

JAVAPRO

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

From Coder To System Designer

19 **LAZY CONSTANTS IN JAVA 26:
DEFERRED IMMUTABILITY AND JVM TRUST**

52 **STOP WRITING YAML: HOW TO DEFINE, TEST,
AND DEPLOY YOUR CLOUD IN PURE JAVA**

64 **BEYOND REST: RETHINKING
APIS WITH GRAPHQL AND JAVA**

76 **VIRTUAL THREADS MEET AI: JAVA CONCURRENCY
IN THE AGE OF INTELLIGENT SYSTEMS**

82 **FROM SPAGHETTI TO HEXAGONS: A PRACTICAL
GUIDE TO CLEAN JAVA ARCHITECTURE**

116 **WILL AI REPLACE ME?
UNDERSTANDING OUR VALUE IN THE AI ERA**





SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

From Writing Code to Designing Systems

Software development has always evolved through shifts in abstraction. From object orientation to frameworks, from monoliths to microservices, from on-premise systems to cloud-native architectures — each step promised to reduce complexity while enabling us to build more powerful systems. Yet every abstraction we introduced caused new layers, demanding deeper understanding rather than less. What we are experiencing now is not just another step in that evolution. It is a structural shift.

Java is no longer just a language we use to write applications. It has become a platform that actively shapes how systems are designed, executed, and scaled. Modern garbage collectors have absorbed performance concerns, AOT techniques reshaped startup behavior, virtual threads redefine concurrency, and new APIs bring hardware capabilities closer to everyday development. Responsibilities that once lived in application code are steadily moving into the runtime.

At the same time, artificial intelligence accelerates another revolutionary transformation. Code generation is no longer the bottleneck. Boilerplate disappears, patterns are reproduced instantly, and entire workflows can be scaffolded in seconds. But this acceleration does not simplify software development. The challenge is no longer writing code. The challenge is understanding systems.

When Complexity Moves, Not Disappears

It is tempting to interpret these developments as simplification. Virtual threads make concurrency easier, buildpacks reduce deployment complexity, infrastructure becomes programmable, AI generates code on demand. But complexity has not vanished - it has shifted. When infrastructure becomes part of the language, we must reason about system boundaries rather than isolated components. When APIs become more dynamic, we must ensure that distributed systems remain coherent and predictable. When AI generates code, we must validate intent instead of syntax. Our effort moves from implementation to interpretation.

This JAVAPRO issue explores that shift from multiple perspectives. It examines how the JVM becomes a co-author, how AI reshapes concurrency and workflows, how modern Java encourages clearer models and stronger guarantees, and how platform capabilities increasingly replace handcrafted solutions.

Architecture as a Stability Mechanism

As the cost of producing code approaches zero, architecture becomes the primary mechanism of control. Without structure, accelerated development leads to accelerated chaos. Systems degrade not because they fail immediately, but because they become harder to understand and evolve. When both humans and AI contribute code, consistency must be enforced deliberately. Architectural patterns such as hexagonal architecture and domain-driven design are no longer optional refinements. They are stabilizing forces. They define boundaries, guide dependencies, and preserve long-term maintainability. They ensure that rapid iteration does not erode clarity. If the platform increases velocity, architecture must provide direction.

Understanding What Actually Runs

While Java raises the level of abstraction, it also reminds us that the underlying system still matters. Class loading, dependency conflicts, and runtime behavior do not disappear simply because they are hidden. When failures occur, they surface at the lowest levels, requiring us to bridge the gap between machine state and application logic. Debugging, observability, and a solid understanding of runtime mechanics remain essential skills. The more powerful our abstractions become, the more important it is to understand where they break — and how to recover when they do.

Systems Are Built by People

As technology evolves, so do teams. Roles become more fluid. Developers, product owners, and tools collaborate more closely. AI participates in implementation, but responsibility remains human. Software development has never been just about code. It is about communication, shared understanding, and ownership. As tools improve, these human factors become more critical. Clear structures reduce cognitive load, and good collaboration prevents fragmentation. Clarity and collaboration determine whether systems remain maintainable — or drift into complexity.

From Code to Understanding

What connects the articles in this issue is a common realization: We are no longer optimizing how we write code. We are learning how to design systems in an environment where code is abundant. Java's transformation into a platform, the rise of AI, and the renewed importance of architecture all point in the same direction: The future of software development will not be defined by how fast we produce code, but by how well we understand the systems we build — and how responsibly we evolve them over time.

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

Contents

JAVAPRO

JAVAPRO

Architect Your Own Experience: Creating Your Individual JCON 2026 Journey	9
Direct Access to Leading Java Experts	10
Premium AI Track: AI in Production Use	10
Hands-on Introduction: The Workshop Day	10
Interactive Sessions and Direct Participation	11
Networking and Community	11
Free Tickets and Participation	11
Dates at a Glance	11
Conclusion	11

Core Java

Atiq Amjad

The Foundations of Data-Oriented Programming in Java	12
Defining DOP vs. OOP: A Paradigm Shift	13
Principles of DOP	14
DOP in Java: Language Features and Concrete Examples	14
Best Practices	16
Common Pitfalls	17
Real-World Use Cases	17
Conclusion	18

Core Java

Babneet Singh

Lazy Constants in Java 26: Deferred Immutability and JVM Trust	19
The Real Problem Is Not Laziness, It Is Trust	20
From Stable Values in Java 25 to Lazy Constants in Java 26	20
Why the Rename Matters	21
Under the Hood: The Hidden Role of <code>@Stable</code>	21
Collections: Where the Second Preview Got Smarter	21
What Most Articles Miss	22
So What Are Lazy Constants Actually Good For?	23
How Lazy Constants Align with Java's Computation-Shifting Goals	23
The Real Takeaway	24
The Lazy Constants Cheat Sheet	24

Core Java

Babneet Singh

Final Is No Longer a Hint in Java 26	25
JEP 500 and the Performance Runway Behind <i>Final Means Final</i>	26
Why This Matters for Performance: The Optimization Runway JEP 500 Unlocks	26
What changes in Java 26	28
The New Mental Model	28
Why <code>--add-opens</code> Is No Longer Enough	29
What Will Trigger Warnings in Practice	29
A Migration Playbook That Actually Works	30
Observability: Two Routes, One Guaranteed	30
Closing Thought	30

Core Java

Damiana Nascimento

Java 26 in Practice: How the JVM Is Changing the Way We Write Code	31
From Classic Java to the JVM as a <i>Co-Author</i>	32

Concurrency: From <i>Managing Threads</i> to <i>Describing Work</i>	32
Memory and Performance: Less Micro-Optimization, More Clarity	33
Integrity and the Type System: Less Magic, More Explicit Contracts	34
Quick Guide: Everyday Decisions in Java 26	35
Conclusion: The JVM as a Co-Author of Your Code	36

Core Java

Ravali Yatham and Gireesh Punathil

Reverse Engineering Java: Backtracking Native Crashes	37
Why Are Crashes Complex?	38
Why a Custom Debugger Is Needed	38
Setup	38
Problem Determination	39
Where Do We Go from Here?	43
Summary	44

API and Frameworks

Anthony Dahanne

Paketo Buildpacks: Ready for Java 26 & More!	45
Paketo Buildpacks: Not the Only Buildpacks Around!	46
Paketo Buildpacks: What Do They Allow You to Do Today?	48
Paketo Buildpacks and Spring Boot: A Love Story	49
What About Java 26? And After?	50

API and Frameworks

Wladi Mitzel

Stop Writing YAML: How to Define, Test, and Deploy Your Cloud in Pure Java	52
How We Got Here	53
Infrastructure in Real Languages	54
Building a Real Application	55
Testing Infrastructure with JUnit	59
CI/CD: One Pipeline for Application and Infrastructure	60
Reusable Components as Shared Libraries	61
Working with Existing Infrastructure	61
Trade-Offs	62
Conclusion	63

API and Frameworks

Vipin Menon

Beyond REST: Rethinking APIs with GraphQL and Java	64
Rest and Beyond	65
Rethinking APIs for AI Agents: The GraphQL Advantage	65
CONCLUSION	67

API and Frameworks

Pascal Wilbrink

Solving Spring AI's UI Challenge with AG-UI's Java SDK	68
Introduction: The Missing Link in Java AI Development	69
What is AG-UI? And Why Does It Matter?	69
Spring AI's Strength (and Its Blind Spot)	69
The AG-UI Java SDK: Protocol Made Practical	70
A Real-World Integration: Setting Up AG-UI with Spring Boot	70
Connecting CopilotKit: The Frontend Side	71
Adding Tools: Making Agents Actionable	72
Shared State: Context That Travels	73
Why AG-UI Changes the Game for Java Developers	74

Open Source and Community-Driven	74
--	----

AI and ML Damiana Nascimento

Virtual Threads Meet AI: Java Concurrency in the Age of Intelligent Systems	76
Introduction: Java, AI, and the New Concurrency	77
AI Workloads in Production: What Changes for Java	77
From Traditional Threads to Virtual Threads in Java 26	77
Where Virtual Threads Shine in AI Systems	78
Reactive or Virtual Threads? Comparing Models	78
The Cost Model Shift: Blocking is No Longer Expensive	79
Architectural Impact: Rethinking Java Services for AI	80
Best Practices and Pitfalls with Virtual Threads	80
The Future of Java Concurrency for Intelligent Systems	80
Conclusion: Simpler Concurrency for AI	81

Architecture and Microservices Enrique Medina Montenegro

From Spaghetti to Hexagons: A Practical Guide to Clean Java Architecture	82
A Culinary Warning and the Dark Side of Code	83
Unveiling the Hexagon	84
Meet the Citizens (From Concept to Code)	84
The Invisible Pillars (Technical Concerns)	87
Compliance and Fulfillment	89
Testing Strategy	89
From Blueprint to Reality (Reference Application)	90
Conclusion: Hexagonal Horizons	92

Architecture and Microservices Marek Dominiak and Vadzim Prudnikau

Do You Still Write Code without Hexagon?	93
Why is it important to consider it?	94
What is Hexagonal Architecture?	94
Example	95
Hexagonal Architecture in the Microservices World	97
Should we use it everywhere?	97
Illusions of Hexagonal Architecture	98
An ‘Important’ Question: “Why Does It Have 6 Sides?”	99
How Is It Different from Clean and Onion Architectures?	99
Conclusion	100

DevOps Kamil Banach

From ‘Class Not Found’ to Calm: Practical Shading and Shadowing for Java Apps	102
The Class Loader and Classpath	103
Class Loader Hierarchy	103
Class Shadowing and Dependency Hell	104
Class Shading and Maven Shade Plugin	105
Practical Use Cases	106
Legal Considerations	107
Summary	107

IDE and Tools Stefan Richthofer

Meet APIdia: A New Approach to API Documentation Browsing	109
Motivation	110

Assessing the Broader Picture	111
Architectural Principles of APIdia	112
Conclusion	115

Human Factors

Florian Sommer

Will AI Replace Me? Understanding Our Value in the AI Era	116
The Discussion	117
Central Fallacies	118
Vibe Coding: Déjà Vu in the No-Code Trap	119
The Difference: LLMs Reach Below the Surface	119
Synthesis	120

Human Factors

Anshika Koul

When Product Owners Commit Code: Small Steps, Big Impact	122
The Pain We Didn't Know Was Pain	123
The Slightly Crazy Question	125
The Humble PR That Changed Everything	125
The Shift We Didn't Expect	126
The Impact You Can't Put on a Dashboard	127
Why It Worked	128
How We Lost It and What That Taught Us	129
Rebuilding, One Small Step at a Time	130
The Bigger Lesson	130

JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
Germany

E-Mail: info@javapro.io
Website: <http://www.javapro.io>

Editor in Chief:

Markus Kett (V.i.S.d.P.)

Editorial:

info@javapro.io

Design, Layout & Print:

Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
Germany

Copyright (c) 2025

Impuls Mediengruppe GmbH

All rights reserved.

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

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

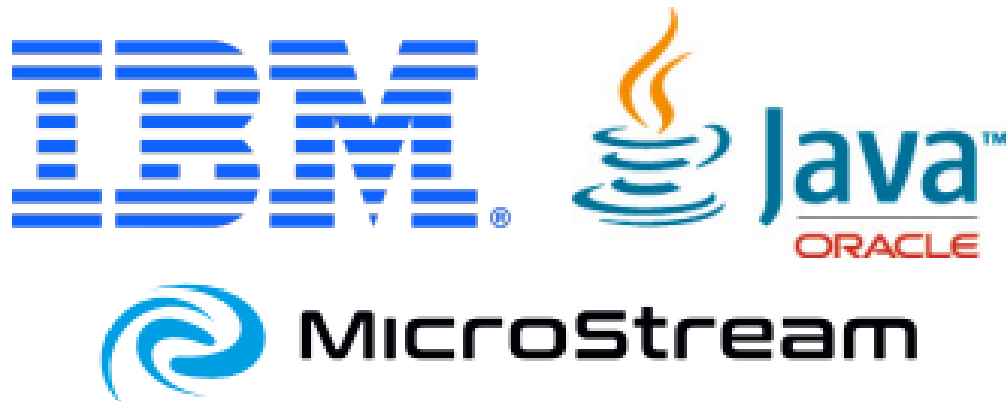
Articles marked with a name do not necessarily reflect the opinion of the editors.

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

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors





APR
20-23



**International
Java Community Conference**
#JCON2026 www.jcon.one

Architect Your Own Experience: Creating Your Individual JCON 2026 Journey

#JAVAPRO **#JCON** **#AI** **#Tickets**

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

The **JCON EUROPE 2026**, taking place from **April 20 to 23** at the **Cinedom Multiplex Cinema** in Cologne, consistently focuses on a format that offers participants maximum flexibility. Instead of a fixed schedule, the individual design of one's own conference experience takes center stage.

With a wide range of sessions, formats, and networking opportunities, visitors can decide for themselves which content, conversations, and contacts they want to focus on.

The central foundation for this is the **JCON Schedule**: participants can create their personal agenda, bookmark relevant **sessions**, and organize targeted **1:1 meetings** with speakers or exhibitors. This turns the conference into something that is not just consumed, but actively shaped—depending on individual interests, projects, and professional priorities.

Direct Access to Leading Java Experts

A central component of JCON is the direct exchange with international top speakers. In the **Mentorship Hub**, participants have the opportunity to speak in small groups with Java Champions such as **Bruno Souza**, as well as other experienced developers, architects, and open-source contributors.

These sessions offer a rare opportunity to discuss specific technical questions, challenge architectural decisions, or address personal career questions directly with leading figures of the Java community.

In addition, individual meetings with speakers can be arranged to deepen specific topics in a targeted manner.

Alongside this direct exchange, current technological developments in particular are brought into focus at the conference.

Premium AI Track: AI in Production Use

With the new [Premium AI Track*](#), JCON sets a clear focus on the practical use of artificial intelligence in Java systems.

On two conference days (April 21–22), deep-dive sessions are offered in a dedicated cinema auditorium, specifically aimed at teams that want to integrate AI into production applications.

The focus is on real-world challenges such as architecture, integration, and operation of AI systems—not theoretical concepts, but concrete solutions for use in enterprise environments.

At the core are, among others, the following sessions:

- **Local Development in the AI Era**_(Kevin Dubois, IBM)_
Shows how AI models can be run locally and integrated into development processes—with a focus on control, costs, data privacy, and reproducible setups.
- **Seven Habits of Highly Effective AI Java Coding**_(Jonathan Vila, Sonar)_
Best practices for the productive use of AI coding tools in Java projects—with a particular focus on code quality, maintainability, and responsible use.
- **Design Patterns for Multi-Agent Systems** (Brian Sam-Bodden, Redis)
Comparison of different architectural approaches for agent-based AI systems—including concrete patterns such as routing, planner–critic loops, and tool integration.

Hands-on Introduction: The Workshop Day

Even before the official start, the conference begins on April 20 with a dedicated [Workshop Day*](#).

Across a total of eight workshops, held in two time slots, participants work directly on concrete tasks and apply technologies in practice. The focus is clearly on hands-on experience.

Limited seats ensure a focused learning environment, allowing participants to deepen their understanding of the topics.

Key highlights include, in particular, workshops such as:

- **Caching and Beyond – Smarter Data Processing with Java** (Christian Kümmel)
Focus on performance, efficient data processing, and scalable architectures—with direct relevance to real enterprise applications.

- **Building Powerful GenAI Apps with Pure Java** (*Florian Habermann*)
A practice-oriented introduction to generative AI within the Java stack—without dependency on other programming languages.
- **Think AI is Only for Python? Think Again** (Don Bourne & Michal Brož, IBM)
The workshop shows how machine learning and AI workloads can be implemented productively within the Java ecosystem.
- **Modernizing Java Applications with AI Assisted Workflows** (*Bruno Borges & Sandra Ahlgrimm, Microsoft*)
Concrete strategies for modernizing existing systems—supported by AI-driven analysis and transformation tools.

Interactive Sessions and Direct Participation

During the sessions, participants can submit and prioritize questions via Slido. The most relevant questions are picked up and discussed live.

The JCON program is complemented by formats such as the Tessl & JAVAPRO Interview Booth, where **Baruch Sadogursky** conducts conversations with speakers on key conference topics as well as on technology, innovation, business, and culture.

At the same time, **Frank Delporte**(Foojay) speaks with participants on site and brings their perspectives and experiences into podcast formats.

Networking and Community

In addition to the sessions, JCON offers numerous opportunities for informal exchange. The **Happy Hour** on two evenings as well as the **VIP event** specifically create space for conversations with speakers, Java Champions, and other participants.

The **expo area** also focuses on exchange and discussion, complemented by **live demos** and direct conversations with technology providers and open-source projects.

Free Tickets and Participation

Participation in JCON EUROPE 2026 is **free of charge** for many members of the Java community, including JUG members, JAVAPRO readers, authors, and JCON alumni.

A **JCON 3 Day Full-Conference Pass*** can be [booked free of charge](#) using the code **JAVAPRO!**

For participants from commercial companies, the JCON 3-day full-conference pass* costs 699 EUR (plus VAT).

**Workshops and the Premium Track AI require an additional pass/upgrade.*

Dates at a Glance

- Workshop Day: April 20, 2026
- Main Conference: April 21–23, 2026
- Premium AI Track: April 21–22, 2026
- VIP Event: April 22, 2026
- Happy Hour: April 21–22, 2026

Conclusion

JCON EUROPE 2026 combines technical content with direct access to experts, hands-on formats, and diverse networking opportunities. Participants can actively shape the conference and use it in a targeted way for their own goals.

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



The Foundations of Data-Oriented Programming in Java

#CoreJava

#OOP

#DOP

Author: Atiq Amjad

He's a technology leader with a strong background in Java and software architecture. Atiq Amjad began his journey as a Java developer, and his passion for the Java ecosystem and open source has stayed with him throughout his career.



LinkedIn

Java has traditionally been an object-oriented language, urging developers to structure code around **objects**, encapsulate state and behavior, hide implementation details, and extend functionality through inheritance. This method has advanced modularity and reusability. However, with recent enhancements, Java has evolved, revolutionizing how systems are developed and offering powerful tools for developers.

Have you ever been in the midst of debugging a complex Java application, only to find that the mutable state across various objects left you tangled in a web of unexpected behaviors and racing conditions? These frustrations are common among Java developers accustomed to Object-Oriented Programming (OOP), where mutable objects often lead to more complex, less predictable systems. Fortunately, with recent language enhancements, Java developers now have capable tools to treat **data** as a first-class citizen.

This article reviews how records, sealed types, and pattern matching form the backbone of **Data-Oriented Programming (DOP)** in Java. This paradigm complements Object-Oriented Programming (OOP) by reducing complexity and building more stable systems.

Defining DOP vs. OOP: A Paradigm Shift

Object-Oriented Programming (OOP): OOP centers around the concept of “objects” that bundle data (attributes) and behavior (methods) together. The core principles include encapsulation, inheritance, and polymorphism. The goal is to model real-world entities in which objects communicate by sending messages to one another. State is often mutable, and behavior is tightly coupled with the data it operates on.

Data-Oriented Programming (DOP): DOP, by contrast, emphasizes the *data* itself. It advocates for separating data structures from the logic that operates on them. The data is typically immutable, simple, and transparent. Logic is expressed through functions or algorithms that take data as input, transform it, and produce new data as output.

Here’s a quick comparison:

DOP vs. OOP Cheat Sheet:

Feature	OOP	DOP
Core Unit	Objects (State + Behavior)	Data (Records/Value Objects)
State	Often Mutable (can change over time)	Immutable (created once, never changes)
Logic	Methods defined within classes	Functions/Algorithms separate from data
Polymorphism	Method overriding	Pattern matching
Focus	How objects behave and interact	What the data is and how it transforms
Transparency	Encapsulated (data often hidden)	Transparent (data structure is clear)
Good at	Boundaries, rich libraries	Data-heavy small services

Abb. 1: DOP vs. OOP Cheat Sheet:

Principles of DOP

DOP is not only about using new language features; it represents a mindset shift based on core principles that, when applied in Java, lead to even more robust and maintainable code.

Separate Data from Logic

This is the fundamental principle. The data structures should primarily describe “what the data is?”, without embedding complex business logic. The logic that operates on this data should live in separate functions or methods, taking data as input and producing results. As a rule of thumb, if changing a field would alter the business rules, it is best to handle it outside the record. This clear boundary helps maintain separation of concerns, ensuring that data representation remains independent of the logic that manipulates it.

Data is Immutable

Once a data structure (like a `record`) is created, its internal state should not change. This data immutability presents notable benefits:

- **Thread Safety:** No need for complex synchronization primitives, simplifying concurrent programming. For instance, a TechCo study found a 25% reduction in concurrency-related defects after refactoring the application to use immutable structures.
- **Predictability:** The *fixed* state of the data increases predictability and makes it much easier to reason about the program’s flow.
- **Testability:** Functions that operate on immutable data are *pure functions*, producing the same output for the same input, and have no side effects.

Data is Defined by Shapes (Algebraic Data Types)

- **Product Types:** Data structures that combine different types (e.g., User has an `id`, `name`, and `email`). Java’s Records are perfect for modeling product types.
- **Sum Types:** Data structures that represent one of several possible types (e.g., a `PaymentMethod` can be a `CreditCard`, `PayPal`, or `Crypto`). Java’s Sealed Types enable elegant modeling of sum types.
- This approach formalizes the data model, making it clear and comprehensive.

Data is Transparent

The internal structure of data should be easily accessible. While OOP often advocates for strict encapsulation and hiding data behind getters/setters (which sometimes leads to an *Anemic Domain Model* anti-pattern), DOP embraces the idea that data, particularly simple value-like data, benefits from transparency. Java `record` naturally exposes its components through accessors named after each component.

DOP in Java: Language Features and Concrete Examples

Records: Immutable Data Carriers

In Java’s past, developers often faced challenges with excessive boilerplate code, manual implementation of methods like `equals()` and `hashCode()`, and potential equality bugs when modeling simple data objects. These issues not only led to verbose code but also introduced room for human error. Records were introduced to address these specific problems, providing a more concise, intuitive approach to data modeling.

Records are a concise, immutable, and transparent way to model data aggregates. They are ideal for representing value objects or DTOs (Data Transfer Objects) when the primary need is to store data rather than to model complex behavior.

OOP-style:

```

1 // Before Records: Lots of boilerplate
2 public class User {
3     private final String id;
4     private final String email;
5     private final UserStatus status;
6
7     public User(String id, String email, UserStatus status) {
8         this.id = id;
9         this.email = email;
10        this.status = status;
11    }
12    // Accessors, equals(), hashCode(), toString() manually implemented...
13 }

```

DOP-style:

```

1 // With Records: Clean, concise, immutable
2 // Auto-generated equals(), hashCode(), toString(), and accessors
3 public record User(String id, String email, UserStatus status) {}
4
5 // UserStatus could be an enum or another record itself
6 public enum UserStatus { ACTIVE, INACTIVE, PENDING }

```

Key benefits of Records:

- **Immutability by Default:** All components are **final**.
- **Concise Syntax:** Generates the constructor, accessors (`id()`, `email()`, `status()`), `equals()`, `hashCode()`, and `toString()` automatically.
- **Transparent:** Clearly defines the data's shape.

Sealed Types: Controlled Hierarchies

Sealed types, whether interfaces or classes, allow developers to declare a type that can be implemented or extended only by a finite, known set of other types. This is essential for modeling algebraic sum types, where a value can take one of several distinct forms.

