. **Propagation:** Using the stolen NPM tokens, the worm authenticates to the NPM registry as the victim. It then:
 - Enumerates all packages maintained by the victim.
 - Injects itself into the `preinstall` or `postinstall` scripts of those packages.
 - Publishes a new patch version (e.g., `v1.0.1` -> `v1.0.2`) to the registry.
4. **Exfiltration:** Stolen credentials were not merely sent to a Command and Control (C2) server. In a display of brazen arrogance, the worm created **public GitHub repositories** under the victim's account, named `Shai-Hulud` or `Shai-Hulud: The Second Coming`, and uploaded the credentials there as a file named `secrets.txt`.

React2Shell

In December 2025, when the industry was still reeling from the Shai Hulud worm, the **React2Shell** vulnerability (CVE-2025-55182) was disclosed. It was a Critical (CVSS 10.0) vulnerability affecting **React Server Components**, specifically in React 19.x and Next.js 15.x/16.x. The vulnerability allowed attackers to execute arbitrary shell commands on the server by sending specially crafted requests to vulnerable endpoints. This was particularly dangerous because React Server Components are often used in server-side rendering (SSR) setups, which have direct access to the server's filesystem and environment.

When the vulnerability was disclosed, a race began between attackers and defenders. Attackers quickly developed proof-of-concept exploits, while defenders scrambled to patch their applications and update their dependencies. In a talk in December 2025, Docker Captain and Co-founder of European container hosting solution Sliplane, Jonas Scholz, revealed that within 72 hours, almost all of the unpatched React applications were being targeted by cryptocurrency miners, which were using the vulnerable servers to mine cryptocurrencies without the knowledge of the server owners. [6]

ATTACK NAME	DATE	VECTOR	PRIMARY TARGET	MECHANISM	IMPACT
NPM Supply Chain	Sept 8, 2025	Social Engineering (Phishing)	Frontend Devs	Malicious npm package update (e.g., chalk)	Crypto-wallet draining (browser-based)
Shai Hulud Worm	Sept 15, 2025	Self-Replicating Worm	CI/CD & Devs	Harvests creds, injects preinstall scripts	Destructive (Wipes \$HOME), Credential Exfiltration to GitHub
React2Shell	Dec 3, 2025	Vulnerability (RCE)	Servers (Next.js/React)	Unsafe Deserialization (Flight Protocol)	Full Server Compromise (CVSS 10.0)

The Worm Spreading

Gord pulled out a knife and cut the jar open. A sandworm-like creature emerged from the jar.



Abb. 17: Woman is sitting at the table, the coffee jar is open and a worm in the shape of Shai Hulud is crawling out. The man with the red hat is in the distance.

“It’s a Shai Hulud,” Rothütle says. “They dwell in the deserts of Arrakis, not sure how it ended up here.” Gord looks at the creature and says, “Borders are thinner than we think.”

Tip of the day. Look into your JARs.

Shai Hulud is a Node.js worm. Why would someone in the Java world care about it?

The Bridge Vector. Modern Java development rarely happens in isolation. Developers use tools like **WebJars** or **mvnrm** to package JavaScript libraries as JAR files for use in Maven/Gradle builds.

In November 2025, it was discovered that the Shai Hulud worm had crossed into **Maven Central**. The tool **mvnrm** (Maven NPM) automatically monitors the NPM registry. When a new version of a package is published on NPM, **mvnrm** automatically wraps it in a JAR and publishes it to Maven Central. When Shai Hulud compromised **posthog-node** on NPM, **mvnrm** inadvertently acted as a bridge, wrapping the malicious **setup_bun.js** payload into **org.mvnrm:posthog-node:4.18.1**. [7]

The Box of Sand

Rothütle says, “We can’t kill this creature with a mere knife. Let’s contain it for now to prevent it from spreading.”



Abb. 18: The man with red fedora is placing the worm in a box full of sand. The woman with red hair is looking closely.

Gord says, “Let’s put it in this box of sand, where it feels at home. We can keep it there until we find a way to neutralize it.”