Imagine a `PaymentMethod`. It can be a `CreditCard`, a `PayPal` transaction or a `Crypto` payment. It cannot be anything else (e.g., `BankTransfer`).

```

1 // Define the sealed interface
2 public sealed interface PaymentMethod
3     permits CreditCard, PayPal, Crypto; // Only these types can implement it!
4
5 // Implementations as Records
6 public record CreditCard(String number, String expiryDate, String cvv) implements
7     PaymentMethod {}
8 public record PayPal(String emailAddress) implements PaymentMethod {}
9 public record Crypto(String walletAddress, String transactionId) implements PaymentMethod
10    {}
11
12 // Unknown implementation of PaymentMethod interface
13 public record BankTransfer(String iban)
14     implements PaymentMethod {} // Compiler will throw compilation error

```

Key benefits of Sealed Types:

- **Compile-Time Guarantees:** The compiler knows all possible direct subtypes and leverages them through pattern matching.
- **Controlled Hierarchies:** Prevents external, unknown implementations and ensures the data model is precise.
- **Enhanced Readability:** Clearly expresses the discrete options for a given type.

Pattern Matching: Logic Over Data

Pattern matching, especially with **switch** expressions and **record** patterns, is the glue that brings Records and Sealed Types to life in a DOP context. It allows developers to inspect the shape of data and execute logic based on that shape, replacing brittle **instanceof** chains and complex *visitor* patterns.

Pattern matching with **instanceof**:

```
1 // Example of pattern matching with 'instanceof'
2 public void processPayment(PaymentMethod method) {
3     if (method instanceof CreditCard card) {
4         // Type pattern: 'card' is a CreditCard here
5         System.out.println("Processing credit card: " + card.number());
6     } else if (method instanceof PayPal pp) {
7         System.out.println("Processing PayPal for: " + pp.emailAddress());
8     }
9     // ... and so on
10 }
```

Pattern matching with **switch** and **record**:

```
1 public String getPaymentSummary(PaymentMethod method) {
2     return switch (method) {
3         // Record Patterns: Deconstruct the record directly
4         case CreditCard(String number, String expiry, String cvv) ->
5             "Credit Card ending in " + number.substring(number.length() - 4)
6             + ", expires " + expiry;
7         case PayPal(String email) ->
8             "PayPal transaction for: " + email;
9         case Crypto(String wallet, String txnId) ->
10            "Crypto payment from wallet: " + wallet;
11         // The compiler ensures all sealed types are handled, or demands a 'default'
12     };
13 }
```

Key benefits of Pattern Matching:

- **Exhaustiveness Checks:** The compiler guarantees that developers handle all possible cases in a switch expression for sealed types, preventing runtime errors.
- **Deconstruction:** Extract components of records directly within the case label, reducing boilerplate.
- **Readability:** Expresses complex conditional logic much more cleanly than nested if-instanceof checks.

Best Practices

Prefer Immutability: Make your data records truly immutable. Avoid setters and exposing internal mutable collections directly.

Keep Records Simple: Records are best for simple data aggregates. Consider breaking down the deeply nested record into smaller, more focused records.

Model the data, not the behavior: A **User** record represents a user. A **PaymentMethod** record represents payment method details. Avoid verb-based names that suggest behavior.

Leverage Sealed Types: Use sealed interfaces and classes whenever you have a concept that can take on a fixed, known set of forms. This makes your domain model explicit and benefits from compiler-enforced exhaustiveness.

Don't Fear Small Objects: DOP naturally leads to more, smaller objects. This is typically a good thing, as smaller objects are easier to reason about, test, and garbage collect.

Validation at Construction: Use compact constructors to enforce invariants immediately. Invalid data should be impossible to create.

Value objects over primitives: Replace raw String/int with semantic types. This strengthens the model and reduces accidental mix-ups.

Common Pitfalls

While DOP delivers notable benefits, it is not a universal solution. Understanding its limitations and best practices assures effective application.

Pitfall	Problem	Solution
Anemic Records	Records with no behavior, logic scattered	Add derived methods, use pattern matching for operations
Nesting Record Definitions	Deeply nested inline record definitions reduce readability and reuse	Follow the Law of Demeter; no more than 3 levels of nesting
Sealing Unstable Domains	Sealing types that frequently change	Only seal when the domain is truly closed; use interfaces for evolving domains
Mutability Leaks	Embedding mutable collections or dates	Use <code>List.copyOf()</code> , <code>Instant</code> instead of <code>Date</code> , immutable wrappers
Over-Patterning	Using pattern matching for simple cases	Traditional polymorphism is fine when it reduces complexity
Premature Abstraction	Creating sealed hierarchies for simple two-variant types	Start with simple records; introduce sealing when you have 3+ variants or need exhaustive matching

Abb. 2: Common Pitfalls

Turning pitfalls into learning checkpoints can offer practical guardrails for developers. Consider embedding questions into your code review process to proactively address potential issues. Here are some questions that can guide your review:

1. Is the separation between data and logic being blurred anywhere in the code?
2. Are there any mutable fields within records or shared data structures that could lead to race conditions?
3. Have you ensured that all sealed types are fully covered in the switch expressions?
4. Are you inadvertently mixing different concerns, such as behavior and data representation, in records?
5. Is there any redundancy in data modeling that could be simplified using records or sealed types?

By consistently addressing these questions, teams can turn common pitfalls into opportunities for improvement and ensure that DOP is applied effectively and efficiently.

Real-World Use Cases

DOP in Java is highly effective in several practical scenarios, especially within modern system architectures.

Cloud-Native APIs and Microservices: Mapping incoming JSON or XML payloads for REST or gRPC services directly to records provides type safety, immutability, and clear data definitions, simplifying serialization and deserialization.

Event-Driven Systems: Events are inherently immutable, so modeling event types as records under sealed interfaces creates a clear schema, supports safer evolution, and enables cleaner consumers through pattern matching.

State Machines and Workflow Processing: Sealed interfaces are perfect for defining the discrete states of a workflow or domain entity, helping to prevent accidental misuse.

Cloud-Native Configuration: Java records are a perfect choice for modeling application configuration, ensuring immutability and easy access. Complex configurations with optional elements or multiple profiles can use Sealed Types for greater clarity.

Domain-Driven Design (DDD): Modeling domain entities as records and value objects provides greater control over data correctness and business rules.

Conclusion

Data-Oriented Programming does not replace object orientation; instead, it leverages Java's newer features to enhance data modeling, especially for small, data-centric services. DOP represents a natural evolution in Java. By using records, sealed types, and pattern matching, developers can build systems that are:

- More robust through compile-time exhaustiveness checking
- Easier to reason about with immutable data and pure functions
- Simpler to maintain with transparent data structures
- Better performing through immutability and value semantics

Start with Small Steps

Convert one DTO to a record, model a domain concept as a sealed hierarchy, or refactor a complex if-else chain using switch and pattern matching. The resulting clarity and safety will quickly demonstrate the value of this approach.

Adopting DOP does not require a complete rewrite, only a shift in perspective:

model data first, make states explicit, and let the compiler work for you.

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

APR
20-23



www.jcon.one

JCON2026

JCON EUROPE 2026

International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



Lazy Constants in Java 26: Deferred Immutability and JVM Trust

CoreJava

Java26

Author: Babneet Singh

Babneet Singh is a committer on the Eclipse OpenJ9 and OMR open-source projects, specializing in core JVM components and Java runtime internals.



LinkedIn

Java developers have long lived with an awkward trade-off. If you want the clarity and optimization benefits of **final**, you must initialize eagerly, in a constructor or static initializer. If you want to defer expensive setup until first use, you usually fall back to mutable lazy-initialization patterns, nullable fields, or double-checked locking. Those techniques work, but they also make code harder to reason about and harder for the JVM to optimize.

JEP 526, delivered as a preview in Java 26, introduces *Lazy Constants* to close that gap. The feature began life in Java 25 as *Stable Values*, and the rename is revealing. The first preview looked more like a low-level mechanism. The second preview is shaped much more like a developer-facing abstraction. JEP 526 explicitly says the Java 26 API is a revised second preview of JEP 502, renamed from *StableValue* to *LazyConstant*, simplified by removing lower-level setter-style methods, and extended with lazy collection factories on *List* and *Map*.

Most introductions stop there and explain how to replace a **null** check with `lazyConstant.of(...)`. That is useful, but it misses the more interesting story. Lazy Constants matter because they expose a form of *deferred immutability* that the JVM can still treat like a constant. The feature is about performance, yes, but not only startup performance. It is really about regaining *JVM trust* after moving initialization out of eager **final** fields. JEP 502 framed the original problem in exactly those terms: developers should not have to choose between flexible initialization and the peak performance associated with **final** fields and constant folding.

The Real Problem Is Not Laziness, It Is Trust

Java developers already know how to be lazy. They have been writing `if (field == null)` checks, holder classes, double-checked locking, and cache maps for years. The problem is not how to postpone initialization. The problem is how to postpone initialization *without giving up immutability semantics and optimizer trust*.

The JEP motivation starts with the limitations of **final**. Final fields are Java's main tool for expressing immutability, but they must be initialized eagerly: in constructors for instance fields and in class initialization for static fields. The declaration order also constrains initialization order. That is fine for trivial constants, but it is clumsy for real systems where *constant* values may still take meaningful time to construct, such as loggers, parsers, repositories, or precomputed structures.

The classic workaround looks familiar:

```
1 class OrderController {
2     private Logger logger = null;
3
4     Logger getLogger() {
5         if (logger == null) {
6             logger = Logger.create(OrderController.class);
7         }
8         return logger;
9     }
10 }
```

This pushes work later, but it creates new costs. Every access has to go through the getter. Concurrency becomes your responsibility. The invariant “*this field changes at most once*” exists only in the developer's head. More importantly, the JVM cannot trust a normal mutable field the way it trusts a true constant. That is the gap JEP 526 is trying to fill with *deferred immutability*: initialized no earlier than needed, at most once, and then effectively constant thereafter.

From Stable Values in Java 25 to Lazy Constants in Java 26

Lazy Constants are not the first version of this idea. Java 25 previewed the API as *Stable Values*. The original API was more flexible and more imperative. It exposed methods such as `orElseSet`, `setOrThrow`, and `trySet`, and it also offered stable suppliers and stable lists. It was powerful, but it felt like an API shaped around the underlying mechanism rather than the main user story.

Java 26 narrows the focus. JEP 526 renames the type to *LazyConstant*, removes the low-level setter-oriented methods, keeps construction-time factory methods that take a computing function, moves lazy list and map factories into *List* and *Map*, and disallows **null** as a computed value. The result is a smaller and clearer API centered on one common pattern: define the computation at declaration time, then call `get()` when you need the value.

That leads to a much simpler idiom:


```

1  class OrderController {
2      private final LazyConstant<Logger> logger =
3          LazyConstant.of(() -> Logger.create(OrderController.class));
4
5      void submitOrder(User user, List<Product> products) {
6          logger.get().info("order started");
7          // ...
8          logger.get().info("order submitted");
9      }
10 }

```

The `get()` call guarantees that the constant is initialized before it returns, and that under concurrent access only one thread performs a *successful* initialization. If initialization fails, a later `get()` may retry. That makes the feature feel much more like a natural Java API instead of a concurrency utility with extra rules.

Why the Rename Matters

The most important sentence in the JEP history is not about syntax, but about naming. The old name, `StableValue`, suited a low-level API closer to the mechanism underneath. The new name, `LazyConstant`, emphasizes the high-level use case: one constant value, initialized on demand.

That rename also reveals what the feature is not. It is not a general-purpose lazy container. It is not an all-purpose cache slot. It is not a future replacement for every custom initialization pattern. The design says, very clearly: this is for a single value that should be computed late, at most once successfully, and then treated as immutable. That narrower definition is exactly what gives the runtime room to optimize.

Under the Hood: The Hidden Role of `@Stable`

This is where the feature becomes much more interesting for JVM engineers and performance-minded developers.

Lazy Constants are implemented in the Java libraries, not as a new language feature and not as a new bytecode instruction. JEP 526 explicitly says it is not a goal to add syntax for lazy fields and not a goal to alter the semantics of `final`. Yet the runtime story is not merely *ordinary library code*; it relies on existing JVM support for trusted stable state. JEP 502 explains that the content of a stable value is stored in a non-final field annotated with the JDK-internal `@Stable` annotation, and JEP 526 says the original `StableValue` name was closer to that underlying JVM mechanism.

`@Stable` tells the JVM that, although a field is technically not `final`, it will not change after its one successful update. If a lazy constant is reachable through a trusted path rooted in `static final`, the JVM can treat its content like a true constant and apply constant-folding optimizations. That is the feature's real value proposition. It is not just laziness. It is laziness with *constant-like semantics*.

Before Lazy Constants, developers had to choose:

- use `final` and get optimizer trust, but initialize eagerly;
- use mutable lazy patterns and defer work, but give up optimizer trust.

Lazy Constants exist to recover both. They are Java's developer-facing wrapper around a JVM idea that JDK internals have benefited from for a while.

Collections: Where the Second Preview Got Smarter

A single constant is useful, but many real systems need pools, registries, or fixed sets of lazily created objects. Java 26 bakes that into the collections API through `List.ofLazy(...)` and `Map.ofLazy(...)`. A lazy list stores each element behind its own lazy constant and initializes elements independently on first access. A lazy map does the same for a known key set.

That is a small API design choice with a big usability payoff. In Java 25, the collection helpers lived under `StableValue`. In Java 26, they live where developers will look for them: `List` and `Map`. This is another sign that the second preview is intentionally moving away from mechanism-first design toward everyday discoverability.

What Most Articles Miss

By now, most public articles cover the happy path: startup wins, constant folding, API simplification, and the rename. The more interesting part for an experienced audience is where Lazy Constants are **NOT** a perfect fit.

Blocking Behavior Can Turn `get()` Into a Contention Point

The usual examples make `get()` look innocent. You call it, the value appears, then future accesses are cheap. That is true when initialization is fast and reliable.

If multiple threads race to initialize the same lazy constant, only one computing thread runs the supplier while the others block until initialization completes. That is correct and predictable. But thread interruption does **NOT** cancel initialization. `LazyConstant::get` does not clear interruption state and does not throw `InterruptedException`. The API documentation also makes clear that there is no timeout or cancellation support for callers waiting on initialization.

Because the API provides no timeout and no cancellation, Lazy Constants are a poor abstraction for anything whose initialization path involves flaky networks, remote discovery, external services, or unpredictable I/O. If the supplier is not truly bounded and reliable, `get()` turns from a constant accessor into a potential contention point.

Retry-After-Failure Can Repeat Side Effects

Another subtle point is the behavior on failure. If the computing function throws, the throwable is propagated, the lazy constant remains uninitialized, and a later `get()` may try again.

That is a sensible design for transient errors, but it has an uncomfortable implication. If your supplier writes a file, registers a resource, logs a metric, updates counters, or performs any partially visible work before failing, the next call may do it again. Many developers assume Lazy Constants are *initialized once*, but the precise truth is *initialized at most once successfully*.

Consider this dangerous pattern:

```
1 private final LazyConstant<Connection> db = LazyConstant.of(() -> {
2     Metrics.increment("db.connection.attempts"); // SIDE EFFECT
3     return Database.connectOrThrow();           // MIGHT FAIL
4 });
```

If `connectOrThrow()` fails, the `LazyConstant` remains uninitialized. The next time a thread calls `db.get()`, the supplier runs again, and your `db.connection.attempts` metric increments a second time.

Memory Retention Is the Hidden Startup Trade-Off

The startup story around Lazy Constants is attractive because it focuses on work you do not pay up front. What gets less attention is the work you cannot undo later.

Once a lazy constant is initialized, its content can never be removed. The lazy constant strongly references its content for as long as the lazy constant itself remains reachable. The computing function is also strongly referenced at least until initialization completes successfully. The Java 26 API documentation warns explicitly that this can become an unintended memory leak.

This is a crucial trade-off: Lazy Constants are not a cache eviction mechanism, not a soft reference, and not a general resource lifecycle tool. They are a permanent one-way transition: empty, then initialized forever. If the *constant* happens to be a large object graph, an oversized parser table, or a rarely used but heavyweight subsystem, you are trading startup cost for long-term retention.

The Optimization Story Is Narrower Than It Sounds

Writers love to say, *Store a `LazyConstant` in a **final** field and the JVM can optimize it like a constant*. That is directionally true, but the real rule is narrower. The Java 26 API docs say constant folding is only possible if there is a direct reference from

a **static final** field to a lazy constant, or if there is a chain from a **static final** field through one or more trusted fields.

That is not the same as *all final-field uses get the same win*.

Instance **final** fields are still weakened by reflective mutability, except in special cases, while **static final** fields remain much more reliably optimizable. JEP 502's risks section calls this out directly. In practice, this means the optimizer payoff is strongest in static singleton-style patterns and more conditional elsewhere. They are not a magic performance pass you sprinkle on every instance field.

Furthermore, they are not a universal replacement for lower-level patterns. For example, you can store an array inside a lazy constant, but doing so does not improve the access performance of the array's individual elements. For that, the new lazy collections, such as `List.ofLazy(...)`, are the better abstraction.

Class Initialization Order Still Matters

It is tempting to see Lazy Constants as a way to escape Java's static initialization headaches. They certainly help reduce eager work, and that alone can simplify startup profiles. But they do not suspend the rules of class initialization.

The Java 26 API documentation warns that use in static initializers can still interact with class initialization order, and cyclic initialization may still produce initialization errors. Delayed work is not the same thing as simpler dependency graphs. If your system already has class-init ordering problems, Lazy Constants may simply move the failure from startup to first use.

One More Practical Limit: Serialization

There is one more practical limitation worth noting. `LazyConstant` is not `Serializable`. That is not a problem for most internal runtime structures, but it does matter if you are tempted to embed lazy constants inside objects that participate in serialization-based frameworks, remote DTOs, or session replication.

So What Are Lazy Constants Actually Good For?

After all those warnings, it is worth being clear: Lazy Constants are still a very good idea.

They are excellent when you have *expensive, immutable, on-demand state* that should be initialized safely and only when first touched. Loggers, optional subsystems, parsers, repositories, pools with independently initialized entries, fixed registries, and component graphs that benefit from staggered warmup are all strong candidates. JEP 526 positions the feature precisely as a way to initialize application state incrementally, on demand, rather than monolithically, while guaranteeing at-most-once initialization in multi-threaded programs.

They are especially appealing in code where the old alternatives were ugly:

- the class-holder idiom only works for static cases and can become verbose;
- double-checked locking is error-prone and blocks constant folding;
- volatile array schemes with `VarHandle` are difficult to read and easy to get wrong;
- concurrent maps defer work but do not give the JVM the same trust guarantees.

Lazy Constants clean up that mess. They just do not make the trade-offs disappear.

How Lazy Constants Align with Java's Computation-Shifting Goals

Lazy Constants are not formally part of Project Leyden, but they align closely with Leyden's broader goal of improving startup, warmup, and footprint by shifting computation to a more useful point in a program's lifecycle.

Seen that way, `LazyConstant` gives developers a standard way to defer initialization until first use while still preserving the JVM's ability to treat the resulting value like a trusted constant when the usage pattern allows it. Unlike traditional lazy-init patterns that rely on mutable state and often weaken optimization, Lazy Constants aim to preserve both flexibility and optimizer trust.

The Real Takeaway

The most useful way to think about Lazy Constants is this:

- They are not Java finally adding *lazy fields*.
- They are not just a prettier `Supplier`.
- They are not a universal replacement for every deferred initialization technique.

They are Java 26's current answer to a much narrower and much more interesting question: *How do we expose deferred immutability to user code without giving up JVM trust?*

That is why Lazy Constants matter. And that is also why the interesting conversation is not “How do I replace a null check?” but “Which initialization problems deserve constant-like semantics, and which ones do not?”

That is the part most articles miss.

The Lazy Constants Cheat Sheet

- **Best fit:** `static final` fields, singletons, parsers, optional subsystems, fixed registries, heavy object graphs that should initialize on first use.
- **Optimization rule:** The JVM optimizes best when the lazy constant is reachable from a `static final` root. Instance-field cases are more conditional.
- **Concurrency:** `get()` blocks competing threads during computation. No timeout, no cancellation, no `InterruptedException`.
- **Failure mode:** *At most once successfully*. If the supplier throws, the next `get()` may try again. Avoid side effects before failure.
- **Memory:** No eviction. Once initialized, the value is held strongly for as long as the `LazyConstant` is reachable.
- **Bad fit:** Flaky I/O, remote service discovery, heavy initialization that must be cancelable, values that need eviction, and lifecycle-driven initialization where the supplier is not known at construction time.

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



Final Is No Longer a Hint in Java 26

CoreJava

Java26

Author: Babneet Singh

Babneet Singh is a committer on the Eclipse OpenJ9 and OMR open-source projects, specializing in core JVM components and Java runtime internals.



LinkedIn

Java 26 (JEP 500) warns when code mutates final fields via deep reflection and introduces explicit controls to scope or deny that behavior. This article breaks down what changes, why it matters for JIT optimizations, and a migration loop you can put into CI.

JEP 500 and the Performance Runway Behind *Final Means Final*

For years, Java has carried a quiet contradiction: the language treats final fields as immutable after initialization, but deep reflection could still mutate them. That single escape hatch forces the JVM to be conservative about both correctness and optimization, because a final value might not actually be final.

Java 26 (JEP 500) starts closing that gap in a pragmatic way: *mutating final fields via deep reflection still works for compatibility*, but the runtime issues warnings by default and introduces an explicit, scoping opt-in for the code that truly needs to keep doing it.

Bottom line: final-field mutation is no longer *just reflection*. It is a capability that must be explicitly enabled.

Key Takeaways

- JEP 500 targets deep reflection that mutates final fields. In practice most hits are instance fields, but the policy is about final fields in general.
- You cannot avoid warnings with `--add-opens` alone anymore. Openness and mutation permission are separate gates.
- Two controls matter: `--enable-final-field-mutation` (who is allowed) and `--illegal-final-field-mutation` (what happens when it is illegal).
- In Java 26, `warn` is the default and it warns at most once per caller module. Use `debug` to build a real inventory.
- Records already set expectations for strong immutability; JEP 500 moves ordinary classes closer to that model.

The Trust Boundary Shift

ASPECT	JDK25 AND BEFORE (PAST)	JDK26 AND AFTER (FUTURE)
Final fields	Mutable via deep reflection	Illegal mutation denied by default (future release)
Runtime assumption	Final fields might change	Final fields are intended to be treated as immutable by default once deny becomes the default
Impact	Weaker integrity, conservative optimizations	Safer behavior, potentially faster code

Before JEP 500, the JVM had to treat final fields as possibly mutable somewhere, because APIs existed that could rewrite them. JEP 500 is the platform putting immutability back on a path to being a trustworthy contract, aligned with *integrity by default*.

Why This Matters for Performance: The Optimization Runway JEP 500 Unlocks

JEP 500 is a compatibility change today, but it creates future space for the JVM to treat final as a stronger optimization and integrity boundary.

More Reliable Constant Folding (and the Chain Reaction After It)

The JEP calls out a concrete example: when the JVM can trust final fields are never reassigned, it becomes possible to apply *constant folding* (evaluate once, reuse many times). It also notes that constant folding is often the first step in a chain of optimizations that can deliver significant speedups.

In practical JIT terms, that chain often includes:

- propagating folded values into branches (enabling dead-code elimination),
- hoisting stable loads out of loops,
- removing redundant loads and checks once a value is proven stable.

Different JVMs implement these differently, but the constraint is shared: if *final might change*, the optimizer must stay conservative.

Deeply Immutable Object Graphs Become Meaningful Again

The JEP is blunt: as long as APIs exist that can mutate finals at will, you cannot rely on final fields to build deeply immutable graphs of objects, and the JVM cannot use those graphs to deliver its best performance optimizations.

Once the platform shifts to deny-by-default and most applications do not enable mutation, final becomes a stronger building block for:

- stable configuration objects,
- immutable value-like aggregates,
- safely shared state with fewer defensive patterns.

Less Reliance on Speculative Optimization and Deoptimization

JEP 500 even discusses an alternative approach: the runtime could speculate that finals are not mutated and deoptimize if it detects mutation. The JEP rejects that as a primary strategy, because the mutation itself is undesirable and because recording mutation needs explicitly is better for the ecosystem.

That matters operationally: fewer speculative assumptions typically means fewer *surprise* deopts and more predictable long-running performance.

A Concrete Example: Unmodifiable Collections and Deserialization

JEP 500 gives a specific performance-shaped example: if deserialization paths can avoid mutating final fields (for example by delegating to constructors or limited-purpose APIs), the JVM could trust the final fields of JDK unmodifiable list implementations even when they were created via third-party deserialization.

That is the pattern to watch: the less final-field mutation exists in real programs, the more aggressively runtimes can treat widely-used objects as stable and optimize around that stability.

Important reality check: Java 26 is still a transition release. The biggest optimization headroom lands when the ecosystem runs clean under [deny](#) and, later, when deny becomes the default.

What changes in Java 26

The two switches that matter

SWITCH	WHAT IT CONTROLS	JAVA 26 DEFAULT AND INTENT
<code>--enable-final-field-mutation=...</code>	Grants a module the capability to mutate final fields via deep reflection	Permanent opt-in, intended to be selective
<code>--illegal-final-field-mutation=allow warn debug deny</code>	Policy for illegal final-field mutation attempts	Default is <code>warn</code> , designed as migration scaffolding

`--enable-final-field-mutation` is a capability, not a workaround

You can enable final-field mutation for:

- all classpath code: `--enable-final-field-mutation=ALL-UNNAMED`
- specific named modules: `--enable-final-field-mutation=M1,M2`

Enabling mutation for a module does not guarantee mutation will succeed. The target package must still be open to the caller (for example via `--add-opens`).

`--illegal-final-field-mutation` is explicitly transitional

The modes are:

- `allow`: permits mutation without warning
- `warn`: permits mutation but warns the first time per module (at most one warning per caller module)
- `debug`: like `warn`, but includes a stack trace for every illegal attempt
- `deny`: throws `IllegalAccessException` for illegal attempts

The roadmap is also explicit: `warn` is the Java 26 default and will be phased out and removed; `deny` becomes the default in a future release; when that happens, `allow` is removed while `warn` and `debug` remain for at least one release.

The New Mental Model

`Field::setAccessible` behavior is unchanged, but `Field::set` is tightened for final fields. Mutating a final field via deep reflection is legal (and warning-free) only if all conditions hold; otherwise it's *illegal* and the policy controls whether it proceeds or fails.

```

1 Attempt Field::set on a final field (after calling setAccessible)
2     |
3     v
4 (1) Did setAccessible(true) succeed?
5     |
6     +--- no --> InaccessibleObjectException (module access) /
7                 other access failure
8     |
9     v
10 (2) Is the declaring package open to the caller module
11     in a way that permits final-field mutation?
12     |
13     +--- no --> Illegal final-field mutation (policy applies)
14     |

```

```

15         v
16 (3) Is final-field mutation enabled for the caller module?
17     |
18     +-- no --> Illegal final-field mutation (policy applies)
19     |
20     v
21 Apply --illegal-final-field-mutation policy:
22     |
23     +-- allow/warn/debug --> mutation proceeds
24                             (warn/debug emits diagnostics)
25     +-- deny                --> IllegalAccessException

```

This is why `--add-opens` alone is no longer a satisfying answer. You now need to know whether the caller has the mutation capability.

Why `--add-opens` Is No Longer Enough

JEP 500 is very explicit: in Java 26, it will not be possible to avoid warnings simply by opening packages.

A particularly useful corner case is what I call the *Field laundering* scenario: module B legally obtains an accessible Field, then passes it to module C. Even if C has mutation enabled, the write can still be illegal if the package is not open to C. The legality is evaluated at the call site under the caller's module identity.

```

1 Module A opens pkg.p -> Module B
2 Module B gets Field f and calls setAccessible(true)
3 Module B passes f to Module C
4
5 Module C has final-field mutation enabled
6 but pkg.p is not open to Module C
7
8 Result: 'Cs f.set(...) is illegal (policy applies)

```

JEP 500 also notes that dynamic opens at runtime (for example `Module::addOpens`, `ModuleLayer.Controller::addOpens`, or `Instrumentation::redefineModule`) do not retroactively grant the *permission to mutate finals* capability to a third module.

What Will Trigger Warnings in Practice

JEP 500 calls out a reality you should plan around: code that mutates final fields via deep reflection is usually *library code*, not application code.

Common sources:

- dependency injection frameworks or legacy injection patterns that write into fields post-construction
- test and mocking frameworks that patch object state for convenience
- older instrumentation and cloning patterns that rewrite object invariants

Serialization: The Non-Goal That Prevents Unnecessary Panic

JEP 500 explicitly says it is not a goal to prevent mutation of final fields by serialization libraries during deserialization, and it describes how JDK serialization can deserialize objects with final fields.

For third-party serialization libraries, the JEP points toward a limited-purpose approach (for example `sun.reflect.ReflectionFactory` for `Serializable` classes) rather than relying on unconstrained deep reflection indefinitely.

Practical takeaway:

- standard Java serialization is not meant to break,
- the warning surface is typically ad hoc reflective field writes outside of legitimate deserialization needs.

A Migration Playbook That Actually Works

This is the opinionated part: treat `warn` like a deprecation notice, not like noise. The whole point is to get you to a world where `deny` is normal.

Step 1: See the First Signal (default warn)

Java 26 will `warn` by default when illegal mutation occurs, and that warning is once per caller module.

Step 2: Inventory Everything (debug)

Use `debug` to identify every call site.

Step 3: Fix, Then Scope the Escape Hatch (Only If Essential)

- **Prefer:** upgrade or refactor the offending library or pattern.
- **Bridge (temporary):** enable only the minimal modules that still need it using: `--enable-final-field-mutation=<module list>`.

Step 4: Gate Your Future (deny in CI)

Once clean, run CI with: `--illegal-final-field-mutation=deny`.

This is the fastest way to prevent regressions and be ready for a future deny-by-default release.

Observability: Two Routes, One Guaranteed

- Guaranteed baseline: start with `--illegal-final-field-mutation=debug`.
- Optional accelerator: if JFR is enabled, the JVM records `jdk.FinalFieldMutation` events for final instance field mutation and for `Lookup.unreflectSetter` usage that obtains a mutating handle.

Closing Thought

JEP 500 is the platform correcting a past trade-off: final should be a reliable integrity boundary, not a polite suggestion. Java 26 gives you the migration window. Inventory with `debug`, burn down the offenders, move CI to `deny`, and you will be ready when the default tightens and the JVM can finally optimize under trustworthy immutability.

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



SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES



www.javapro.io
www.javapro.io



Java 26 in Practice: How the JVM Is Changing the Way We Write Code

CoreJava

Java26

VirtualThreads

Author: Damiana Nascimento

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



LinkedIn

Java 26 in practice shows a fundamental shift in how modern Java systems are built. Virtual threads, modern GC, AOT caching, and the Vector API are shifting long-standing responsibilities from the application into the JVM — forcing developers to rethink everyday design decisions.

Java 26 in practice changes how we think about new Java features. When we think about new Java, we usually look first at syntax. Pattern matching, records, new JDK APIs grab attention. With Java 26 in practice, however, the most profound impact is different. It is not in language constructs.

The real change is in the virtual machine that runs our code. The JVM has taken on responsibilities that were traditionally handled by libraries. These include fine-grained thread management. Also memory optimization, startup warm-up, and even deep reflection tricks.

This shift fundamentally changes how we design systems. Java 26 in practice raises two pragmatic questions. How do virtual threads affect everyday decisions? How do JVM-level enhancements and vectorized computation reshape software design? What should we adopt? What should we avoid? Rather than listing features, we explore real impact. This new JVM contract influences concurrency, architecture, performance, and security.

From Classic Java to the JVM as a Co-Author

For many years, the default mental model for Java was simple yet painful. We managed a small number of native threads. Thread pools needed careful sizing. Blocking operations were feared. **Java 26 in practice** changes this completely. Web servers no longer cap requests by pool size. Business logic fits naturally within limits. Timeouts and queues become less critical.

This setup shaped everyday decisions. Blocking expensive threads? Avoid it. Push code into callbacks and reactive flows. Even naturally synchronous business logic suffered. GC pauses demanded constant attention. We fought object allocation wars. Ad-hoc caches and mutable reuse became standard. The type system? Deep reflection filled the gaps.

Reflection even mutated final fields. The ecosystem now sees this as dangerous. **Java 26 in practice** explicitly flags it with warnings. Java 26 crystallizes a fundamental shift. The JVM moves beyond neutral runtime. It becomes a co-author of our systems [InfoWorld JDK 26 coverage].

Virtual threads absorb scaling costs. No more complex async code. Ahead-of-Time caching (Project Leyden) tackles startup. G1/ZGC improvements reduce manual memory work.

Stricter reflection warnings help. The Vector API matures steadily. Java 26 draws clear boundaries around safe, performant code.

Classic questions change answers:

“Can I block here?” → Yes.

“Do I need this pool?” → No.

“Write this in C?” → Vector API.

“More reflection?” → Warning.

Java 26 in practice repositions decisions between app code and JVM. We explore this through two lenses. How do these changes affect design? What to adopt or avoid?

Concurrency: From *Managing Threads* to *Describing Work*

The most immediate impact of the modern JVM is felt in concurrency, where virtual threads and structured concurrency fundamentally change everyday decisions about I/O handling and parallel orchestration. In the past, the guiding principle was *do not block expensive native threads*, which pushed teams toward reactive or asynchronous architectures even for workflows that were naturally synchronous — such as a REST service calling a database and an external API.

In Java 26, the JVM takes on the cost of scaling virtual threads — thousands or even millions of them — over a small number of native carrier threads. This allows developers to write blocking, readable code without fear of exhausting resources, exactly as defined by the official [JEP444 specification](#), now widely adopted as the foundation for structured concurrency in Java 26.

What to Adopt: *One Request = One Thread* as the Default

Adopt virtual threads as the default choice for any blocking I/O point: web controllers, HTTP clients, JDBC drivers, and calls to legacy services. For parallel orchestration, use structured concurrency, which provides a *try-with-resources-like* semantic for groups of tasks, with cascading cancellation and explicit timeouts.

Consider this classic example: an endpoint that fetches user data and orders in parallel.

Before (Java 17, manual futures):

```
1 Future<String> userFuture = pool.submit(() -> fetchUser(id));
2 Future<String> ordersFuture = pool.submit(() -> fetchOrders(id));
3 String user = userFuture.get(2, TimeUnit.SECONDS);
4 String orders = ordersFuture.get(2, TimeUnit.SECONDS);
```

After (Java 26, structured concurrency + virtual threads):

```
1 try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
2     var userFuture = scope.fork(() -> fetchUser(id)); // vthread
3     var ordersFuture = scope.fork(() -> fetchOrders(id)); // vthread
4     scope.joinUntil(Instant.now().plusSeconds(2));
5     return merge(userFuture.get(), ordersFuture.get());
6 }
```

The *before* version requires an external thread pool, manual cancellation management, and carries the risk of thread leaks. The *after* version is synchronous and readable, leaving multiplexing and cleanup to the JVM. In practice, this means that decisions such as *reactive* or *synchronous*? are no longer imposed by technical constraints, but instead become deliberate architectural choices.

What to Avoid: Oversized Pools and Callback Hell

Avoid fixed or cached thread pools for web requests — they artificially cap concurrency and make tuning more complex than necessary. Steer clear of cascading callbacks or long `CompletableFuture` chains when the flow is sequential or involves simple fan-out; on a modern JVM, these patterns are unnecessary and often counterproductive.

For libraries that still *pin* virtual threads (by blocking carrier threads), isolate them on dedicated platform threads. This should be the exception, not the rule.

Impact on Modern Design

This shift reshapes modern development. Web services are increasingly designed with a *blocking-first* mindset, using structured scopes for points of complex orchestration. Frameworks such as Spring Boot and Quarkus already integrate virtual threads natively, simplifying server configuration.

The result is domain code that stays closer to business intent (*do this in parallel, cancel on timeout*) and far less concurrency infrastructure scattered throughout the application. At scale, this enables thousands of concurrent requests with the same memory and CPU footprint, changing cloud sizing calculations and influencing decisions around microservices.

Memory and Performance: Less Micro-Optimization, More Clarity

While virtual threads are transforming concurrency, continuous improvements in garbage collection (G1 and ZGC) — now consolidated in Java 26 — are reshaping how we approach memory and startup, influencing everyday decisions around allocation, scalability, and application profiles. In the past, Java developers waged a constant battle against the GC: aggressive heap tuning, object pools to *reuse everything*, premature ad-hoc caches, and even choosing alternative languages for CLIs or serverless workloads due to slow cold starts.

In Java 26, the JVM absorbs much of this cost, making room for more direct and readable designs.

What to Adopt: Modern GC as the Default, Clarity Over Frugality

Adopt G1 as the default garbage collector — now optimized with cheaper write barriers and reduced synchronization between application threads and the GC — or choose ZGC for latency-critical scenarios. For startup-sensitive applications, experiment with Leyden’s *Ahead-of-Time* (AOT) object caching, which pre-materializes hot data and code into reusable artifacts, independently of the GC. Early reports indicate significant cold-start improvements, often on the order of tens of percentage points, according to benchmarks reported by [InfoQ](#).

In practice, this enables bolder design decisions: create immutable objects and functional collections without allocation paranoia; use lightweight DTOs and composition freely, letting the JVM handle throughput. For profiles such as serverless functions or command-line tools, Leyden makes Java competitive with *native* languages — without requiring you to rewrite logic just to warm up the JVM.

What to Avoid: Hand-Tuned GC and Object Pools

Avoid exotic heap configurations and young/old generation tuning that were common in Java 8–17; modern G1 and ZGC deliver predictable throughput and pause times out of the box. Steer clear of manual object pools or reusable mutable structures built solely to *save allocations* — the cost of creating and discarding objects has dropped significantly. For short-lived workloads, stop avoiding Java because of warm-up concerns; Leyden addresses this at the JVM level, without requiring specialized application code.

Impact on Modern Design

These changes reshape modern development by pushing memory optimizations into the runtime, enabling architectures that are more focused on the domain. High-throughput services can see gains of 5–15% simply by moving to an optimized G1 configuration, according to public benchmarks and reports from the OpenJDK community on recent G1 improvements, without any changes to business code.

Microservices and batch jobs increasingly use Java in scenarios where GraalVM or other languages were once the obvious choice due to faster startup. Everyday design shifts toward *clear models first, optimizations later* — with many of those optimizations now handled by the JVM. As a result, operational complexity is reduced (less tuning, less GC monitoring), freeing teams to prioritize business logic over performance infrastructure.

Integrity and the Type System: Less Magic, More Explicit Contracts

Beyond concurrency and memory, Java 26 also signals a clear shift in how we approach object integrity and decision logic, with mature pattern matching and increasing restrictions on deep reflection — changes that directly affect the design of entities, APIs, and frameworks. Historically, bypassing the type system through reflection (including mutating **final** fields) was common in dependency injection, dynamic proxies, and ORMs, but it resulted in fragile code that was difficult to optimize. Now, the JVM enforces clearer boundaries, pushing the ecosystem toward a style that is more closely aligned with its type system.

What to Adopt: Pattern Matching and APIs with Explicit Contracts

Adopt pattern matching in **instanceof** and **switch** for decision logic and flow validation—now with support for primitive types and greater expressiveness, reducing **if/else** chains with manual casts. For vectorized computation (Vector API, JEP 529), apply it to numeric hot spots identified through profiling, writing loops that the JVM can map to SIMD (Single Instruction, Multiple Data) without leaving Java, as specified in the official JEP 529.

A common day-to-day example is processing heterogeneous data (such as parsing JSON with mixed types), where pattern matching replaces visitors and long **instanceof** chains with declarative switches, with exhaustiveness guarantees enforced by the compiler.


```

1 // Before: instanceof chains
2 if (obj instanceof String s) { ... }
3 else if (obj instanceof Integer i) { ... }
4
5 // After: pattern switch
6 return switch (obj) {
7     case String s -> processString(s);
8     case Integer i -> processInt(i);
9     default -> throw new IllegalArgumentException();
10 };

```

With the Vector API, scoring routines or array-processing workloads can achieve native-level performance without JNI. For vectorization, avoid native intrinsics or C/JNI-based micro-kernels; the Vector API delivers comparable performance in portable Java code.

Impact on Modern Design

These evolutions reshape modern development by making the JVM's type system both stricter and more expressive. Domain entities and DTOs increasingly become immutable by default, with explicit constructors replacing *magic injection*. Library APIs grow more predictable and easier to optimize because they adhere closely to JVM rules.

Vectorized computation in pure Java opens the door to analytics and lightweight ML engines for teams without native-code expertise, integrating naturally into the JVM ecosystem. In day-to-day work, decisions such as *should I use reflection for this proxy?* or *should I write this in C to get SIMD?* increasingly become *does the JVM provide an API for this?* — simplifying stacks and reducing operational complexity.

Quick Guide: Everyday Decisions in Java 26

Java 26 in practice means rethinking concurrency, memory, and startup from JVM-first principles.

To turn theory into practice, here is a concise checklist of what to adopt, what to avoid, and how these changes are reflected in the everyday decisions you make today.

Adopt:

- **Virtual threads as the default:** Use `Executors.newVirtualThreadPerTaskExecutor()` for web servers, HTTP clients, and blocking I/O. One request equals one thread, without artificial limits.
- **Structured concurrency for orchestration:** Use `StructuredTaskScope` with `ShutdownOnFailure` for parallel fan-out, timeouts, and automatic cancellation.
- **Modern GC without heavy tuning:** Prefer G1 or ZGC as defaults; let the JVM handle throughput and pause behavior.
- **AOT caching (Leyden) for cold starts:** For CLIs, serverless workloads, and short-lived microservices — achieve faster startup without changing business code.
- **Pattern matching and the Vector API:** Declarative switches for decision logic; vectorized loops for numeric hot spots, without JNI.
- **Immutable entities:** Records, `final` fields, and explicit constructors — prepare for a JVM that enforces stronger integrity.

Avoid:

- **Fixed or cached thread pools for web workloads:** They artificially limit concurrency; virtual threads scale better and more simply.
- **Callback hell and long `CompletableFuture` chains:** Unnecessary for synchronous flows or simple fan-out scenarios.
- **Micro-optimizing allocations:** Object pools and premature mutable reuse — modern GC makes this counterproductive.
- **Deep reflection and mutating `final` fields:** Warnings in Java 26 signal future issues; prefer explicit contracts.
- **JNI for SIMD or numeric micro-kernels:** The Vector API delivers portable performance in pure Java.
- **Avoiding Java due to slow startup:** Leyden addresses this at the JVM level, not in application design.

Java 26 changes the daily development flow: you model intent (*run this in parallel with a timeout*) instead of infrastructure (pools, manual backpressure). Designs become *synchronous by default*, with optimizations (GC, AOT, vectorization) handled by the JVM. Teams focus on domain logic and business rules, with less operational tuning and more homogeneous stacks — from microservices to analytics, all in scalable and readable Java.

Conclusion: The JVM as a Co-Author of Your Code

Virtual threads, JVM enhancements, and vectorized computation are not just features of Java 26 — they represent a new contract between developers and the virtual machine. The JVM has taken on responsibilities that once fell entirely on application code: scaling concurrency without complicating logic, optimizing memory without manual tuning, accelerating startup without architectural hacks, and enforcing integrity without leaving room for unchecked reflection.

Everyday decisions change accordingly: *can I block her* becomes *yes*; *do I need a custom pool?* becomes *no*; *should I write this in C for performance?* becomes the Vector API; *__can I mutate this_* **final** *field?* becomes an immediate warning. Software design naturally flows toward synchronous and readable code by default, with structured scopes for complexity and immutability for safety. Modern development is reshaped around a simple principle: less infrastructure, more domain. Teams express business intent (*run X in parallel with a timeout*) while the JVM handles scale, GC, and SIMD. From serverless to analytics, Java 26 turns *simple code that scales* from an architectural promise into an everyday reality.

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

JCON2026
www.jcon.one

JAVAPRO



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Reverse Engineering Java: Backtracking Native Crashes

CoreJava

Bytecode

Heapdump

JVM

Authors: Ravali Yatham and Gireesh Punathil

Ravali Yatham is a Java L3 Support Engineer with Runtimes team, in IBM India Software Labs. She is skilled in debugging complex production issues from large enterprise-grade deployments. She also worked on Java on Cloud and Java serviceability improvements. She contributed to the highly successful open source projects like Node.js and openj9.



LinkedIn

Gireesh Punathil is an IBM master inventor, Senior Technical Staff Member, committer or steering committee member for several leading open source projects. He is part of IBM Application Runtimes team that develops and supports websphere, java, eclipse and many other tools and frameworks. He spent a good share of his career in developing, porting and debugging platform tooling software: web servers, virtual machines, compilers, ide, monitoring agents and cloud components.



LinkedIn

“what abstraction conceals is exposed at the fault site.”

Java applications run on an abstracted, memory safe platform. But when JVM crashes, things collapse at the bottom native layer, and the root cause analysis becomes very hard. This is because the abstractions that lie between the actual crash context and the program context are too wide and deep, and the knowledge about the pathways between these abstractions are neither well documented nor well known to the developers.

This article walks through debugging a Java crash from the ground up: starting with raw native instructions and moving through JVM internals, calling conventions, object layouts, and the runtime’s translation of high-level semantics into low-level behavior. This guide equips engineers with the foundational skills to backtrack a failure, interpret crash artifacts with confidence and isolate root causes independently.

Why Are Crashes Complex?

Java, as a platform-independent language, and the JVM, as a platform-independent execution engine, deliberately hide the underlying platform details. This abstraction is a great feature and works perfectly fine at the develop, build and execution phases. However when the JVM crashes, the failure surfaces in the native layer in a raw and context-less manner, and completely detached from the abstracted semantics Java developers are familiar with.

To understand such failures, we must peel back those layers of abstraction multiple times. That means we need to reconstruct the layers between native instructions, JVM internals, the Java bytecode and the original Java code. Only by de-abstracting each layer we can meaningfully connect the fault site to the actual fault source.

For example a crash that manifests like below:

```
0x7ff2345bc  mov rdx, [rax, 0xa0]
```

would mean nothing in the Java source. This needs to be de-abstracted to for example:

```
x = obj.y;
```

form, for a developer to relate to, and make some reasoning about.

Why a Custom Debugger Is Needed

General-purpose debuggers like **gdb**, **lldb**, **dbx** and **windbg** only see JVM as a native (C++) application and show artifacts (stack trace, register context, raw memory etc.) from the point of view of native state. However, a JVM crash encompasses both: the native state where the real crash occurred, and the managed runtime and application state whose semantics led to that native state. Both the states are critically important in problem determination. No single off-the-shelf tool understands this combined picture.

A custom debugger translates states in both these layers: it helps to decode JVM frame layouts, interpret metadata, reconstruct Java stacks from native frames, understand calling conventions generated by the JIT and map raw addresses back to bytecodes, methods or objects. The custom debugger makes the JVM’s internal structures legible, correlates native artifacts with Java-level behavior and offers the cross-layer visibility required to backtrack a crash. Without these capabilities, a normal debugger shows only meaningless addresses and apparently corrupted frames, leaving the real root cause invisible.

Setup

While the methods and principles discussed here apply across JVM implementations, the diagnostic tooling, command syntax, internal VM structures and generated machine code shown are specific to the [Semeru JVM](#) and are used for illustration.

Environment Setup: Ensure that the JVM is running on a supported operating system and hardware architecture. Refer to the official [IBM documentation for supported platforms](#).

Diagnostic configuration setup: For the JVM to generate meaningful diagnostic data during a failure, specific diagnostic settings must be configured at OS level to avoid truncated coredumps before starting the application.

[Mustgather for Runtimes Java Technology](#) document provides detailed guidance on must-gather setup/gather with respect to platforms and problem scenarios.

Tools: Unlike other production anomalies, crash debugging usually requires usage of one or more tools depending on the nature of the crash, the crashing context and the execution environment (platform, architecture etc.) Below listed are the most commonly used tools for crash debugging.