One way to contain supply chain threats like Shai Hulud is to **sandbox** the development environment. Shai Hulud bridged the gap between development and production environments. So far, everyone has taken good care of securing their production environments, but the development environments were left exposed.

- **Containerized development environments.** By running your development environment inside a container, you can reduce the blast radius of a potential compromise. If a malicious package tries to access your credentials or filesystem, it will be contained within the container, preventing it from affecting your host system. Still, if you use volumes to share files between the container and the host, you need to be cautious, as this can potentially allow the malicious code to access sensitive files on your host system.
- **Remove development environments.** Many solutions, like **GitHub Codespaces** or **Gitpod**, provide cloud-based development environments. By using these, you can avoid running potentially vulnerable code on your local machine. However, you need to ensure that the cloud environment is properly secured and that sensitive credentials are not exposed in the cloud.
- **Keep your credentials out of reach.** Use secret management tools to keep your credentials out of your codebase and filesystem. Make publishing packages more difficult by setting up manual approval processes for package updates, and avoid using tools that automatically publish packages to the registry.
- **Hardware keys.** Enforce using hardware security keys (e.g., YubiKey) for authenticating to package registries. This adds an extra layer of security, making it harder for attackers to gain access even if they compromise your credentials.

Conclusion

Rothütle spends the next few days researching the creature, trying to find a way to neutralize it. He discovers that the creature is vulnerable to water, which is ironic given that it thrives in the desert. They decide to fill the box with water to contain the creature.

Now, although the worm is neutralized, the source must be identified and eradicated. The attack vector must be closed to prevent future infections. The incident serves as a stark reminder of the evolving nature of supply chain threats and the importance of vigilance in securing the software development lifecycle.



Abb. 19: The woman with hood and red hair is walking in the Black Forest, with the man with the red fedora following her behind. There is a castle in the distance on top of the hill.

To read more about Gord and Rothütle’s adventures in the Black Forest, check out the book “Black Forest Shadow: A Dark Fantasy Guide to Docker and Kubernetes Security”, available on [DockerSecurity.io](https://dockersecurity.io).

References

1. Sonatype.
[*Unnecessary Risk: The Persistence of Open Source Vulnerabilities.*](#)
Sonatype, 2025.
2. Phil Muncaster.
[*Log4Shell Downloaded 40 Million Times in 2025.*](#)
Infosecurity Magazine, December 2025.
3. Tom Abai.
[*NPM Supply Chain Attack: Sophisticated Multi-Chain Cryptocurrency Drainer Infiltrates Popular Packages.*](#)
Mend.io, September 2025.
4. Mohammad-Ali A'râbi.
[*The Largest NPM Supply Chain Attack Ever and How to Defend Against It.*](#)
Dev.to, September 2025.
5. CISA.
[*Widespread Supply Chain Compromise Impacting npm Ecosystem.*](#)
America’s Cybersecurity and Infrastructure Security Agency, September 2025.
6. Jonas Scholz.
[*React2Shell Critical Vulnerability \(CVE-2025-55182\).*](#)
Docker Meetup Freiburg, December 2025.
7. OPSWAT.
[*Shai-Hulud 2.0: How to Secure Your Software Supply Chain Against the Second Wave.*](#)
OPSWAT, December 2025.

› [Back to the Table of Contents](#) ‹

For your IT projects you don't need a know-it-all.

You need a **{BUDDY}**.



Richard Fichter
CEO @ XDEV



Outdated software? Rising maintenance costs? Security risks? We help you to make Java applications fit for the future - with a clear concept and at eye level. In addition to modern tools, we offer premium support for your **Java modernization**.

- **Not a know-it-all - a buddy:** We support your team with pragmatic methods without playing the wise guy.
- **Modernization with strategy:** agile methods & proof of concept for a secure update.
- **Robust solutions:** We rely on proven technologies and practical concepts - without unnecessary complexity.

Let us move your Java project forward together!

**BEST
{BUDDYS}**
IN CODE

trusted by

ATU

EMPORIO ARMANI

CONRAD

Fraport

Red Bull

Allianz



Arrange a free discovery call here!

www.xdev.software

XDEV