Platform	Debugger
linux	gdb
windows	windbg
mac	lldb / gdb
aix	dbx
zos	dbx

Custom (runtime aware) debugger: For OpenJ9 system dumps, the primary tool is jdmpview that is bundled within the JDK. jdmpview can interpret JVM internal structures, display Java and native stacks, reconstruct mixed call stacks (native + interpreted + JIT), inspect heap objects and VM metadata. More details can be found in the [jdmpview documentation](#).

Diagnostic data: when your Java application crashes, there are a number of diagnostic data files generated by default: java-core, snaptrace, jitdump and system(core) dump. The system dump (core file) is the most critical artifact in the crash context. It contains a snapshot of the entire process address space at the time of the crash, including registers, native stack, Java stack, heap memory and JVM internal structures.

Problem Determination

Launch

Alright, let us start our debugging journey! First, launch the core file on to the debugger.

```
jdmpview -core <corefile>
```

Extract Crash Context (jdmpview)

Next, let us start by examining the crash context. This includes information on the failing module, failing thread, and its entire register context including the failing instruction, and the signal which the processor raised at the time of the crash.


```

> !gpinfo
Failing Thread: !j9vmthread 0x950800
Failing Thread ID: 0x2113bf (2167743)

J9Generic_Signal_Number=00000018
Signal_Number=0000000b
Error_Value=00000000
Signal_Code=00000001
Handler1=00007F4063E19710
Handler2=00007F40655D8410

RDI=0000000000950800 RSI=0000000000000007
RAX=0000000000000004 RBX=00007F40551584A0
RCX=00007F4065291360 RDX=00007F405515BA10
R8=00007F3FB10CA450 R9=00007F3FB035CF80
R10=0000000000000000 R11=000000000000000D
R12=00007F405C0FC818 R13=00007F405C03E5F0
R14=00007F406569A440 R15=0000000000000000
RIP=00007F4062618FA6 RSP=00007F40551583D0

Module=/home/javauser/java/java.8.0.8.45/
jre/lib/amd64/compressedrefs/libj9vmti29.so
Module_base_address=00007F4062610000

```

Extract Failing Thread (jdumpview)

Clearly, this is the lowest level context with little or no clues about the application context. So let us de-abstract the thread and map it into the application context:

```

jdumpview > info thread 2167743
process id: 2167471
thread id: 2167743
associated Java thread:
name: remote-data-transport-thread
Thread object: java/lang/Thread @ 0x7005ec298
Native info: !j9vmthread 0x950800 !stack 0x950800
Daemon: true
Thread.State: RUNNABLE
JVMTI state: ALIVE RUNNABLE

```

So the failing thread is “`remote-data-transport-thread`”. Though this information alone is not going to be enough, we have a foot hold in the wild!

“method of walking through unclear terrains:

step back. breathe and assess.

hold your ground. gather the data.

then look for a wedge. a narrow opening.
 open it up. cut through it. widen it.
 turn the wedge into a corridor.
 move through it. make progress.
 gather new insights in the process. repeat.
 make new pathways. progress again!
 that is how we create order from chaos.”

Extract Java Stack (jdumpview)

Now that we know the failing thread, let us see what is its “Java” state. This is well known as the Java stack.

```
> !stack 0x00950800
!j9method 0x628DE0 org/transporter/remote/Remote.xport0(Ljava/lang/String;)V
!j9method 0x628DE0 org/transporter/remote/Remote.xport0(Ljava/lang/String;)V
!j9method 0x628B60 org/transporter/remote/Remote.xport(Ljava/nio/file/Path;)V
!j9method 0x60A078 org/transporter/remote/Remote.xport(Ljava/nio/file/Path;)V
!j9method 0x615AD8 org/transporter/remote/Agent.run(bool)V
!j9method 0x614270 org/transporter/remote/Agent$Worker.run()V
!j9method 0x0635D0 java/lang/Thread.run()V
```

Extract Native Stack (gdb)

Now that we have a strong sense of “where” the crash is happening through de-abstracting from the native to the Java level, lets now repeat the same for “what” the crash is, how did it manifest, what it means in the immediate context and what it means in the application context.

```
>where
#0 0x7f40650856d5 in ?? ()
#1 0x7f406569a440 in newPath () from /home/javauser/java/java.8.0.8.45/jre/lib/amd64/compressedrefs/libjvm.so
#2 0x7f4055155a4d in ?? ()
#3 0x7f4055155a10 in ?? ()
#4 0x7f4065610382 in omdump_create (portLibrary=0x7f406569a440, filename=0xb...) at ../../omr/port/unix/omrodump.c:188
#5 0x7f40654f528f in doSystemDump (agent=0x7f405c05cb20, label=0x7f4055155a10 "/home/javauser",...) at dmpagent.c:763
#6 0x7f40654f0f75 in protectedDumpFunction (portLibrary=0x7f406569a440, userData=0x7f4055155970) at dmpagent.c:2899
#7 0x7f40655d9099 in omrsg_protect (portLibrary=0x7f406569a440, fn=0x7f40654f0f60...) at ../../omr/port/unix/omrsignal.c:425
#8 0x7f40654f4712 in runDumpFunction (agent=<optimized out>, label=0x7f4055155a10 "/home/javauser",...) at dmpagent.c:2877
#9 0x7f40654f489d in runDumpAgent (vm=0x7f405c03e5f0, agent=0x7f405c05cb20, detail=0x7f4055156050, ...) at dmpagent.c:2796
#10 0x7f406550b17d in triggerDumpAgents (vm=0x7f405c03e5f0, self=0x950800, eventFlags=8192, ...) at trigger.c:1052
#11 0x7f4063e19312 in generateDiagnosticFiles (portLibrary=0x7f406569a440, userData=0x7f4055156530) at gphandle.c:1162
#12 0x7f40655d9099 in omrsg_protect (portLibrary=0x7f406569a440, fn=0x7f4063e191b0, ...) at ../../omr/port/unix/omrsignal.c:425
#13 0x7f4063e1949a in vmSignalHandler (portLibrary=0x7f406569a440, gpType=24, ...) at gphandle.c:837
#14 0x7f40655d8648 in mainSynchSignalHandler (signal=11, sigInfo=0x7f4055157830, ...) at ../../omr/port/unix/omrsignal.c:1066
#15 0x7f4062618fa6 in jvmtiGetClassSignature (env=0x7f405c0fc818, klass=<optimized out>, ...) at jvmtiClass.c:343
#16 0x7f40563b0525 in Xport::resolve0(MethodInfo*, _jmethodID*, bool) () from /home/javauser/lib/libxport.so
#17 0x7f40563b1c73 in Xport::resolve(ASGCT_CallFrame*) () from /home/javauser/lib/libxport.so
#18 0x7f4056394d85 in Java_org_transporter_remote_Remote_xport0 () from /home/javauser/lib/libxport.so
```

Java being managed runtime, the faults are contained and preprocessed for several dump functions to capture various fault contexts. So frame 0 to frame 14 belong to the JVM that handles the dump processing. This means the actual failure occurred in frame 15, **jvmtiGetClassSignature** method of JVM’s JVMTI module. If we look beneath that, we see it is triggered from the native function **Java_org_transporter_remote_Remote_xport0**, which needed methods and classes to be resolved as part of the transport function.

A rough theory we can make at this point is that some bad input was passed to the JVMTI function from one of its callers. But no evidence, only a hypothesis! Let us prove or disprove it.

Extract Failing Instruction (gdb)

Next, get the failing instruction.

```
(gdb) frame 15
#17 0x7f4062618fa6 in jvmtiGetClassSignature at jvmtiClass.c:343
(gdb) info registers rip
rip 0x7f4062618fa6 <jvmtiGetClassSignature+198>
```

Then, dump the signal context (from the gpinfo above):

```
J9Generic_Signal_Number= 00000018
Signal_Number= 0000000b
Error_Value= 00000000
Signal_Code= 00000001
```

The most important information here is the signal number and the signal code.

Signal number: 0xb (11) is the infamous SIGSEGV (segmentation violation)

Signal code: 0x1 (1) means the reason for SIGSEGV: SEGV_MAPERR (address not mapped to object)

Now, dump few instructions in and around that address to get a little more context. Which instruction caused this? What it was trying to do? Which address is not mapped?

```
0x7f4062618fa6 <+198>: mov r10,QWORD PTR [rbp+rax*1+0x4]
```

This means load the value stored at memory address %rbp + %rax + 4 into r10.

Clearly, the source location is holding the bad memory address.

Lets compute that manually:

```
(gdb) info registers rax rbp
rax 0x4
rbp 0xffffffffffffffff
```

So clearly, rbp (-1) is pointing to a bad memory location.

So the immediate root cause of the crash is that rbp holds invalid content.

Backtrack (gdb)

How did rbp get that value? Let us backtrack a bit.

```
0x7f4062618f85 <+165>: test rbp,rbp
// check if rbp is NULL
0x7f4062618f88 <+168>: je 0x7f4062619130 <jvmtiGetClassSignature+592>
// jump for null case handling
0x7f4062618f8e <+174>: mov rbp,QWORD PTR [rbp+0x0]
// de-reference rbp into itself
0x7f4062618f92 <+178>: test rbp,rbp
// again check for NULL
0x7f4062618f95 <+181>: je 0x7f4062619130 <jvmtiGetClassSignature+592>
// handle null case
0x7f4062618f9b <+187>: mov rax,QWORD PTR [rdi+0x8]
// prepare for the call: probably some args
0x7f4062618f9f <+191>: mov rax,QWORD PTR [rax+0x930]
// prepare for the call: probably some args
=> 0x7f4062618fa6 <+198>: mov r10,QWORD PTR [rbp+rax*1+0x4]
// r10 = *(rbp + rax + 4) crash, bad rbp or rax
```

Wait a minute. What is this rbp, and where it has come from? Let us trace it back till the beginning of the method.

```
(gdb) x/7i 0x7f4062618ee0
```

```
0x7f4062618ee0 <+0>: push r15
0x7f4062618ee2 <+2>: push r14
0x7f4062618ee4 <+4>: push r13
0x7f4062618ee6 <+6>: push r12
0x7f4062618ee8 <+8>: mov r12,rdi
0x7f4062618eeb <+11>: push rbp
0x7f4062618eec <+12>: mov rbp,rsi
```

At this point, we also want to learn a little bit about the x86-64 ABI summary to understand the transition between functions.

The first six arguments are passed in registers rdi, rsi, rdx, rcx, r8, and r9 in that order, with the return value in rax.

- Registers are divided into two types: caller saved (volatile) and callee saved (non-volatile).
- Caller saved registers (rax, rcx, rdx, rsi, rdi, r8, r9, r10 and r11) can be freely mutated by the callee.
- Callee saved registers (rbx, rbp, r12, r13, r14 and r15) must be preserved by the callee before writing.
- rbp was traditionally used as a base pointer, but it is not required as per the ABI.
- The stack grows downward, the tip pointed to by rsp. local variables are typically allocated by subtracting from rsp.

With that lets look at the preamble of our method:

```
push r15
push r14
push r13
push r12
// as per above spec, these registers are callee saved,
// so the function must preserve it if it plans to use it.
mov r12, rdi
// first argument goes into r12
push rbp
// callee saved rbp is saved before use.
mov rbp, rsi
// second argument goes into rbp.
```

So the badness in rbp is originated in rsi. and that is our second parameter. What is the second parameter? Lets look at the code:

```
jvmtiError JNICALL
jvmtiGetClassSignature(jvmtiEnv* env,
    jclass klass,
    char** signature_ptr,
    char** generic_ptr)
```

Root cause: Xport::resolve0() passed a corrupted kclass pointer to jvmtiGetClassSignature().

The function expects kclass to reference a valid J9Object (the class object). Instead, the value was 0xffffffffffffffff, which is not a valid heap address. Subsequent dereferencing of this invalid pointer caused the failure.

Where Do We Go from Here?

We could repeat the same descent: step into Xport::resolve0() and trace how the corrupted klass value was produced and propagated. The methodology would not change. It is the same disciplined backtracking: validate inputs and follow data flow.

We stop here, because the purpose of this article is not to fully resolve this one problem, but to illustrate the discipline and tooling required to de-abstract a JVM crash and trace it to its true source. While the problem will change from story to story, the method does not!

Summary

Java promises memory safety, portability and abstraction from the underlying platform. But when it fails, the crash surfaces at the lowest native layer: with raw instructions, registers, memory addresses and faulting pointers. The semantics we are comfortable with (objects, methods, classes) are nowhere in sight.

In this article, we demonstrated the process of reconstructing the bridge between the original Java intent and the low level fault site. We began at the bottom, at the exact native instruction that failed. From there, we interpreted control flow and data flow with the help of the application binary interface, explained JVM internals where necessary, reducing abstraction at each step, bringing more and more clarity.

Once we understand how arguments flow through registers, how objects are represented in memory and how the runtime actually operates, a crash dump stops being scary, and becomes a rich source of information and signals. It reveals rich insights that no trace tool can fully capture!

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



Paketo Buildpacks: Ready for Java 26 & More!

API and Frameworks

Builpack

DevOps

SpringBoot

Author: Anthony Dahanne

Software Developer for 15+ years, Anthony Dahanne's favorite topics are containerization (Docker and Kubernetes), building tools, Continuous Integration and, of course, core Java development. Having recently joined HeroDevs, he works on patching and releasing EOL OSS Java and Spring projects. He's also a maintainer of the Paketo Java buildpacks. In his spare time, he works on various open source projects : from Mastodon bots written in NodeJS or Go, to Android apps!



LinkedIn

Paketo Buildpacks, the default way to create container images with Spring Boot, will include support for Java 26 the same week it will be released in major OpenJDK distributions (Oracle, BellSoft Liberica, Azul Zulu, etc.) Let's take this as an opportunity to (re)discover what Paketo Buildpacks are and what they allow you to do!

Paketo Buildpacks: Not the Only Buildpacks Around!

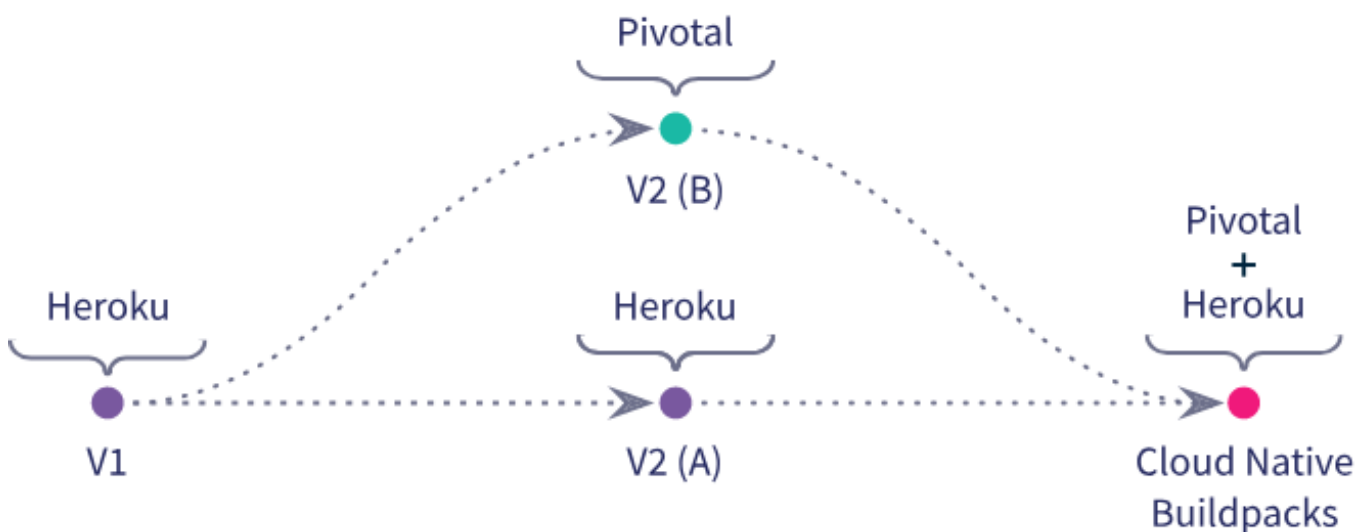
A World of Buildpacks

My first memory of using buildpacks dates back to when I worked for a bank around 2020 (yes, that year!) and my team created and maintained a bunch of shell scripts that would copy a templated Dockerfile, add the public certificates from the corporate Certification Authority (CA), and then build the Dockerfile into an image (and even create a Helm chart!); but we soon learned that there were open-source buildpacks that supported most of our use cases!

To be fair, I had already been using buildpacks when I tested Heroku in 2012, but I just did not know it: they were hidden by the platform (well, if you pay attention to the logs, you can see them in action). Then a few years later I heard (during a KubeCon) about Heroku engineers (now Salesforce engineers) talking about a new open specification to define the lifecycle and tooling around buildpacks to build OCI (Docker) images.

And finally, when I landed at VMware at the beginning of 2023, I joined a team of Cloud Foundry (yes, the self-hosted PaaS running on VMs built with BOSH and its Java Buildpack written in ... Ruby!) and Paketo buildpacks maintainers and their closed-source Tanzu equivalents.

Everything then clicked in my head: the CNCF buildpack (Cloud Native Buildpacks, CNBs) project is about the specification, lifecycle and the `pack` CLI; while Paketo Buildpacks are an implementation (Heroku and Google Cloud maintain different implementations) of this specification (detect and then build) for many programming languages: Go, PHP, .NET, Apache HTTPd, many others, and of course Java.



Cloud Foundry (sometimes referred to as v2) and Paketo Buildpacks (sometimes referred to as v3) are very different though: if we take the example of the Java buildpack:

- in Cloud Foundry it is only made of a single Ruby-based buildpack for all use cases (Spring Boot customization, APM integration injecting agents, etc.) and it never builds the jar/war nor does it create an OCI image but rather a process that runs on a Diego Cell (an isolated container running on a VM) and is exposed to a gateway
- in Paketo Buildpacks, it is a collection (often called a composite buildpack) of single purpose Go-written buildpacks (a buildpack to get a JDK, another to use maven, gradle, APM integrations, etc.) that eventually creates an OCI image

I also need to mention another family of buildpacks: the Nixpacks – well I recently discovered they're being deprecated and replaced with railpacks – they use BuildKit under the hood and support Java (but they do not support as many integrations as Paketo)

Where and How Are Buildpacks Used?

Buildpacks (any kind of buildpacks actually) have always been used in a context of Platform as a Service (PaaS) – Heroku, Cloud Foundry, Tanzu as I laid out previously.

But you can see usage of buildpacks in other PaaS such as:

- Railway – they use Nixpacks
- Paketo Buildpacks are used (or at least are supported via CNBs specification) almost everywhere else!
 - Fly.io, Porter
 - AWS App Runner, Google Cloud, Microsoft Azure Container Apps, IBM Cloud Code Engine, DigitalOcean App Platform
 - Epinio, the self-hosted Kubernetes PaaS

It makes sense: with all those platforms, the promise is to let the developer care about their code, not about the deployment; the platform will use the buildpacks to detect what is needed, and how to build and bring the required components

Let's not forget the end users (application architecture teams usually) who roll out their own internal PaaS! I have seen many engineers working for big organizations like banks or insurance companies use Paketo Buildpacks to make their build and deployment standard across teams and programming languages.

“But I’m Fine with the Good Old Dockerfile” You Say

I understand that statement! The Dockerfile is well understood today, so why would you go and learn another higher-level construct?

In the last chapter, I already started hinting at some advantages of the buildpacks over the Dockerfile in a big organization context:

- with Paketo Buildpacks, you can provide all your teams with an internal PaaS (integrate them to your Continuous Integration and when they release a tag the teams get their OCI image published) – no more time spent writing or maintaining a Dockerfile
- with Paketo Buildpacks, you control how images are built: no more proliferation of Dockerfiles across projects: every team will build the same way, and you can even make sure some concerns like CA public certificates, APM agents (Datadog, AppDynamics) and SBOMs are integrated exactly how you want for the whole organization
- with Paketo Buildpacks you have the best support available for Spring Boot app deployment: including integration with Kubernetes Service Binding, and performance customization (CDS, AOT Cache, AOT, unpacked format – more on that in the Spring Boot specific chapter below) – without trying to figure out how a training run is supposed to happen!

Benefit from the ecosystem for common use cases : custom CA certificate



Maintaining a Dockerfile means making sure the base OS, its dependencies, and the JVM are on the latest supported version, to avoid building with a CVE-ridden base.

So you're now wondering, “how can the Paketo maintainers make sure I always build with the latest and greatest versions (latest JVM, latest APM agent, etc.)?”

Well, we rely on a lot of automation: we have a [pipeline-builder](#) that generates GitHub Actions workflows that execute every week to download and checksum the latest versions, so that the updated buildpack will refer to them. We could have used Renovate (actually there are some discussions about it) or maybe even Dependabot, but I believe our automation predates those tools!

Also, Paketo Buildpacks do not include those dependencies: they are [online](#) buildpacks in the sense that if, for example, the detection notices you need Java 21 from Zulu, it will download it (most often from GitHub, verifying the checksum of course). That makes the component buildpacks lighter since they don't have to include several 200MB+ JDKs!

Paketo Buildpacks: What Do They Allow You to Do Today?

The Java Composite Buildpack

Let's focus on the Java buildpack, which includes more than 25 component buildpacks! Here are the highlights of the most popular ones:

- Apache Tomcat, TomEE and Open Liberty buildpacks: provide you with the servlet container of your choice if you set an environment variable or you need a servlet container (in case you just built a [war](#) archive)
- The APM (Application Performance monitoring) buildpacks will not only download the latest version of the wanted agent, but will also configure your app and startup script (which we often call [helper](#)) to configure and use them: Azure Application Insights, Datadog, Google Stackdriver (now renamed to Google Cloud Operations)

But there are others available, not included in the Java composite, that you can use too: Elastic, AppDynamics, New Relic, etc. – by simply specifying their buildpack name when you use the [pack](#) CLI or the Spring Boot plugins.

Those are great integrations, but maybe all you need is just the OpenTelemetry buildpack that provides you with the agent that can export to the OpenTelemetry Collector or directly to any OTLP/gRPC/HTTP capable backend; to stop relying on a given vendor agent!

- The JVM buildpack: there is only one in the composite, and it's BellSoft Liberica

But Azul Zulu, Oracle JDK, GraalVM JDK, Eclipse Adoptium Temurin, Eclipse OpenJ9, Amazon Corretto, SapMachine and Alibaba Dragonwell buildpacks are available simply by specifying their names – more on them in the Java 26 chapter!

- The packagers buildpacks: to allow you to simply provide source code to the buildpacks: Maven and Gradle of course, but also SBT, Leiningen and Clojure tools for the Scala and Clojure developers!
- The utilities buildpacks: [ca-certificates](#) to allow you to include a custom CA Certificate in the JVM truststore or keystore (at build time or even at runtime); the [syft](#) buildpack to run syft against each and every layer (like the JVM layer, app layer, etc.) to provide you with a detailed Software Bill of Materials (SBOM) of the image you built.

I think you get the idea: there's a buildpack for each need – I'd be surprised if you would ever run into a situation where there is no buildpack for your use case!

The Java-Native-Image Composite Buildpack

There is a second Java composite buildpack, similar to the previous, but focused on the creation of GraalVM-based apps, that will create a native binary of your Java application and set the entrypoint accordingly, thanks to the [native-image](#) buildpack

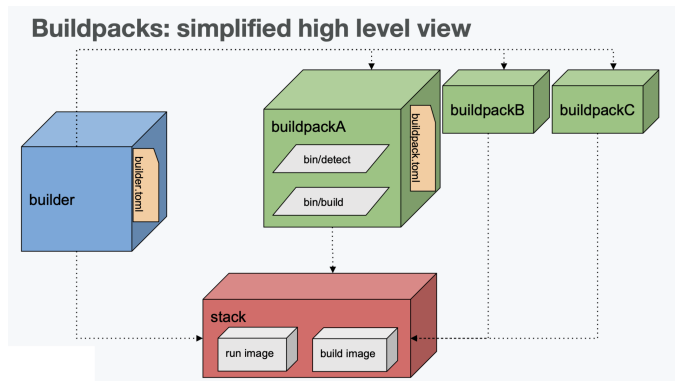
So the JVM and Integrations Can Be Chosen; What About the Base Image?

For the base images, Paketo Buildpacks, by default, rely on either Ubuntu (Bionic, Jammy and Noble) or Redhat (UBI 8, UBI 9 and soon UBI 10) base images. So without creating a new builder yourself, you can opt out of the box trust Paketo.

Talking about Ubuntu based images, did you know the [apt](#) buildpack, to install some system libraries in your image, is now an official Paketo Buildpack? Try it out!

But it's not that hard to use any OS of your choice as a base image.

You can use the `--run-image` argument when you build your image, but in that case you would have to make sure that your app will be compatible with a whole different image than the one that was used during build.



I would rather advise you to create your own builder (a builder defines a list of buildpacks, and the base images at build-time and runtime), it's not that hard, [I have an example based on Chainguard Wolfi distroless](#) (those base images that contain the absolute minimum OS libraries to run your app; perfect to reduce the attack surface)

Paketo Buildpacks and Spring Boot: A Love Story

Spring Boot started integrating (via the maven and gradle plugins to build OCI images) with Paketo Buildpacks since [Spring Boot 2.3.0.M1](#), back in January 2020, approximately at the same time that Pivotal was acquired by VMware.

Even a dedicated `spring-boot` buildpack was created (I see you Quarkus fans! don't feel abandoned by this article, there is a Quarkus extension to build using buildpacks, [a nice article from one of the maintainers](#) explains it all!).

This buildpack implemented several ideas:

- injecting the `spring-cloud-bindings` library, so that Kubernetes Service Binding (an operator injecting environment variables and secrets to your container) can help your app autoconfigure credentials and its data source
- Spring Generations: to warn the user if they're relying on an unsupported version of Spring Boot (we have GitHub actions workflows that retrieve the latest support date for all Spring Boot versions)
- configure less memory / threads for reactive apps; help with Spring Boot native image configuration, etc.

But let's talk about the latest and most exciting feature: Spring Boot performance.

Spring Boot Performance – Paketo Buildpacks Got You Covered!

What do I mean by Spring Boot performance with Paketo Buildpacks?

- GraalVM native builds? that has already been implemented along with the `native-image` buildpack, for quite some time now (since Spring Boot 3, November 2022)
- [CRaC](#) (Coordinated Restore at Checkpoint) builds? well, I have to admit there's no support for CRaC with Paketo Buildpacks – the issue is that with CRaC when you take a snapshot of the app, you could potentially bake confidential information, like data source credentials, directly into the OCI image... not saying it's impossible to avoid this, but it was never implemented with buildpacks as far as I know (another limitation of CRaC is that it only works on Linux, but that does not impact Paketo Buildpacks that are only for Linux based images anyway)

No, I mean AOT (Ahead-of-Time) Processing, CDS (Class Data Sharing – now transitioned into AOT Cache since Java 24) and... the simple yet effective unpacked layout.

To benefit from AOT Processing, you have to build your app with `process-aot` (via the maven or gradle plugin) and then you just tell the buildpacks to include the `-Dspring.aot.enabled=true` JVM argument at runtime (there's a shortcut environment variable for that, `BP_SPRING_AOT_ENABLED`). You may already know that AOT processing is a required step before you can turn your app into a native binary with GraalVM.

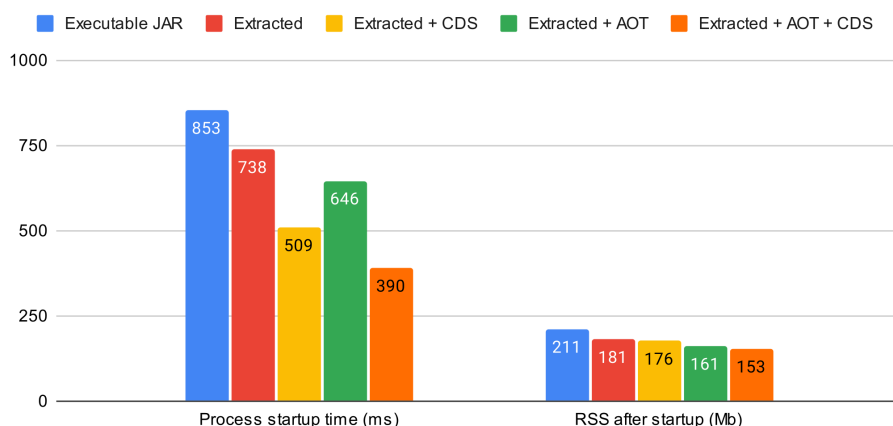
Unpack is just turning your Spring Boot app into an unzipped form, which no longer requires the Spring Boot class loader and is necessary for AOT Cache. The command to do that, since Spring Boot 3.3 is: `java -Djarmode=tools -jar my-app.jar extract --destination my-app-extracted`

AOT Cache (previously CDS) was more complicated to integrate in the buildpack: it requires a training run (you start your app until all beans are loaded and then you shut it down) to generate a caching file that will make your app start significantly faster. The difficulty is that the caching file must be generated in the exact same conditions (same JVM, same permissions, same timestamp even!) to be loaded for the next runs!

Those performance enhancements are real, [you can see several benchmarks](#) and see the startup time gets 4 times faster and the app memory consumption was divided by 4!

Have a look at this [benchmark from the Spring Blog](#):

WebMVC process startup time (ms) and RSS after startup (Mb)



You may wonder why all those performance improvements are not enabled by default... well there are some corner cases (if you use different profiles that include different libraries, if some code tries to connect to a production only resource during training run, etc.) that could at best make the improvements not effective, and at worst break the build...

That said, if you opt in to improvements that are not compatible with your Spring Boot version, the buildpack will know and tell you!

What About Java 26? And After?

Java 26 brings in many new features, but the one feature I believe will have the most impact on the Paketo Buildpacks is [AOT Object Caching](#), enhancing Project Leyden (CDS / AOT Cache) to support all garbage collectors, including ZGC; before that only select garbage collectors could leverage this optimization feature (see, another example you can't apply automatically those performance improvements).

After Java 26 support is added by a supported vendor, we'll add this new version to the associated buildpack in under a week BUT we'll still keep Java 21 as the default version [until September 2026](#) when we'll have Java 25 set as the default version and Java 26 will be dropped for Java 27 (only Java LTS versions stay for a long time, even maybe... forever?!)

And after Java 26? Well, we know our users want more flexibility with base images (in particular the ability to use any distroless image easily) and also extended SBOM support.

› [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



Stop Writing YAML: How to Define, Test, and Deploy Your Cloud in Pure Java

API and Frameworks

CI/CD

DevOps

OpenSource

CloudInfrastructure

Author: Wladi Mitzel

Wladi Mitzel is a Solutions Architect at Pulumi, where he works with engineering organizations to adopt infrastructure as code, build internal developer platforms, and structure their teams around cloud-native practices. He spent over a decade as a Java developer, building enterprise systems in Java EE and Spring Boot before taking on DevOps and platform engineering responsibilities during a large-scale cloud migration. That experience, seeing firsthand how infrastructure decisions affect developer velocity, led him into cloud architecture. He went on to spend four years at AWS as a Solutions Architect, specializing in containers and Kubernetes and guiding large enterprises through cloud transformations. He later joined Alibaba Cloud, advising on cloud architecture, AI workloads, and compliance in regulated environments. Across these roles, a recurring theme has been helping organizations close the gap between application development and infrastructure operations. He is based in Germany.



LinkedIn

As Java developers, we apply rigorous engineering practices to application code: type safety, unit testing, refactoring, code review. Then we switch to deploying that code and write hundreds of lines of YAML, accepting the lack of expressiveness, missing abstractions, and missing flexibility as the cost of doing business. Or worse, we log into the cloud console and click our way through to the deployment.

This article explores an alternative: defining cloud infrastructure in Java, with the same tools and practices we use for application code. We'll build a working example, a Java service deployed on AWS, covering the full lifecycle from resource definition through testing to CI/CD. While this example uses Quarkus, the infrastructure patterns demonstrated here apply equally for Spring Boot, Jakarta EE, or any other containerized Java framework. The companion code is available at github.com/wlami/stop-writing-yaml-javapro.

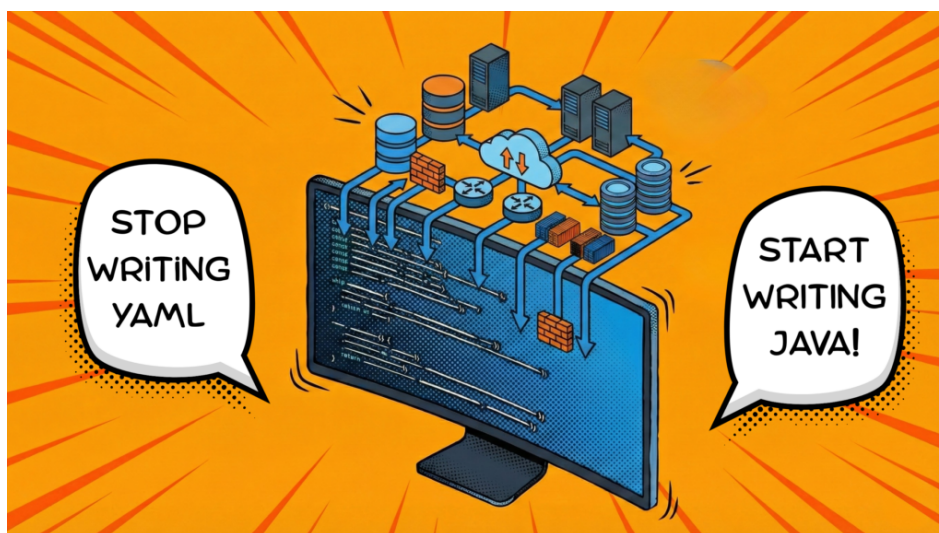


Abb. 3: When your Java code builds the cloud: infrastructure as code turns resource definitions into real architecture.

How We Got Here

Infrastructure management has evolved through distinct stages:



Manual provisioning through cloud consoles and dashboards introduced the “snowflake” problem: each environment was unique, unrepeatable, and prone to human error.

Scripted automation via shell scripts or Python improved repeatability but often lacked idempotency: running the same script twice could produce different results or fail on the second run.

Declarative infrastructure-as-code solved idempotency. Tools like CloudFormation, Terraform, and Kubernetes popularized YAML and domain-specific languages like HCL, the language Terraform uses, for defining a “desired state”. The tool reconciles actual infrastructure with the declared state.

This third stage is where most organizations are today, and where the problems start.

Where Declarative DSLs Break Down

YAML and HCL are configuration formats, not programming languages. Need to create resources across multiple availability zones with alternating configurations? In Java, that is a loop with conditional logic. In configuration languages like HCL, however, expressing logic like this can be awkward or even impossible.

The absence of proper abstraction mechanisms leads to copy-paste reuse: the same anti-pattern we would reject in application code. CloudFormation templates routinely exceed 1000 lines for moderately complex setups.

The developer experience gap is stark. In your Java IDE, you have autocomplete, type checking, safe refactoring, and instant test feedback. In YAML, a misplaced indentation silently breaks your deployment. Refactoring means find-and-replace across files. Testing means deploying to the cloud and waiting.

A concrete example of what this gap looks like. An SQS queue in CloudFormation YAML:

```
1 AWSTemplateFormatVersion: '2010-09-09'
2 Resources:
3   ProcessorQueue:
4     Type: AWS::SQS::Queue
5     Properties:
6       QueueName: processor-queue
7       VisibilityTimeout: 300
8       MessageRetentionPeriod: 86400
```

Newer infrastructure-as-code tools advance by allowing you to define resources in a general-purpose language. Here is the same queue in Java, using Pulumi.

```
1 var queue = new Queue("processor-queue", QueueArgs.builder()
2   .visibilityTimeoutSeconds(300)
3   .messageRetentionSeconds(86400)
4   .build());
```

Both define the same resource. But in the Java version, your IDE autocompletes `visibilityTimeoutSeconds`, the compiler rejects `visibilityTimeoutSecond` (note the missing 's'), and you can CMD+click into the `QueueArgs` source to read every available property. In the YAML version, you learn about typos when the deployment fails.

When Infrastructure Becomes Invisible

A story from a previous employer: months after I left, a colleague deleted what appeared to be a “demo bucket” I had created via ClickOps. With no code, no documentation, no tracked dependencies, nobody knew what that bucket did... until internal developer tooling broke.

This is not unique to ClickOps. Any infrastructure that cannot be discovered, refactored, or tested using standard developer tools is at risk. When there is no compile-time checking, no CMD+click navigation, no “find all usages”, critical infrastructure becomes invisible until something breaks.

Infrastructure in Real Languages

The idea is straightforward: treat infrastructure the same way we treat application code. Use real programming languages with real tools.

The Landscape

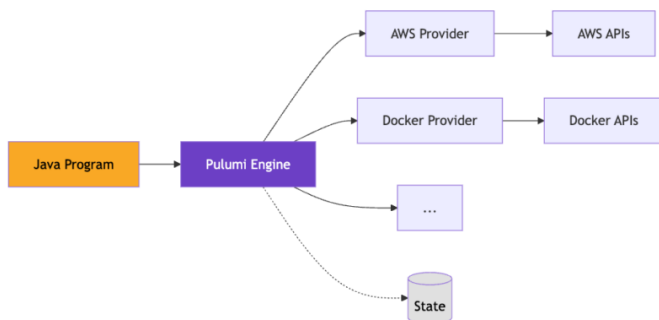
Several tools allow you to write your infrastructure code this way. AWS Cloud Development Kit (CDK) lets developers define AWS infrastructure using TypeScript, Python, Java, and other languages, but is limited to AWS. HashiCorp's CDKTF (CDK for Terraform) brought programming-language support to Terraform's multi-cloud model but was deprecated in late 2025.

Pulumi is an open-source, multi-cloud infrastructure-as-code tool supporting general-purpose languages, including Java. The core engine is Apache 2.0 licensed. This article uses Pulumi for its examples, but the underlying idea, infrastructure as testable, refactorable code, applies to any infrastructure-as-code tool that uses general-purpose programming languages.

How Pulumi Works

With Pulumi, you write a standard Java application that describes infrastructure. The Pulumi CLI executes your program and provisions resources across cloud providers.

The following diagram shows the runtime architecture. Your Java program communicates with the Pulumi engine, which dispatches resource operations to provider plugins. Each provider translates resource declarations into cloud API calls. State is persisted between runs, so the engine can compute the minimal set of changes needed:



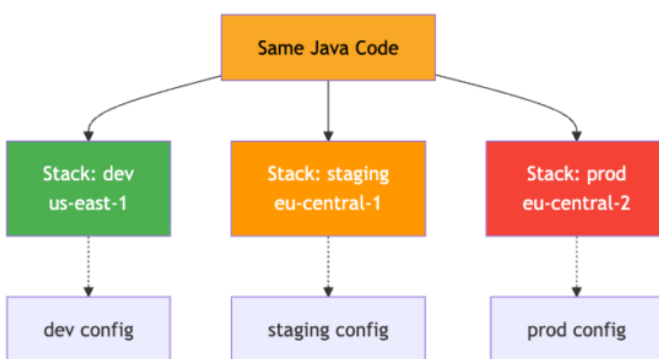
The key concepts:

Resources are strongly-typed Java objects that represent cloud resources (S3 buckets, RDS databases, Lambda functions). You instantiate them, set properties, and Pulumi manages creation order and dependencies.

Outputs wrap values that are not yet known, such as a resource's ID or endpoint, before they are created. `Output<T>` is conceptually similar to `CompletableFuture<T>`: you compose outputs using `.applyValue()` and `Output.format()` instead of accessing the values directly.

Providers are plugins that communicate with cloud APIs: AWS, Azure, GCP, Kubernetes, and 120+ others. They provide type-safe access to every resource property.

Stacks are isolated instances of your program. The same Java code deploys to different environments: dev, staging, production. Each with its own configuration and separate state:



This also enables per-developer stacks, so engineers work in isolation without sharing a dev environment. Stack configuration includes built-in secrets management: values marked as secret are encrypted before they are written to the config file and remain encrypted in state and masked in CLI output. No external secrets manager is required for this to work.

Components bundle multiple resources into reusable abstractions: the infrastructure equivalent of extracting a class from duplicated code.

Building a Real Application

Let's build something concrete: a containerized Java service that reads messages from a queue, processes them, and writes results to a PostgreSQL database, deployed on AWS. In this example, we'll use Quarkus, a Java framework optimized for containers and an alternative to Spring Boot. The infrastructure patterns shown here work with any containerized Java application.

Project Structure

Two separate Gradle projects:

- `quarkus-processor/`: the application with its Dockerfile
- `infrastructure/`: Pulumi infrastructure code

The infrastructure project pulls in the Pulumi SDK and provider packages:

```
1 dependencies {
2     implementation 'com.pulumi:pulumi:[1.3,2.0)'
3     implementation 'com.pulumi:aws:6.+'
4     implementation 'com.pulumi:docker-build:0.+'
5 }
```

The entry point is a standard `main` method. Inside `Pulumi.run()`, you define resources:

```
1 public static void main(String[] args) {
2     Pulumi.run(ctx -> {
3         // all resource definitions go here
4     });
5 }
```

Defining Resources

For the complete application, we need a Virtual Private Cloud with subnets and security groups, an SQS message queue, an RDS PostgreSQL database, a container registry (ECR) repository, a Docker image build, and an ECS Fargate service. The full implementation is roughly 200 lines of Java, compared with over 1,000 lines of equivalent CloudFormation YAML.

Here's the Docker image build, which compiles the Quarkus application and pushes it to ECR as part of the infrastructure deployment:

```
1 var image = new Image("app-image", ImageArgs.builder()
2     .context(BuildContextArgs.builder()
3         .location("../quarkus-processor")
4         .build()
5     ).push(true)
6     .tags(ecrRepository.repositoryUrl()
7         .applyValue(url -> List.of(url + ":latest")))
8     .registries(RegistryArgs.builder()
9         .address(ecrRepository.repositoryUrl())
10        .username(authToken.applyValue(token -> token.userName()))
11        .password(authToken.applyValue(token -> token.password()))
12        .build()
13    ).build());
```

When you run the `pulumi` CLI with the `up` operation, Pulumi invokes Docker to build the Quarkus application from its Dockerfile, pushes the image to ECR, and then provisions the ECS task referencing that image. Application code and infrastructure are deployed together.

Infrastructure Logic in Java

200 lines of raw resource definitions get the job done, but real infrastructure has business requirements that vary. Different environments need different database configurations. Production needs replicas and extended backup retention. Dev does not. In a DSL like HCL or YAML, expressing this kind of conditional logic means reaching for workarounds: `count`, `for_each`, ternary expressions, and nested conditionals that become difficult to read and impossible to unit test.

In Java, it is a function.

Consider a method that maps Service Level Agreement (SLA) tiers to database configurations. `SLATier` is an enum with three values: `BRONZE`, `SILVER`, `GOLD`. `DatabaseConfig` is a record that captures the infrastructure decisions: whether to use Aurora or plain RDS, how many days to retain backups, whether to enable monitoring, and how many read replicas to create.

```

1 public static DatabaseConfig getConfigForSLA(SLATier tier) {
2     return switch (tier) {
3         case BRONZE -> DatabaseConfig.builder()
4             .useAurora(false) // Single RDS instance
5             .backupDays(1)
6             .monitoring(false)
7             .replicaCount(0)
8             .build();
9         case SILVER -> DatabaseConfig.builder()
10            .useAurora(true) // Aurora cluster
11            .backupDays(7)
12            .monitoring(true)
13            .replicaCount(1) // 1 read replica
14            .build();
15        case GOLD -> DatabaseConfig.builder()
16            .useAurora(true) // Aurora cluster
17            .backupDays(30)
18            .monitoring(true)
19            .replicaCount(2) // 2 read replicas
20            .build();
21    };
22 }

```

This is a pure function. It takes an enum and returns a record. There are no cloud calls, no Pulumi engine, and no side effects. We will test it with JUnit in the next section.

The returned config drives resource creation with ordinary control flow. An **if** statement picks the database engine. A **for** loop creates replicas:

```

1 var config = getConfigForSLA(SLATier.SILVER);
2
3 if (config.useAurora()) {
4     //Define an Aurora Cluster
5     var cluster = new Cluster("app-db", ClusterArgs.builder()
6         .engine("aurora-postgresql")
7         .databaseName("processor")
8         .backupRetentionPeriod(config.backupDays())
9         .storageEncrypted(true)
10        .build());
11    //Add replicas which increase availability
12    for (int i = 0; i < config.replicaCount(); i++) {
13        new ClusterInstance("app-db-replica-" + i,
14            ClusterInstanceArgs.builder()
15                .clusterIdentifier(cluster.id())
16                .instanceClass("db.t3.medium")
17                .engine("aurora-postgresql")
18                .build());
19    }
20 } else {
21     new Instance("app-db", InstanceArgs.builder()
22         .engine("postgres")
23         .instanceClass("db.t3.medium")
24         .allocatedStorage(20)
25         .backupRetentionPeriod(config.backupDays())
26         .build());
27 }

```

BRONZE creates a single RDS instance. **SILVER** creates an Aurora cluster with one read replica. **GOLD** creates Aurora with two replicas and 30-day backups. Change the enum value, and the infrastructure changes with it.

This is the kind of logic that is natural in Java and awkward in a configuration language. An **if/else** that chooses between two different resource types, and a loop that creates a variable number of resources based on a computed value. The equivalent in HCL requires restructuring the configuration around **count** and conditional expressions that obscure the intent.

Grouping Resources into Components

As this kind of logic grows, you want to package it for reuse. Pulumi provides a mechanism for this: component resources. A component is a Java class that extends Pulumi's base class `ComponentResource` and creates child resources in its constructor. From the outside, it looks like a single resource with its own inputs and outputs. Inside, it can generate as many real cloud resources as necessary.

In the companion repository, the database logic above is wrapped into a `ManagedDatabaseComponent`, and the ECS Fargate setup (cluster, task definition, IAM roles, logging) is wrapped into a `ContainerServiceComponent`. The main program uses them like this:

```
1 var database = new ManagedDatabaseComponent("app-db",
2     DatabaseArgs.builder()
3         .slaTier(SLATier.SILVER)
4         .username("appuser")
5         .password(config.requireSecret("dbPassword"))
6         .databaseName("processor")
7         .build());
8
9 var service = new ContainerServiceComponent("processor-service",
10     ContainerServiceArgs.builder()
11         .image(imageRef)
12         .environment(Map.of(
13             "DATABASE_URL", database.connectionString(),
14             "QUEUE_URL", queue.url()))
15         .slaTier(SLATier.SILVER)
16         .port(8080)
17         .build(),
18     subnetIds, sgIds);
```

The caller does not need to know whether `ManagedDatabaseComponent` creates a single RDS instance or an Aurora cluster with replicas. That decision is internal, driven by the SLA tier.

A note on the password: `config.requireSecret("dbPassword")` reads a value from the stack's configuration file where it is stored as ciphertext:

```
1 infrastructure:dbPassword:
2   secure: v1:LoMrsybSW3y3T+YY:Zw1vs1Ey2U8s8+Qf2CzC2p7vds0R2Na1QP6LVA==
```

Pulumi decrypts it at deployment time and returns an `Output<String>`. The secret propagates through the resource graph with the same protection: it is masked in logs and encrypted in the state file. In Terraform, values stored in state are plaintext by default. Here, encryption is the default for any value marked as secret.

Running `pulumi up` evaluates the program, builds the dependency graph, and shows a preview:

1	Previewing update (dev)		
2			
3	Type	Name	Plan
4	+ pulumi:pulumi:Stack	infrastructure-dev	create
5	+ custom:database:ManagedDatabase	app-db	create
6	+ aws:rds:Cluster	app-db	create
7	+ aws:rds:ClusterInstance	app-db-primary	create
8	+ aws:rds:ClusterInstance	app-db-replica-0	create
9	+ custom:container:ContainerService	processor-service	create
10	+ aws:cloudwatch:LogGroup	processor-service-logs	create
11	+ aws:ecs:Cluster	processor-service-cluster	create
12	+ aws:iam:Role	processor-service-exec-role	create
13	+ aws:iam:Role	processor-service-task-role	create
14	+ [...]	[...]	
15	+ aws:ecs:TaskDefinition	processor-service-task	create
16	+ aws:ecs:Service	processor-service-service	create
17	+ aws:ec2:Vpc	app-vpc	create
18	+ aws:ec2:Subnet	app-subnet-a	create

19	+		aws:ec2:Subnet	app-subnet-b	create
20	+		aws:ec2:SecurityGroup	app-sg	create
21	+		aws:sqs:Queue	processor-queue	create
22	+		aws:ecr:Repository	app-repo	create
23	+		docker-build:Image	app-image	create
24					
25	Resources: +21 to create				

The tree shows how components organize resources. `app-db` and `processor-service` appear as parent nodes that group their child resources. After confirmation, Pulumi builds the container, pushes it to ECR, and provisions all 21 resources. If you run it again, Pulumi compares its state with your program, finds no differences, and does not change any resources because of its idempotence.

Testing Infrastructure with JUnit

Because the previously introduced `getConfigForSLA` method is a pure function, it's easily testable with plain JUnit. You don't need a Pulumi engine, cloud APIs, or mocks:

```

1  @Test
2  void bronzeSLAUsesSingleRDS() {
3      var config = ManagedDatabaseComponent.getConfigForSLA(SLATier.BRONZE);
4      assertFalse(config.useAurora(),
5          "Bronze tier should use simple RDS, not Aurora");
6      assertEquals(0, config.replicaCount());
7      assertEquals(1, config.backupDays());
8  }

```

The Silver and Gold tests follow the same structure and verify Aurora with 1 or 2 replicas, respectively. The pattern applies to other components too. The `ContainerServiceComponent` extracts resource allocation into static methods:

```

1  @Test
2  void silverTierAllocatesCorrectResources() {
3      assertEquals(512, ContainerServiceComponent.getContainerCpu(SLATier.SILVER, 256));
4      assertEquals(1024, ContainerServiceComponent.getContainerMemory(SLATier.SILVER, 512))
5      ;
6      assertEquals(2, ContainerServiceComponent.getDesiredCount(SLATier.SILVER));
7      assertEquals(30, ContainerServiceComponent.getLogRetentionDays(SLATier.SILVER));
8  }

```

These tests run in milliseconds as part of a normal `./gradlew test`. They run without Pulumi CLI and do not need either cloud credentials or cost-incurring deployments. The design principle: extract infrastructure decisions into pure methods, test those methods with standard JUnit.

What to Test and What Not To

Infrastructure tests should verify your decisions, not your cloud provider's API. Good tests cover SLA-to-configuration mappings, naming conventions, tagging logic, and custom validation. Don't test whether AWS actually creates an RDS instance when you call its API; that's Amazon's job.

The testing pyramid applies to infrastructure the same way it applies to application code:



Unit tests catch configuration errors before any cloud resources are created. Integration tests (deploying to ephemeral stacks) verify that your infrastructure actually works end-to-end.

A Note on Policies

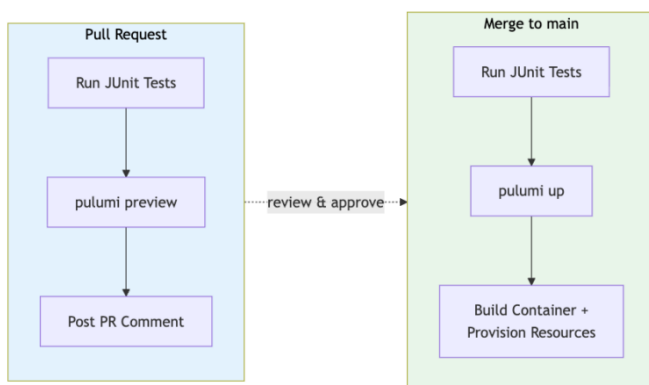
Pulumi's Policy-as-Code feature (CrossGuard) enables organization-wide guardrails: blocking unencrypted storage, enforcing tagging, and limiting instance sizes. Currently, policies can only be authored in TypeScript or Python, though they enforce resources created by any Pulumi language.

Policies and tests serve different purposes. Tests verify your component logic during development. Policies enforce organizational standards during deployment, regardless of which team wrote the code or what language they used.

CI/CD: One Pipeline for Application and Infrastructure

Since the infrastructure is written in Java, it fits into the same CI/CD pipeline as the application.

Our GitHub Actions workflow has two jobs:



For **pull requests**, the pipeline runs JUnit tests, then executes `pulumi preview`. This computes a diff between the current infrastructure and the desired state, without modifying anything. The diff is posted as a PR comment, so reviewers can see exactly which resources would be created, updated, or deleted before the code is merged.

On **merge to main**, the pipeline runs the same tests, then executes `pulumi up`. Pulumi builds the application container, pushes it to ECR, and provisions or updates all resources. If any step fails, the update stops. Resources already created remain in place, and the state file records partial progress, so the next `pulumi up` picks up where it left off.

Infrastructure changes become pull requests. When a developer needs to add an environment variable to the ECS task or resize the database, they change the infrastructure code in the same PR. Tests run, reviewers see the diff, approve, and the change rolls out to production.

Environments

Different environments use the same code with different configurations. Dev gets a single small RDS instance and one ECS task. Production gets the Aurora cluster with read replicas. Same Java code, different config files, completely separate state.

Secrets like database passwords work the same way across different stacks: each environment has its own encrypted values in its config file. In Continuous Integration, the decryption passphrase is usually injected via a pipeline secret. You can also inject secrets from AWS Secrets Manager, GitHub Actions secrets, or other sources your organization uses.

The same pattern works in any CI/CD system. GitLab CI, Jenkins, Azure DevOps: they all just run `pulumi preview` on PRs and `pulumi up` on merges.

Reusable Components as Shared Libraries

Components can be packaged as JARs and published to your internal Maven repository. Teams consume them as Maven/Gradle dependencies:

```
1 dependencies {  
2     implementation 'com.yourcompany.platform:container-service:2.1.0'  
3     implementation 'com.yourcompany.platform:managed-database:1.5.0'  
4 }
```

This changes the platform team's role. Instead of being gatekeepers who manually provision infrastructure, they become library authors who encode organizational standards into versioned components. Application teams consume these components and operate independently, within guardrails enforced by the component code itself.

When the platform team needs to roll out a change, enabling Performance Insights on all Gold-tier databases, for instance, they update the component and publish a new version. Teams adopt it on their own schedule, just as they'd adopt any library update.

Pulumi also supports multi-language components: write a component in Java, and Pulumi generates typed SDKs for TypeScript, Python, and Go. This matters in polyglot organizations where standardizing on one language isn't realistic.

Working with Existing Infrastructure

The question always comes up: "We have hundreds of resources already running in production. We can't start over."

You don't have to.

Importing Resources

Pulumi CLI can adopt existing cloud resources into its state using the `pulumi import` operation. You write the Java code describing a resource, then tell Pulumi to import it instead of creating a new one:

```
1 var existingDb = new DbInstance("legacy-db", DbInstanceArgs.builder()
2   .identifier("production-user-db")
3   .engine("postgres")
4   .instanceClass("db.t3.medium")
5   .allocatedStorage(100)
6   // ... other properties matching current state
7   .build(), CustomResourceOptions.builder()
8     .import_("production-user-db") // import_ with underscore - "import" is a Java
      keyword
9     .build());
```

Pulumi reads the current state from AWS, verifies it matches your code, and adopts the resource. From that point forward, drift between code and actual infrastructure is detected on every `pulumi preview`.

Coexisting with Terraform

Pulumi and Terraform can manage different resources in the same environment without conflict: they maintain separate state. Start new services with Pulumi while existing infrastructure stays in Terraform. Using the `terraform-state` provider, Pulumi can also read Terraform state files to reference outputs, allowing your Pulumi program to look up a database connection string from Terraform-managed resources without migrating them.

Converting YAML to Code

The `pulumi convert` command converts CloudFormation, Terraform HCL, and Kubernetes YAML to Pulumi code in any supported language:

```
1 # Convert Kubernetes YAML to Java
2 pulumi convert --from kubernetes --language java --out ./pulumi-app
3
4 # Convert CloudFormation to Java
5 pulumi convert --from cloudformation --language java --out ./pulumi-infra
```

The generated code is a starting point, not a finished product. The value comes after conversion: five copy-pasted database definitions become a loop, repeated configuration becomes a method, and hardcoded values become parameters.

When to Migrate

Not everything needs to be migrated. Stable Terraform modules that manage infrastructure that rarely changes can remain unchanged. Focus on infrastructure that changes frequently, is hard to manage in its current form, or is tied to services under active development.

The pragmatic question isn't "should we migrate everything?" but "where will migration actually reduce errors or speed up the team?" Infrastructure nobody touches can stay as-is until there's a real reason to change it.

Trade-Offs

Using a general-purpose language for infrastructure has costs. They are worth understanding before committing to the approach.

Verbosity for simple cases. A domain-specific language optimized for resource declarations can be more concise. A three-line HCL block expands to six or seven lines of Java with builder and `.build()` calls. If your infrastructure consists of only a few static resources with no conditional logic, a DSL might be the better choice. The crossover point occurs when infrastructure involves real logic, conditionals, loops, and environment-specific behavior, at which DSLs begin to strain.

Deferred values. Any tool that lets you write imperative code against resources that don't exist yet needs a way to represent values that are not yet known. Pulumi uses `Output<T>`, CDK uses `Token`. This is not a quirk of a specific tool; it is a consequence of the model. Composing these deferred values adds friction: string concatenation becomes `Output.format()`, and code that threads outputs through multiple resources reads more like reactive programming than straightforward Java.

Team readiness. Writing infrastructure in Java does not turn infrastructure engineers into Java developers, and writing Java does not make application developers infrastructure-literate. The approach works best when teams already have some overlap between these skills or are willing to deliberately develop it. Without that investment, a shared language can become a shared source of confusion.

These costs are front-loaded. The return increases with complexity: the more logic, environments, and cross-team reuse your infrastructure involves, the greater the investment grows.

Conclusion

We covered a working example: a Quarkus service with its entire infrastructure defined in Java. The infrastructure code uses standard language features: switch expressions for SLA tier mappings, loops for read replicas, and records for configuration data. Tests are run with JUnit, deployment is handled by GitHub Actions, and the components are reusable Java classes.

The central argument is not about Pulumi specifically. It is that infrastructure code deserves the same engineering practices we apply to application code: type safety, testing, refactoring, and code review. Tools like Pulumi and AWS CDK make this possible. Whether the approach fits your team depends on the complexity of your infrastructure and the extent to which it involves real logic rather than static resource declarations.

The complete implementation (components, tests, CI/CD workflow) is at github.com/wlami/stop-writing-yaml-javapro.

› **Back to the Table of Contents** ‹



Beyond REST: Rethinking APIs with GraphQL and Java

API and Frameworks

GraphQL

REST

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.



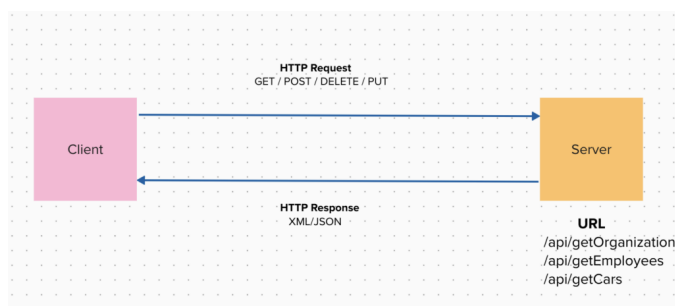
LinkedIn

Distributed systems need little introduction today. They form the backbone of modern software architectures, enabling independent components to communicate, coordinate, and share resources seamlessly at scale. With the industry's shift toward microservices, the importance of distributed systems has grown exponentially. In this landscape, APIs have emerged as the primary language of communication. APIs follow different implementation patterns. It is essential to think beyond simple request-response models and REST most of the time. For any API, “know your consumer” is the base principle. This blog explores a real-world use case and explains why it requires rethinking traditional request-response patterns and REST-based designs.

Rest and Beyond

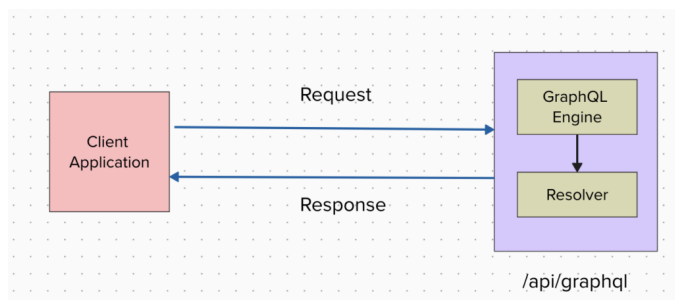
REST: Simple, Stateless and Everywhere

REST has been the undisputed champion for building web APIs. Its simplicity, statelessness and standard HTTP methods baseline make it the go to architecture even in today's data intensive world. REST treats everything as a resource, identifies each resource using a URI, and operates on it using standard HTTP methods.



GraphQL: Flexible, Typed and Client-Driven

GraphQL(<https://graphql.org/>), developed by Facebook in 2012 and open-sourced in 2015 is a query language for APIs. People describe it as the SQL for APIs. The main advantage of GraphQL over REST is that you ask for what data you want and you get exactly that. GraphQL provides a single endpoint. It maintains a strong contract between the client and server through its schema.



Rethinking APIs for AI Agents: The GraphQL Advantage

When building AI agents and tools one critical decision is choosing between REST and GraphQL. Requests are often unstructured and expressed in plain text. LLMs enable agents to understand intent and map these requests to the appropriate APIs that can fulfill them. REST can be challenging where we need to involve multiple endpoints. Let's examine a real-world scenario in which conversational agents and bots process a plain-text query and produce a structured, accurate response.

Assume we have an application that contains organizations, employees and their car information.

The Example Query

“Give me the name of the employee who works for IBM and owns a Civic model car_”

REST

With REST, this requires

- Finding employees who work at Organization “IBM”
- Filtering those who own a Honda civic car
- Returning only the name of the employee

For easy consumption, let's assume we have a specialized endpoint that has some advanced search functionalities that can accept organization name and car information in a single request.

```

1 Request:
2 GET /api/search?organizationName=IBM&carModel=Civic
3
4 Response:
5 {
6   "employees": [{
7     "id": "1",
8     "firstName": "Tom",
9     "lastName": "Bob",
10    "email": "tom.bob@ibm.com",
11    "position": "Software Engineer",
12    "organizationId": "1"
13  }],
14  "cars": [{
15    "id": "1",
16    "model": "Civic",
17    "brand": "Honda",
18    "employeeId": "1"
19  }],
20  "organizations": [{
21    "id": "1",
22    "name": "IBM Corp",
23    "address": "123 Business St"
24  }]
25 }
```

If a single request that can handle this scenario is not available then we will end up involving multiple endpoints to get the desired results. Despite, of having a specific endpoint that can cover this scenario, let's see to the challenges that we can find in this particular use case.

- Over fetching of data(Like address, email id, position) and this could lead to an overhead of processing the data before it can be served to the user.
- Requires custom endpoints like above for every pattern.

GRAPHQL

With GraphQL, we can easily get rid of supporting multiple endpoints to cover use case as it deals with a single endpoint.

```

1 Schema:
2
3 type Query {
4   advancedSearch(filters: SearchFilters): SearchResult!
5 }
6
7 input SearchFilters {
8   carModel: String
9   organizationName: String
10  # ... other filters
11 }
12
13 type SearchResult {
14   cars: [Car!]!
```

```

15  employees: [Employee!]!
16  organizations: [Organization!]!
17  }
18
19  type Employee {
20    id: ID!
21    firstName: String!
22    lastName: String!
23    email: String!
24    position: String
25    organization: Organization!
26    car: Car
27  }

```

The above schema defines the data model of the underlying resources, making it strongly typed and establishing a clear contract between the client and the server. Each field that represents a complex type is mapped to an underlying resolver implementation. In this blog, the resolvers are implemented using the Java-based [GraphQL-Java](#) library. The same approach can be applied using any framework or language

```

1  Query:
2  query AdvancedSearch {
3    advancedSearch(filters: { carModel: "Civic", organizationName: "IBM" }) {
4      employees {
5        firstName
6        lastName
7      }
8    }
9  }
10
11 Response:
12 {
13   "data": {
14     "advancedSearch": {
15       "employees": [
16         {
17           "firstName": "Tom",
18           "lastName": "Bob"
19         }
20       ]
21     }
22   }
23 }

```

As shown in the example above, the response remains crisp, brief, and exactly to the request. This brings in advantage for an LLM to consume the response without any further processing. Here, GraphQL offers significant advantages.

- It Handles complex, highly interconnected data without any need to fetch it from multiple API endpoints.
- It returns simple responses tailored exactly to the request. This avoids over fetching and under fetching.
- Supports a wide variety of complex query patterns from a single domain model.
- Enables seamless mapping from natural language to API requests as the data maintains a full relationship with the domain model.

CONCLUSION

This blog uses a specific use case to demonstrate scenarios where GraphQL can outperform REST. It does not suggest that GraphQL is a replacement for REST. The two are complementary technologies, each suited to different problem spaces. Architectural decisions should be guided by a clear understanding of the consumer and the problem being solved. REST continues to be highly effective for use cases such as

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



Solving Spring AI's UI Challenge with AG-UI's Java SDK

API and Frameworks

AG-UI

Spring AI

Author: Pascal Wilbrink

Pascal Wilbrink is a senior software engineer with over a decade of experience and a core member of the AI Collective Netherlands. He is passionate about making AI more accessible through clean code, intelligent automation, and clear documentation. Specializing in Java, AI agents, and frontend development, Pascal focuses on the practical side of AI integration—turning complex problems into resilient, real-world solutions. When he's not building open-source tools, he's helping developers build smarter applications, one SDK at a time.



LinkedIn

Introduction: The Missing Link in Java AI Development

Ask any Java developer who has tried to build an AI feature in a production application and they'll tell you the same thing: the backend isn't the hard part anymore. Spring AI makes it easy to invoke an LLM, define prompts, and create tools. But turning these interactions into a smooth, interactive user experience? That's where the struggle begins.

Most developers solve this by inventing custom message formats, WebSocket events, and ad-hoc rules for how the UI should render tool calls. Each application reinvents the bridge between frontend interactions and agent intelligence. The result? A fragile, inconsistent layer that must be rebuilt for every project.

AG-UI addresses this gap head-on. It provides a clean, protocol-driven way for frontends and AI backends to communicate—standardized messages, tool calls, streaming events, and contextual data. And with the **AG-UI Java SDK**, Java developers now have a first-class implementation of that protocol for Spring Boot.

In this article, we'll explore how AG-UI fills the missing link between user interfaces and Java-based AI agents, demonstrating practical integration with Spring AI and CopilotKit.

What is AG-UI? And Why Does It Matter?

As AI applications evolve, the interaction model is shifting from simple prompt-response patterns to rich, agent-driven workflows. Users expect conversational interfaces that can call tools, display intermediate steps, and adapt to context. What's been missing is a **standard way** for frontends and AI backends to speak the same language. **AG-UI** is that missing standard.

At its core, AG-UI is a **client-server protocol** designed specifically for agent-based user interfaces. It defines how a UI sends events to an agent, how an agent streams back state updates, and how both sides coordinate tool calls and contextual information.

AG-UI handles:

- **Streaming agent messages** and intermediate reasoning steps
- **Tool execution** triggered from backend or UI
- **Shared state** between frontend and agent
- **Multi-agent orchestration** and message routing
- **Structured observability** for debugging and monitoring



Why does this matter for Java developers? Because UI expectations are rising. An AI that simply returns text is no longer enough. AG-UI provides the protocol to make modern, interactive AI experiences possible, and the Java SDK delivers it in a form that fits naturally into the Spring ecosystem.

Spring AI's Strength (and Its Blind Spot)

Spring AI has quickly become the go-to solution for Java developers who want to bring LLM capabilities into their backend services. It provides a clean abstraction for prompt templates, model operations, and tool definitions—all using patterns familiar to anyone who has worked with Spring Boot. But Spring AI deliberately stops at the backend layer. It focuses on intelligence and orchestration, not on how those interactions reach the frontend. Modern AI applications require far more than a simple `/chat` endpoint. Users expect agents that stream intermediate reasoning, execute tools dynamically, react to UI events, and present structured outputs in real time. Without a standard communication layer, every team ends up building this themselves: custom WebSocket endpoints, ad-hoc JSON schemas, manual state synchronization, and brittle tool call coordination.

AG-UI fills this gap. It provides a standardized protocol that sits between Spring AI and the frontend, giving Java developers a ready-made communication layer for agent-driven interfaces. Spring AI handles intelligence, AG-UI handles interaction, and your application becomes simpler and more maintainable.

The AG-UI Java SDK: Protocol Made Practical

A protocol is only useful if developers can adopt it easily. That's why the **AG-UI Java SDK** was built: to bring AG-UI's event model and agent lifecycle to the Spring ecosystem in a clean, idiomatic way.

At the foundation is **streaming HTTP**. AG-UI uses a long-lived HTTP connection to stream events between frontend and backend, avoiding WebSocket complexity while still enabling continuous, event-driven communication. The SDK handles the entire pipeline automatically.

The Java SDK provides three core capabilities:

- 1. Streaming Event Handling** Incoming AG-UI events (user messages, UI commands, context updates) are deserialized into typed Java objects. Outgoing events—agent messages, state updates, tool invocations—stream back via server-sent events.
- 2. Agent Orchestration** The SDK connects events to your agent logic, whether you're using Spring AI, LangChain4j, or custom engines. It manages sessions, state transitions, and structured responses without glue code.
- 3. Tool Invocation** Tools triggered from UI actions are simple Java classes or Spring components. When an agent emits a tool call, the SDK executes the method and streams results back with proper metadata.

The result: your UI speaks AG-UI, your backend speaks AG-UI, and the SDK ensures both sides stay perfectly aligned.

A Real-World Integration: Setting Up AG-UI with Spring Boot

Let's walk through an actual implementation. Setting up AG-UI with Spring AI requires three components: an agent configuration, a controller endpoint, and the AG-UI service.

Step 1: Add Dependencies

```
1 <dependency>
2   <groupId>com.ag-ui.community</groupId>
3   <artifactId>spring-ai</artifactId>
4   <version>1.0.1</version>
5 </dependency>
6 <dependency>
7   <groupId>com.ag-ui.community</groupId>
8   <artifactId>spring-server</artifactId>
9   <version>1.0.1</version>
10 </dependency>
```


Step 2: Configure Your Agent

```
1 @Configuration
2 public class AgentConfig {
3     @Bean
4     public SpringAIAgent chatAgent(ChatModel chatModel) {
5         ChatMemory memory = MessageWindowChatMemory.builder()
6             .chatMemoryRepository(new InMemoryChatMemoryRepository())
7             .maxMessages(10)
8             .build();
9         return SpringAIAgent.builder()
10            .agentId("assistant")
11            .chatModel(chatModel)
12            .chatMemory(memory)
13            .systemMessage("You are a helpful AI assistant.")
14            .state(new State())
15            .build();
16     }
17 }
```

Step 3: Create the Endpoint

```
1 @Controller
2 public class AgUiController {
3     @Autowired private AgUiService agUiService;
4     @Autowired private SpringAIAgent chatAgent;
5
6     @PostMapping("/agui/chat")
7     public ResponseEntity<SseEmitter> chat(@RequestBody AgUiParameters params) {
8         SseEmitter emitter = agUiService.runAgent(chatAgent, params);
9         return ResponseEntity.ok()
10            .cacheControl(CacheControl.noCache())
11            .body(emitter);
12     }
13 }
```

That's it. Your Spring Boot application now speaks AG-UI natively, streaming agent responses and events to any compatible frontend.

Connecting CopilotKit: The Frontend Side

CopilotKit is a React framework for building AI-powered interfaces that works seamlessly with AG-UI. It provides components for chat, agent state, and tool rendering—all protocol-aware.

Step 1: Install CopilotKit

```
1 npm install @copilotkit/react-core @copilotkit/react-ui
```

Step 2: Configure the Provider

```

1 import { CopilotKit } from "@copilotkit/react-core";
2 import { CopilotSidebar } from "@copilotkit/react-ui";
3
4 function App() {
5   return (
6     <CopilotKit runtimeUrl="http://localhost:8080/agui/chat">
7       <CopilotSidebar>
8         <YourApplication />
9       </CopilotSidebar>
10    </CopilotKit>
11  );
12 }

```

Step 3: Use the Chat Interface

```

1 import { useCopilotChat } from "@copilotkit/react-core";
2
3 function ChatComponent() {
4   const { messages, sendMessage } = useCopilotChat();
5
6   return (
7     <div>
8       {messages.map(msg => (
9         <div key={msg.id}>{msg.content}</div>
10      ))}
11     </div>
12   );
13 }

```

CopilotKit handles all AG-UI protocol details—parsing streaming events, rendering tool calls, managing state updates. The frontend and Spring Boot backend communicate through the standardized AG-UI protocol, with no custom integration code required.



Adding Tools: Making Agents Actionable

Agents become powerful when they can execute actions. In AG-UI + Spring AI, tools are standard Spring AI `ToolCallback` instances that the agent can invoke during conversations.

```

1 public class WeatherTool implements Function<WeatherRequest, String> {
2   @Override
3   public String apply(WeatherRequest request) {
4     return "Temperature in " + request.location() + " is 22°C";
5   }
6 }
7
8 record WeatherRequest(String location) {}

```

Register with Agent

```

1  @Bean
2  public SpringAIAgent weatherAgent(ChatModel chatModel) {
3      ToolCallback weatherTool = FunctionToolCallback.builder(
4          "get_weather",
5          new WeatherTool()
6      )
7      .description("Get current weather for a location")
8      .inputType(WeatherRequest.class)
9      .build();
10
11     return SpringAIAgent.builder()
12         .agentId("weather-assistant")
13         .chatModel(chatModel)
14         .toolCallback(weatherTool)
15         .systemMessage("You are a weather assistant.")
16         .build();
17 }

```

Tool Execution Flow

The AG-UI protocol streams each step to the frontend in real-time, creating transparent, observable AI interactions. CopilotKit automatically renders tool calls in the UI, showing users exactly what the agent is doing.

Shared State: Context That Travels

One of AG-UI's most powerful features is shared state—a key-value store that travels between frontend and backend with every request. This allows agents to maintain context beyond chat history.

Setting Initial State

```

1  @Bean
2  public SpringAIAgent contextualAgent(ChatModel chatModel) {
3      var state = new State();
4      state.set("user_timezone", "America/New_York");
5      state.set("language", "en");
6
7      return SpringAIAgent.builder()
8          .agentId("contextual-agent")
9          .chatModel(chatModel)
10         .state(state)
11         .systemMessage("You are a helpful assistant.")
12         .build();
13 }

```

Dynamic System Messages

```

1  return SpringAIAgent.builder()
2      .systemMessageProvider(agent -> {
3          String lang = agent.getState().get("language");
4          String tz = agent.getState().get("user_timezone");
5          return String.format(
6              "Respond in %s. User timezone: %s", lang, tz
7          );
8      })
9      .build();

```

Frontend State Updates

```
1 import { useCopilotContext } from "@copilotkit/react-core";
2
3 function LanguageSelector() {
4   const { setContext } = useCopilotContext();
5
6   const changeLanguage = (lang: string) => {
7     setContext({ language: lang });
8   };
9
10  return <button onClick={() => changeLanguage('es')}>Spanish</button>;
11 }
```

State flows bidirectionally: the frontend updates context, the backend receives it immediately, and the agent adapts its behavior accordingly. This creates truly contextual AI interactions.

Why AG-UI Changes the Game for Java Developers

Before AG-UI, Java developers building AI features faced a fundamental problem: **Spring AI stops at the model layer**. You got great tools for calling LLMs and defining prompts, but translating those interactions into a responsive UI required custom infrastructure.

AG-UI solves this by providing:

- 1. A Standard Protocol** No more inventing custom message formats or WebSocket conventions. AG-UI defines exactly how agents and UIs communicate, making your backend instantly compatible with any AG-UI frontend like CopilotKit.
- 2. Zero Boilerplate Streaming** Streaming responses, tool calls, and state updates are handled automatically. You focus on agent logic, not plumbing.
- 3. Framework Flexibility** Use Spring AI, LangChain4j, or any other agent framework. AG-UI adapts to your architecture rather than forcing a specific pattern.
- 4. Production-Ready Features** Multi-agent routing, conversation threading, observability events, and error handling come built-in.

For Java teams, this means **AI features ship faster, with less code, and with better user experiences**. The gap between backend intelligence and frontend interaction finally has a bridge.

Open Source and Community-Driven

AG-UI is **fully open source** and available under the MIT license. The protocol and all SDK implementations are community-driven projects designed to evolve with the needs of developers building agent-based applications.

The **Java SDK and Spring AI integration** were developed by Pascal Wilbrink as a community contribution to bring AG-UI to the Java ecosystem, providing Spring Boot developers with first-class support for the protocol.

Important Note: AG-UI is actively under development and subject to change. The protocol and APIs may evolve as the community identifies improvements and new use cases. We recommend following the repository for updates and participating in discussions about the protocol's direction.

If you're interested in contributing, have feedback, or want to collaborate on improving the Java SDK or the AG-UI protocol itself, please visit:

GitHub Repository: <https://github.com/ag-ui-protocol/ag-ui>

The project welcomes contributions—whether it's improving documentation, adding features, reporting issues, or helping shape the protocol's future. Join the community and help build the standard for agent-driven user interfaces.

Conclusion

Spring AI brought LLM capabilities to Java developers in a familiar, Spring-native way. But getting those capabilities into production applications required solving a harder problem: **how to build interactive, agent-driven user interfaces.**

AG-UI provides the missing piece. It's a protocol-first approach that standardizes the communication layer between AI backends and modern frontends. With the AG-UI Java SDK and frontend frameworks like CopilotKit, Spring Boot developers can now build UI-native AI experiences without reinventing infrastructure for every project.

The result? **Faster development, cleaner code, and AI applications that feel as responsive and interactive as users expect.**

If you're building AI features in Java, AG-UI is worth exploring. It might just be the bridge your application needs.

Resources:

- AG-UI Protocol: <https://github.com/ag-ui-protocol/ag-ui>
- Java SDK: <https://github.com/ag-ui-protocol/ag-ui/tree/main/sdks/community/java>
- CopilotKit: <https://copilotkit.ai/https://copilotkit.ai>
- Example Applications: <https://github.com/ag-ui-protocol/ag-ui/tree/main/sdks/community/java/examples>

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

JCON2026
www.jcon.one

JAVAPRO



APR
20-23

📍 CINEDOM COLOGNE

JCON EUROPE 2026
International Java Community Conference



Virtual Threads Meet AI: Java Concurrency in the Age of Intelligent Systems

AI and ML # Java26 # VirtualThreads # Performance # Concurrency

Author: Damiana Nascimento

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



LinkedIn

AI workloads change how Java applications scale and behave. This article explains how Java 26 and virtual threads simplify concurrency in AI-heavy systems, replacing complex reactive setups with clearer and more efficient designs. It shows when virtual threads outperform traditional async models and how concurrency choices impact system architecture.

Introduction: Java, AI, and the New Concurrency

The rapid growth of AI-driven systems is reshaping how we design and scale Java APIs. RAG systems, LLM inference, and recommendation pipelines bring thousands of simultaneous requests with heavy network I/O, vector databases, and external services.

Traditional approaches — from heavyweight thread-per-request models to complex reactive stacks — struggle to address this new reality. **Java virtual threads for AI concurrency** arrive as the solution: simple synchronous code with massive scalability.

This article shows how Java 26 and virtual threads simplify AI architectures, when they outperform traditional async models, and their real impact on code readability, debugging, and system performance.

AI Workloads in Production: What Changes for Java

Virtual threads fundamentally change how AI systems behave in production. A request hits the Java API. It generates embeddings, queries a vector database, calls an external LLM, and returns the response. Each step involves heavy I/O: network, database, third-party APIs.

Under real load, thousands of requests arrive per second. Each blocks waiting for Vector DBs, LLM providers, or observability services. Traditional models struggle here: platform threads exhaust memory, reactive frameworks create callback hell.

Figure 1 shows this typical flow. Client → Java API → Embeddings → Vector DB → LLM → Response, with metrics flowing parallel.

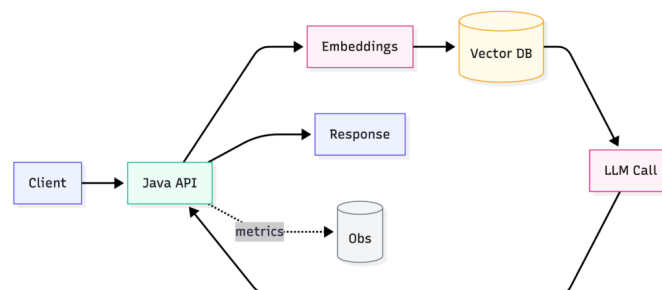


Abb. 4: Figure 1: Typical Java API flow for AI systems with RAG and LLM

Java virtual threads for AI concurrency solve this naturally. Each request gets its lightweight thread. They block without consuming resources. Thousands run simultaneously with readable synchronous code.

From Traditional Threads to Virtual Threads in Java 26

Platform threads have high cost. Each consumes about 1MB of memory and kernel resources. With 10,000 simultaneous requests, you need 10GB just for threads. The OS limits this number.

Project Loom changed that. Introduced in JDK 19, it matured through Java 26. Virtual threads are lightweight threads created by the JVM, not the OS. They typically require only a few kilobytes of stack storage. The JVM manages millions efficiently.

Virtual threads are scheduled by the JVM over a limited set of underlying carrier threads. When a virtual thread encounters a blocking I/O operation, its execution state is temporarily detached, allowing the carrier thread to continue executing other virtual tasks. This is what enables I/O-bound workloads to scale efficiently, CPU-bound tasks, however, keep the carrier busy and prevent other virtual threads from making progress.

Pinning happens when a virtual thread cannot be unmounted, for example inside long **synchronized** blocks or during certain native calls, and that effectively turns it back into an expensive platform thread for the duration of that section

```
1 // Traditional: limited to few threads
2 // Platform: 200 threads = 200MB + kernel overhead each
3 ExecutorService platform = Executors.newFixedThreadPool(200);
4
5 // Virtual Threads: millions of threads, few carriers (multiplexing)
6 var virtual = Executors.newVirtualThreadPerTaskExecutor();
```

In the example above, `newVirtualThreadPerTaskExecutor()` creates one virtual thread per submitted task. This model allows thousands of concurrent blocking operations without proportional memory growth, which is particularly suitable for AI APIs calling LLMs or vector databases.

In Java 26, virtual threads have full support: profiling JFR, Spring/Quarkus integration, and even debuggers showing readable stack traces per virtual thread. Virtual threads are now production-ready and fully supported across the Java ecosystem.

Where Virtual Threads Shine in AI Systems

Inference servers receive thousands of requests per second. Each needs to call an LLM, wait for response, process, and return. Virtual threads shine here: one thread per request, simple synchronous code.

RAG pipelines are another perfect example. Client sends question → embedding → vector search → LLM prompt → response. Each step blocks waiting for I/O. Virtual threads scale this naturally.

```
1 // Example: RAG pipeline with virtual threads
2 try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {
3     var future = exec.submit(() -> {
4         var embedding = embeddingService.generate(text);
5         var ctx = vectorDb.search(embedding);
6         return llm.generate(prompt + ctx);
7     });
8     return future.get();
9 }
```

Each request runs in its own virtual thread. They pause waiting for external APIs. Thousands process simultaneously without exploding memory. **Java virtual threads for AI concurrency** make this trivial.

AI microservices also benefit. Chatbots, recommenders, anomaly detection — all with high volume of blocking calls. Virtual threads eliminate reactive complexity without sacrificing throughput.

Reactive or Virtual Threads? Comparing Models

Reactive frameworks (WebFlux, Vert.x) dominated for years. But they have high cognitive cost: Mono/Flux, operators, back-pressure. Virtual threads offer a simpler alternative for AI workloads.

See the practical comparison in a table:

ASPECT	REACTIVE	VIRTUAL THREADS
Code	Complex (flatMap, zip)	Natural synchronous
Debug	Fragmented stack traces	Complete stack traces
Memory	Fixed pools	A few kilobytes of stack storage per virtual thread
Throughput	Excellent (I/O)	Excellent (I/O)
p99 Latency	Good	Comparable or lower in I/O-bound workloads

Abb. 5: Table 1: Practical comparison between reactive models and Virtual Threads for AI APIs

```
1 // Reactive (WebFlux)
2 Mono<String> result = Mono.fromCallable(() -> embeddingService.generate(text))
3   .flatMap(embedding -> vectorDB.search(embedding))
4   .flatMap(results -> llm.generate(prompt + results));
5
6 // Virtual Threads
7 String result = embeddingService.generate(text);
8 var results = vectorDB.search(result);
9 return llm.generate(prompt + results);
```

For AI APIs with heavy I/O, virtual threads match performance but win on simplicity. **Java virtual threads for AI concurrency** reduce complexity at no cost. Virtual threads excel in I/O-bound workloads. CPU-bound tasks should use dedicated executors or structured parallelism.

The Cost Model Shift: Blocking is No Longer Expensive

For decades, Java architects were strongly incentivized to avoid blocking. Thread-per-request models consumed 1MB+ per connection. Scalability demanded event loops, non-blocking NIO, and reactive streams with Mono/Flux chains.

Virtual threads *don't eliminate reactive programming*. They redefine the economics of blocking.

When a virtual thread performs a blocking I/O operation, the JVM suspends its execution and makes the underlying carrier thread available to run other virtual threads. **Blocking becomes cheap:** mostly scheduling overhead rather than an OS-thread-sized memory footprint..

```
1 // OLD MENTAL MODEL: blocking = scalability killer
2 // NEW REALITY: blocking = natural + efficient
3 String result = llm.generate(prompt); // blocks (I/O) but scales with virtual threads
```

Architectural implications:

- **Synchronous-first design** returns as default for I/O-heavy APIs
- **Reactive becomes optional**—use for streaming, not orchestration
- **Domain clarity > non-blocking purity** guides architecture decisions

Reactive retains value for true streaming (WebSockets, Kafka) and explicit backpressure. But for many **AI workloads** (RAG, inference APIs), virtual threads make blocking the scalable default.

Architectural Impact: Rethinking Java Services for AI

With Loom, AI services can return to synchronous clarity without sacrificing scale. We no longer need complex reactive layers. Synchronous code becomes viable again, now with massive scalability.

Architectures become simpler. Fewer reactive operators, less manual backpressure. Each endpoint can use one virtual thread per request, naturally waiting for external APIs.

Observability improves dramatically. Complete stack traces per virtual thread make debugging easier. JFR and distributed tracing (OpenTelemetry) work natively. See *Figure 1* again.

Backpressure shifts from reactive stream coordination to explicit concurrency control mechanisms such as semaphores, rate limiters, or bounded executors. Limit active virtual threads or use Semaphore for LLM call rate limiting. Control without reactive complexity.

Java virtual threads for AI concurrency enable scalable synchronous microservices. Less code, more readability, same performance. Architecture gains simplicity and maintainability.

Best Practices and Pitfalls with Virtual Threads

Virtual threads are not a silver bullet. Virtual threads shine on I/O, but beware CPU-intensive work. Heavy computation blocks carrier threads. Offload to traditional pools in those cases.

Limit outbound LLM concurrency explicitly using a bulkhead (for example, a Semaphore) and complement it with timeouts and circuit breakers to prevent cascading failures. For instance, cap the system at 50 parallel calls to the external provider, independent of incoming request volume.

In a real system, the executor would typically be reused as a shared component.

```
1 // Concurrency control for LLM calls (bulkhead)
2 Semaphore llmSemaphore = new Semaphore(50);
3
4 try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {
5     Future<String> f = exec.submit(() -> {
6         // Keep the snippet focused; handle interruption based on your cancellation
6         // policy
7         llmSemaphore.acquireUninterruptibly();
8         try {
9             return llm.generate(prompt);
10        } finally {
11            llmSemaphore.release();
12        }
13    });
14
15    String result = f.get(); // propagate or handle execution failures
16 }
```

Watch for long native calls (old JDBC, C libraries). They can pin virtual threads to carriers. Migrate to modern drivers or isolate in platform threads.

Java virtual threads for AI concurrency require monitoring. Use JFR to detect pinning and bottlenecks. Active virtual thread metrics are essential in production.

Test with real load. Simulate 10k req/sec against your Vector DB + LLM. Measure p99 latency and memory usage. In internal benchmarks, I/O-bound workloads showed comparable or improved p99 latency with significantly reduced code complexity.

The Future of Java Concurrency for Intelligent Systems

Frameworks are adopting virtual threads rapidly. Spring Boot 3.2+ provides production-ready support for virtual threads. Quarkus and Micronaut already optimize for Loom. Helidon ships too.

Structured Concurrency prevents thread leaks and simplifies parallel AI pipelines. `StructuredTaskScope` creates a scoped boundary where all child tasks must complete (or be canceled) before the scope exits. This eliminates the classic *one task hangs forever* problem.

```

1 // Parallel embedding + cache lookup (both MUST complete)
2 try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
3     var embeddingTask = scope.fork(() -> embeddingService.generate(text));
4     var cacheTask = scope.fork(() -> cache.getSimilar(text));
5
6     scope.join(); // Blocks until ALL succeed OR ANY fails
7     scope.throwIfFailed(); // Propagates first failure
8
9     return llm.generate(embeddingTask.resultNow() + cacheTask.resultNow());
10 }

```

Key benefits for AI workloads:

- **Cancellation propagation:** `scope.shutdown()` cancels embedding *and* cache lookups instantly
- **Failure isolation:** One slow LLM call doesn't hang the entire request
- **Scoped lifecycle:** No orphaned tasks surviving endpoint completion
- **Timeout orchestration:** `scope.join(5, SECONDS)` prevents slow embeddings from blocking

Perfect for **multi-model fallback** (GPT-4 → Claude → Llama) and **parallel preprocessing** (embedding + metadata fetch).

Java AI libraries are growing. LangChain4j, Semantic Kernel, and Spring AI integrate virtual threads. Synchronous scalability becomes standard for intelligent APIs.

Java virtual threads for AI concurrency define the future. Less forced reactivity, more natural code. Java returns to synchronous roots with massive scaling power.

Conclusion: Simpler Concurrency for AI

Virtual threads transform AI APIs. Simple synchronous code scales to thousands of requests. This significantly reduces the cognitive overhead associated with deeply nested reactive pipelines.

Test it yourself. Build a simple RAG endpoint. Measure p99 latency at 10k req/sec. Compare virtual threads vs reactive. Empirical measurement remains essential. Evaluate under real load and observe latency and memory behavior.

Virtual threads redefine how Java handles concurrency in AI-driven systems. Java 26 delivers massive scalability with synchronous code clarity. Time to migrate your AI systems.

References

1. JEP 425: Virtual Threads (JFR) - Pinning monitoring
2. JEP 444: Virtual Threads - Official stabilization

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



From Spaghetti to Hexagons: A Practical Guide to Clean Java Architecture

Architecture and Microservices

HexagonalArchitecture

DDD

SOLID

Author: Enrique Medina Montenegro

Enrique Medina Montenegro is a Senior software architect and technical author specializing in Clean Architecture, Hexagonal Architecture, and Domain-Driven Design, with extensive experience building enterprise systems in Java and Spring.



LinkedIn

A Culinary Warning and the Dark Side of Code

Welcome, fellow architect and beloved developer, to our hexagonal city-state—a realm where pragmatism reigns supreme and flexibility is our greatest strength. This guide is not an ivory tower doctrine filled with theoretical perfection, but rather a versatile toolbox tempered in the forge of real-world challenges and countless production battlefields.

Before we can erect the citadel of clean architecture, we must first acknowledge the reality of our industry. We must navigate the “Dark Side of Code”. The software world is rife with culinary disasters that would make a chef weep. We are all too familiar with **Spaghetti Code**, a tangled mess where program flow twists and turns like pasta, making maintenance a nightmare where references and dependencies jump all over the place. But the menu of disaster is extensive.

We encounter **Spaghetti with Meatballs**, a chaotic fusion where object-oriented “meatballs” float aimlessly in a sea of procedural spaghetti, often the result of a lack of coding standards or language constraints. We face **Ravioli Code**, where the pursuit of modularity creates thousands of tiny, loosely coupled components that become impossible to track and reason about. And then there is **Lasagna Code**, the layered architecture gone wrong, where rigid tiers create a monolithic structure so fragile that a change in the database foundation ripples all the way up to the UI roof.

Beyond pasta, we battle behavioural anti-patterns. “**Hands in the Pants**” represents a violation of encapsulation, where developers bypass interfaces to grope the internal workings of a component. This often manifests as Anemic Domain Models—objects that are merely bags of getters and setters without behaviour.

Consider this Anemic Model, a classic example of the “Hands in the Pants” anti-pattern:

```
1 // Anemic domain model
2 public class Order {
3     private List<Item> items;
4     private double totalPrice;
5
6     public List<Item> getItems() { return items; }
7     public void setItems(final List<Item> items) { this.items = items; }
8     // ... getters and setters for totalPrice
9 }
10
11 // Usage: External logic manipulating internal state
12 Order order = new Order();
13 order.setItems(itemList);
14 double total = 0;
15 for (Item item : order.getItems()) {
16     total += item.getPrice();
17 }
18 order.setTotalPrice(total);
```

In this scenario, external code is reaching into the `Order` object to manipulate its state. To fix this, we must move toward a rich, hexagonal domain where we encapsulate behaviour:

```
1 public class Order {
2     private List<Item> items;
3
4     public void addItem(final Item item) {
5         items.add(item);
6     }
7     public double calculateTotalPrice() {
8         return items.stream().mapToDouble(Item::getPrice).sum();
9     }
10 }
```

We also face “**The One Liner**”, where code is compressed into unreadable puzzles, and “**Happy Path-Driven Development**”, where we optimistically ignore edge cases until production crashes. To escape this culinary hellscape and these pervasive code smells, we turn to the Hexagon.

Unveiling the Hexagon

The Hexagonal Architecture, also known as “Ports and Adapters,” is not just a pattern; it is a structural framework for achieving **Dependency Inversion**. It shares principles with Clean Architecture and Onion Architecture, emphasizing independence from frameworks, UIs, and databases. The core principle is simple yet profound: **Dependencies always point inward**.

The Spatial Perspective: Mapping the City-State

To truly understand this architecture, we must visualize it spatially, traveling from the center outwards:

1. **The Center (The Domain):** At the very heart lies the **Domain**, containing our Entities and Business Rules. This is the “what” and “why” of our application. It knows nothing of the outside world—no databases, no web requests, no frameworks.
2. **The Inner Ring (Application Services):** Surrounding the domain are the **Application Services**. These represent (implement) the Use Cases. They act as orchestrators, coordinating the flow of data and operations between the outside world and the domain core.
3. **The Edges (Ports):** The boundaries of our hexagon are defined by the **Ports**.
 - **Input Ports** (Left side) define the contracts for how the world talks to us (Use Cases).
 - **Output Ports** (Right side) define the contracts for what the application needs from the world (e.g., APIs, persistence, notifications).
4. **Beyond the Hexagon (Adapters):** Outside the walls lie the **Adapters**, the implementation details that bridge the gap to reality.
 - **Primary (Driving) Adapters** (e.g., REST Controllers) invoke input ports to drive the application.
 - **Secondary (Driven) Adapters** (e.g., JPA Repositories, SMTP clients) implement output ports to be driven by the application.

This structure ensures that the core remains isolated. We can swap a database or a UI framework without touching the business logic, achieving true modularity and testability.

Meet the Citizens (From Concept to Code)

Now that we have the blueprint, let us meet the citizens who inhabit our hexagonal city-state.

Input Ports: Gateways to the Core

Input Ports serve as the contract for our **Use Cases**. They define the specific interactions available to the outside world. A Use Case typically accepts a **Command** (for state-changing operations) or a **Query** (for data retrieval).

Crucially, these Commands and Queries should be self-validating POJOs that carry data, but no behavior. They represent the intent of the user.

Here is an example of a `PlaceOrderCommand` that enforces its own integrity upon creation:

```

1 public class PlaceOrderCommand {
2     private final String customerId;
3     private final List<OrderItemDTO> items;
4
5     public PlaceOrderCommand(final String customerId, final List<OrderItemDTO> items) {
6         this.customerId = Objects.requireNonNull(customerId, "Customer ID must not be
7             null");
8         this.items = Objects.requireNonNull(items, "Order items must not be null");
9         if (items.isEmpty()) {
10             throw new IllegalArgumentException("Order must contain at least one item");
11         }
12     }
13     // Getters...
14 }

```

To further decouple the execution of these commands, we can employ a **Command Bus**. Instead of injecting a specific service into a controller, the controller sends a command to a bus, which routes it to the appropriate handler. This pattern separates the intent (the Command) from the execution (the Handler).

The Application Layer: Orchestrating Business Processes

Application Services are the conductors of our architectural orchestra. They sit in the Application Layer and bridge the external actors and the internal domain. Their primary responsibility is **Orchestration**: they coordinate the flow of data, managing interactions between domain objects and output ports. They do *not* contain business rules; that is the domain's job.

However, real-world use cases often involve “secondary tasks” that accompany a primary operation. For instance, after signing up a user (primary task), we might need to send a confirmation email or update statistics. Implementing these directly in the service violates the *Single Responsibility Principle*.

The solution is **Domain Events**. By dispatching an event after the primary task, we can delegate secondary side effects to separate handlers.

Here is a refactored `SignupApplicationService` using Domain Events:

```

1 public class SignupApplicationService implements SignupUseCase {
2     private final UserRepository userRepository;
3     private final DomainEventDispatcher domainEventDispatcher;
4
5     public SignupApplicationService(final UserRepository userRepository, final
6         DomainEventDispatcher domainEventDispatcher) {
7         this.userRepository = userRepository;
8         this.domainEventDispatcher = domainEventDispatcher;
9     }
10
11     @Override
12     public void signup(final SignupCommand command) {
13         // 1. Check business rules (e.g., email uniqueness)
14         if (this.userRepository.existsByEmail(command.getEmail())) {
15             throw new IllegalArgumentException("Email already in use");
16         }
17         // 2. Create the user (Primary Task)
18         final User user = new User(command.getName(), command.getEmail());
19         this.userRepository.save(user);
20
21         // 3. Publish Domain Event to delegate 'secondary tasks'
22         this.domainEventDispatcher.dispatch(new UserSignedUpEvent(user));
23     }
24 }

```

Now, separate handlers can listen for `UserSignedUpEvent` to send emails or update profiles, keeping the service pure and adhering to the *Open/Closed Principle*.

The Domain Layer: The Heart of the Hexagon

We arrive at the core: the Domain Layer. Here, we must shift from an Anemic Model to a **Rich Domain Model**. Our domain must adhere to three functional principles: **Purity** (no side effects), **Encapsulation** (hiding internal state), and **Completeness** (handling all scenarios).

The domain is populated by specific building blocks:

- **Entities & Value Objects:** We should use Value Objects over primitives to prevent “Primitive Obsession”.
- **Domain Services:** When logic spans multiple aggregates (e.g., calculating a chess rating based on two players), use a Domain Service rather than forcing the logic into one entity.
- **Factories:** Use Factories to handle complex object creation, ensuring that objects are never instantiated in an invalid state. They are the guardians of domain integrity at the point of creation.

Output Ports: The Diplomatic Corps

Output Ports are the channels through which our city-state communicates with the outside world. They are **Role Interfaces**, defined by looking at the specific interaction required by the domain, not by the implementation details of the external tool.

A common pitfall is defining “Header Interfaces” that simply mimic the public methods of an implementation. Instead, we should define interfaces based on the *role* they play for the application.

Repositories: The Memory Keepers

Repositories are a special type of Output Port. While they manage persistence, they belong to the domain’s conceptual world. A modern best practice in this architecture is **Interface Slicing**. Instead of creating one giant, generic repository that exposes every CRUD operation, we should create fine-grained interfaces for specific needs.

Consider this Sliced Repository example:

```

1 // Fine-grained interfaces in the Domain/Application layer
2 public interface CreateCustomerOutputPort {
3     void save(final Customer customer);
4 }
5
6 public interface FindCustomersOutputPort {
7     Optional<Customer> findById(final CustomerId id);
8     List<Customer> findAll();
9 }
10
11 // Implementation in the Adapter layer
12 @Repository
13 @RequiredArgsConstructor
14 public class JpaCustomerRepository implements CreateCustomerOutputPort,
15     FindCustomersOutputPort {
16     private final SpringCustomerRepository springCustomerRepository; // Internal Spring
17     private final CustomerMapper customerMapper;
18
19     @Override
20     public void save(final Customer customer) {
21         final JpaCustomer jpaCustomer = this.customerMapper.convertToJpaEntity(customer);
22         this.springCustomerRepository.save(jpaCustomer);
23     }
24     // ... implementation of findAll
25 }
```

This approach prevents “Leaky Abstractions” where database-specific methods pollute the domain and adheres to the Interface Segregation Principle.

The Invisible Pillars (Technical Concerns)

Having built the visible structure, we must now attend to the invisible pillars that support our architecture: Data Flow, Control Flow, Error Handling, and Validation.

Data Flow and Mapping Strategies

How does data cross the boundaries of our hexagon? We generally identify three strategies for mapping data between layers:

1. **Bypass Mapping:** The Domain object is used everywhere (Controller -> Service -> Repository). This is simple, but creates tight coupling and leaky abstractions. It is a path of least resistance that often leads to technical debt.
2. **Bridge Mapping:** Adapters have their own models, but map them to the Domain object before passing them to the core. This is better, but the Domain is still exposed to the outer layers.
3. **Barrier Mapping (Recommended):** Full separation. Input adapters map Requests to Commands. Services map Commands to Domain objects. Output ports accept DTOs or Persistence Entities mapped from Domain objects. This creates a “firewall” around the core, ensuring true separation of concerns.

The golden rule of Data Types is: **Boundary Types** (Requests, DTOs) and **Adapter Types** (JPA Entities) must *never* leak into the **Domain Types**.

Control Flow: Orchestrating the Symphony

Control Flow represents the heartbeat of our system. We must distinguish between **Coordination** (managing interactions within a use case) and **Orchestration** (managing complex workflows across the system).

We can also leverage **CQS (Command Query Separation)**. Methods should either change state (Command) or return data (Query), but not both. Furthermore, **CQRS (Command Query Responsibility Segregation)** allows us to split the system into Write models (handled by the Domain) and Read models (Queries potentially bypassing the domain for performance), optimizing each for its specific task.

Error Handling: The Either Revolution

In the Java world, we have an addiction to Exceptions. But Exceptions act like a “GOTO” statement, disrupting control flow and violating the Principle of Least Astonishment. Why should a “User Not Found” scenario—a perfectly valid business outcome—cause the program execution to jump unexpectedly?

Instead of throwing exceptions for business errors, we should treat errors as **values**. We use the **Either<L, R>** monad (often from libraries like Vavr). The **Left** side holds the error (Business or Technical), and the **Right** side holds the success value.

Consider this refactoring from Exceptions to **Either**:

Traditional (Bad):

```
1 // Opaque signature, relies on hidden exceptions
2 public List<DossierDTO> fetchPortfolio(String user) {
3     // ... checks
4     if (portfolio == null) throw new PortfolioNotFoundException();
5     return portfolio;
6 }
```

Error Values (Good):

```

1 // Explicit signature declaring the error possibility
2 public Either<ApplicationError, List<DossierDTO>> fetchPortfolio(final String user) {
3     try {
4         List<DossierDTO> portfolio = externalService.get(user);
5         if (portfolio == null) {
6             return Either.left(new PortfolioNotFoundError(user));
7         }
8         return Either.right(portfolio);
9     } catch (RestClientException e) {
10        return Either.left(new PortfolioServiceNotAvailableError(e));
11    }
12 }

```

This forces the consumer to handle the error explicitly, leading to safer and more predictable code.

Validations: The Always-Valid Domain Model

We must distinguish between **Syntactic Validation** (checking input formats like email regex) and **Semantic Validation** (checking business invariants like “email must be unique”).

To enforce the **Always-Valid Domain Model**, an object should never exist in an invalid state. The “isValid()” method approach is flawed because it implies the object was created in an invalid state first. Instead, we use **Static Factory Methods** to validate *before* instantiation.

```

1 @RequiredArgsConstructor(access = AccessLevel.PRIVATE)
2 public class User {
3     private final String email;
4
5     // Factory method returning a Validation object (Vavr)
6     public static Validation<List<DomainError>, User> validateThenCreate(final String
7         email) {
8         return isValidDomainEmail(email)
9             ? Validation.valid(new User(email))
10            : Validation.invalid(List.of(new DomainError("Invalid email domain")));
11    }
12 }

```

This ensures that if you hold a reference to a `User` object, it is guaranteed to be valid.

Transactionality: A Framework-Agnostic Output Port

Transactionality is crucial for data integrity, but we should not pollute our domain with framework-specific annotations like Spring’s `@Transactional`. Instead, we define transactionality as an Output Port using a custom `@Transactional` annotation (in a Shared Kernel) and use an Adapter (via Spring AOP) to map it to the actual transaction manager at runtime.

```

1 @Target({ElementType.METHOD, ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface Transactional {
4     // Custom properties for rollback rules with Either
5     Class<?>[] rollbackForWithEither() default {};
6 }

```

This keeps our Application Core completely framework-agnostic.

Compliance and Fulfillment

How do we ensure our code actually reflects these principles? We need a strategy for code organization and automated enforcement.

Architectural Cartography: Blueprinting the Code

The structure of your code should “scream” its architecture. We should avoid organizing by “kind” (putting all services in one `services` package, all controllers in `controllers`). Instead, we use an **Architecturally Expressive** strategy that mirrors the hexagonal structure.

A recommended multi-module Maven structure looks like this:

- **hexagonal-ref-app (Root)**
 - **shared-kernel**: Common DTOs, Errors, Domain Events, Validations.
 - **application-core**: The Hexagon itself.
 - * **domain**: Entities, Value Objects.
 - * **input-ports**: Use Cases interfaces, Commands/Queries.
 - * **application**: Handlers/Services implementation.
 - * **output-ports**: Interfaces for external needs.
 - **adapters**:
 - * **api-adapter**: REST Controllers (Primary Adapter).
 - * **persistence-adapter**: JPA Repositories (Secondary Adapter).
 - **spring-boot-assembly**: The execution entry point.

Crucially, we use Java’s **package-private** visibility to enforce boundaries. Only Ports (Interfaces) and DTOs should be public. The implementations (like our previous example of `CreateArticleHandler`) should be hidden from the rest of the world, accessible only via dependency injection.

ArchUnit: The Guardian of Integrity

We cannot rely solely on developer discipline. We use **ArchUnit** to automate our architectural rules. It acts as a tireless sentinel.

We can define rules such as “Adapters must not depend on other Adapters” or “Domain must not depend on Frameworks.”

Example Rule (Command Checker):

```
1 final ArchRule commandDependencyRule = classes()
2   .that().resideInAnyPackage("..application.command..")
3   .should().onlyDependOnClassesThat()
4   .resideInAnyPackage("java..", "..shared..", "..application.command..");
```

This ensures Commands remain pure and don’t depend on the Web or Persistence layers. By using `FreezingArchRule`, we can apply these rules to legacy projects, “freezing” current violations while preventing new ones, allowing for gradual improvement.

Testing Strategy

Testing in a Hexagonal Architecture is simplified by the strict separation of concerns. We employ specific typologies to target each layer effectively.

Unit Testing Static Factory Methods

We test the domain logic in complete isolation. Since our objects are “Always Valid,” we must verify that the factories reject invalid input.

```
1 @ParameterizedTest
2 @MethodSource("invalidArguments")
3 void validateThenCreateShouldReturnError(String email) {
4     Validation<Error, User> result = User.validateThenCreate(email);
5     assertThat(result.isInvalid()).isTrue();
6 }
```

Unit Testing Handlers

Handlers are orchestrators. To test them, we use **Mockito** to mock the Output Ports. We are not testing the database; we are testing that the handler *calls* the database port correctly.

```
1 @ExtendWith(MockitoExtension.class)
2 class DownloadArticleHandlerTest {
3     @Mock DownloadArticleOutputPort outputPort;
4     @InjectMocks DownloadArticleHandler handler;
5
6     @Test
7     void handleShouldReturnContent() {
8         // given
9         given(outputPort.download(any(), any())).willReturn(Either.right(content));
10        // when
11        var result = handler.handle(query);
12        // then
13        assertThat(result.isRight()).isTrue();
14        verify(outputPort).download(any(), any());
15    }
16 }
```

This ensures the orchestration logic is correct without needing a real database connection.

Unit Testing Adapters

Adapters are heterogeneous and require specific strategies.

- **REST Controllers:** We can test that they correctly map HTTP requests to Commands and DTOs to Responses.
- **External APIs:** We avoid making real network calls. We can use `MockRestServiceServer` (if using Spring’s Rest-Client) or `MockedConstruction` (if using generated OpenAPI clients) to simulate API behavior.

Testing a Controller with Mockito:

```
1 @Test
2 void profileShouldReturnResponse() {
3     // given
4     given(queryBus.query(any())).willReturn(Either.right(userProfile));
5     // when
6     ResponseEntity<?> response = controller.profile(auth, request);
7     // then
8     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
9 }
```

From Blueprint to Reality (Reference Application)

To cement these concepts, let us dissect the “Create Article” flow in our reference application, a fully-fledged Hexagonal Masterpiece.

The Input Port (Use Case)

Located in `application-core/input-ports`. It extends `CommandHandler` and uses a self-validating `Command`. This defines the contract.

```

1 @UseCase
2 public interface CreateArticleUseCase extends CommandHandler<CreateArticleCommand> {}
3
4 @Getter
5 @RequiredArgsConstructor(access = AccessLevel.PRIVATE)
6 public class CreateArticleCommand implements Command {
7     // fields...
8     public static Validation<Error, CreateArticleCommand> validateThenCreate(...) { ... }
9 }

```

The Application Service (Handler)

Located in `application-core/application`. It orchestrates the flow.

```

1 @ApplicationService
2 @RequiredArgsConstructor
3 final class CreateArticleHandler implements CreateArticleUseCase {
4     private final AuthorOutputPort authorOutputPort;
5     private final ArticleRepository articleRepository;
6
7     @Override
8     @Transactional
9     public Either<Error, Void> handle(final CreateArticleCommand command) {
10         return this.authorOutputPort.lookupAuthor(command.authorId())
11             .flatMap(author -> ArticleMapper.INSTANCE.toArticle(command, author).toEither())
12             .flatMap(this.articleRepository::save);
13     }
14 }

```

Notice the use of `flatMap` to chain operations. If any step fails (returns `Either.Left`), the chain stops, and the error propagates. If it succeeds, it flows to the next step.

The Domain Entity

Located in `application-core/domain`. It uses the Factory Method pattern to ensure validity upon creation.

```

1 public class Article {
2     public static Validation<Error, Article> validateThenCreate(...) {
3         // validations...
4         return Validation.valid(new Article(...));
5     }
6 }

```

The Output Port

Located in `application-core/output-ports`.

```

1 @OutputPort
2 public interface AuthorOutputPort {
3     Either<Error, AuthorDTO> lookupAuthor(final String id);
4 }

```

The Adapter (External API)

Located in `author-external-adapter`. It implements the port and handles the gritty details of the REST call.

```

1  @Adapter
2  @RequiredArgsConstructor
3  final class AuthorExternalAPIAdapter implements AuthorOutputPort {
4      private final RestClient restClient;
5      @Override
6      public Either<Error, AuthorDTO> lookupAuthor(String id) {
7          // Uses Try to capture exceptions from the REST call and convert them to Error
8          // values
9          // Returns Either.right(dto) or Either.left(error)
10     }
11 }
```

The Assembly

Finally, in `spring-boot-assembly`, we use `@ComponentScan` with our custom filters to wire everything together. This is where the magic happens, turning our isolated components into a running application.

We use **Dependency Injection (DI)** to wire our components together. To make our code even more expressive, we define custom annotations that serve as architectural signposts: `@Adapter`, `@ApplicationService`, and `@DomainService`.

```

1  @ComponentScan(basePackages = "com.emedina", includeFilters = @ComponentScan.Filter(
2      type = FilterType.ANNOTATION,
3      classes = {Adapter.class, ApplicationService.class, DomainService.class}
4  ))
```

This tells Spring exactly what to look for, turning our DI configuration into a semantic map of our architecture.

Conclusion: Hexagonal Horizons

We have traversed a vast landscape, moving from the messy reality of “Spaghetti Code” and “Lasagna Architecture” to the structured elegance of the Hexagonal Citadel. We have embarked on a quest to fulfill the promises of our Hexagonal Architecture, deliberately steering clear of the siren song of implementation details.

This journey has been more than a technical exercise; it has been a manifesto for clean, maintainable, and expressive code. We have witnessed how a codebase can scream its purpose and semantics. Imagine a bug cropping up in article updates. In a traditional application, this might spark a wild goose chase through sprawling service classes. But in our hexagonal city-state, the path is clear. We would swiftly navigate to the `input-ports` module, then trace our steps left to the `api-adapter` and right through `application-core` to `output-ports`—no hesitation, no false leads.

We have embraced error values over exceptions, made the implicit explicit, and decomposed concerns into modules that tell a story. Most crucially, we have fortified our domain, that precious core of business logic, against the shifting sands of external frameworks and technologies.

This clarity is about empowering developers. It is about creating systems that are built to evolve. Whether you are refactoring an existing Goliath or crafting a new David, the principles we have explored provide a robust foundation. Our Hexagonal Architecture is not just about drawing boundaries; it is about creating a living, breathing ecosystem where business logic thrives.

So, dear architect, as you stand at the threshold of your next project, armed with the insights from this journey, embrace the hexagonal way. Let your code sing its purpose, let your architecture breathe flexibility, and let your creative spirit soar in this newfound architectural freedom. The hexagonal revolution awaits—are you ready to lead the change?

This article tries to summarize the much more in-depth and comprehensive exploration of the Hexagonal Architecture covered in the book “Decoupling By Design”



Do You Still Write Code without Hexagon?

Architecture and Microservices

Hexagon

PortsAndAdapters

CleanArchitecture

Authors: Marek Dominiak and Vadzim Prudnikau

Marek Dominiak is a CTO of a software house, a hands-on architect, and a Trainer with 18 years of experience mainly in the Java world. He has spent a lot of time mentoring developers and leading teams, with a big focus on creating sustainable solutions - systems that will last for many years.



LinkedIn

Vadzim Prudnikau is a Hands-on Architect, IT Trainer, and Conference Speaker with 20+ years of experience, passionate about building high-quality software, mentoring developers, and sharing knowledge. He has trained over 550 professionals in Architecture, DDD, EventStorming, and TDD, built and deployed more than 10 production-ready projects, led multiple development teams, and created a successful Java department from scratch. Along the way, he has mentored 20+ developers and conducted 100+ technical interviews. His expertise lies in Java, but he also has experience with Kotlin, Groovy, and other programming languages. He believes that strong technical skills, continuous learning, teamwork, and knowledge-sharing are the keys to success in software development.



LinkedIn

During our training in architecture, we often ask participants if they are familiar with Hexagonal architecture, and it's surprising to see only a few hands raised initially. After a brief presentation, we see a few more hands go up, as participants realize they have been using Hexagonal architecture without realizing it.

As hands-on architects, we rely heavily on Hexagonal architecture in our daily work. However, we notice that many people are still unaware of the significant benefits it can bring to their projects.

Let's find out together the reasons why you should consider adopting Hexagonal architecture today.

Why is it important to consider it?

Before diving into Hexagonal architecture, let's understand the following concepts:

1. **Application code:** This refers to business logic focusing on the core functionality of the software. That can be tax calculation rules, a process of user registration, and so on.
2. **Infrastructure code:** This deals with specific technologies that support the application code. Essentially, any SQL query, an HTTP request, an SMTP call, or anything related to a specific technology fits into this category.
3. **Separation of concerns:** An established principle in programming that states that distinct aspects of a software system should be separated and modularized to promote maintainability, readability, and reusability.

Separating application code and infrastructure code is important for several reasons:

- **First, it simplifies testing.** Let's consider an application used for tax calculation that needs to retrieve some data to provide the proper tax rate from the database. When application and infrastructure code are separated, creating straightforward automated tests for tax calculation rules becomes much easier. However, if those were tied together, the database must be running along with the tests, making them significantly slower.
- **Second, it enhances flexibility.** Consider a real-life example where a team needed to replace one messaging system with another (Apache Artemis -> Apache ActiveMQ). By keeping the messaging-specific code separate from the application code, the transition could take much less time, and the core functionality remained unchanged. Had the messaging system been tightly integrated with the application code, this would have been a much more challenging task.
- **Third, it promotes reusability.** Consider what happens when a development team is asked to support a new type of client for the application – for instance, in addition to HTTP over REST, we now want to support SOAP. If the application code was mixed with the infrastructure code (specific to HTTP over REST), it would be difficult to accommodate this change. However, if we kept them separate, we could add new code supporting SOAP while reusing all the existing application code.
- **Additionally,** the separation of concerns allows for **better maintainability**, as it becomes easier to identify and fix issues without affecting other parts of the system.

In conclusion, keeping application and infrastructure code separate is crucial for effective software development.

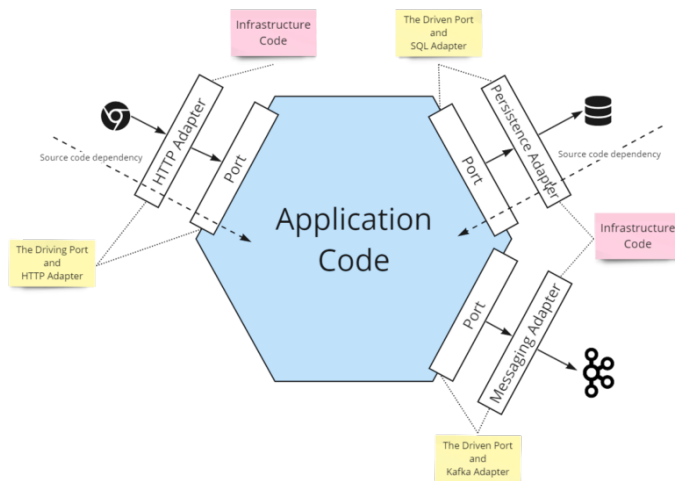
Hexagonal architecture is a solution that promotes this separation, ensuring maintainability, flexibility, testing, and high cohesion, so read on.

What is Hexagonal Architecture?

Hexagonal architecture, also known as Ports and Adapters, is an architectural pattern that enables the separation of business logic from infrastructure-related aspects such as HTTP, databases, file systems, and others.

You will soon discover that Hexagonal architecture is not a revolutionary new concept, but rather a well-established principle closely aligned with other patterns such as Clean Architecture, Dependency Inversion, and general Object Oriented Programming (OOP) principles.

The following picture (among many available on the internet) illustrates the main idea:



The picture above illustrates a distinct separation between Ports and Adapters, with the Application Code being only aware of the Ports.

A **Port** is a technology-agnostic interface in your code. The **Driving Port** allows inputs (for example, an HTTP call) to come to the Application Code, while the **Driven Port** enables the Application Code to work with the outputs (such as saving data to a database or sending a message to Kafka).

On the other hand, an **Adapter** is a technology-specific implementation that allows interaction with real devices like databases, message brokers, browsers, SMTP servers, and so on. Adapters depend on Ports and have no direct relationship with the Application Code in your system.

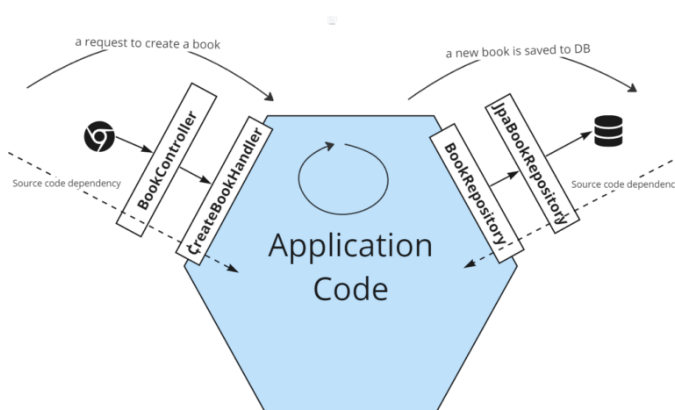
By the way, REST Controllers are an example of an HTTP Adapter, therefore they should never contain any business logic.

In the next section, we will examine the same picture accompanied by a code example.

Example

In the example, you will see a conceptual picture and a folder structure that you might have in your project, as well as the implementation. However, please note that you may need to adapt it to your specific needs. For example, you may need to add common classes, and additional attributes, or make other modifications since this example is simplified for clarity and is agnostic of the technical stack.

The following picture is similar to the one you have already seen, but it shows specific implementations for Ports and Adapters:



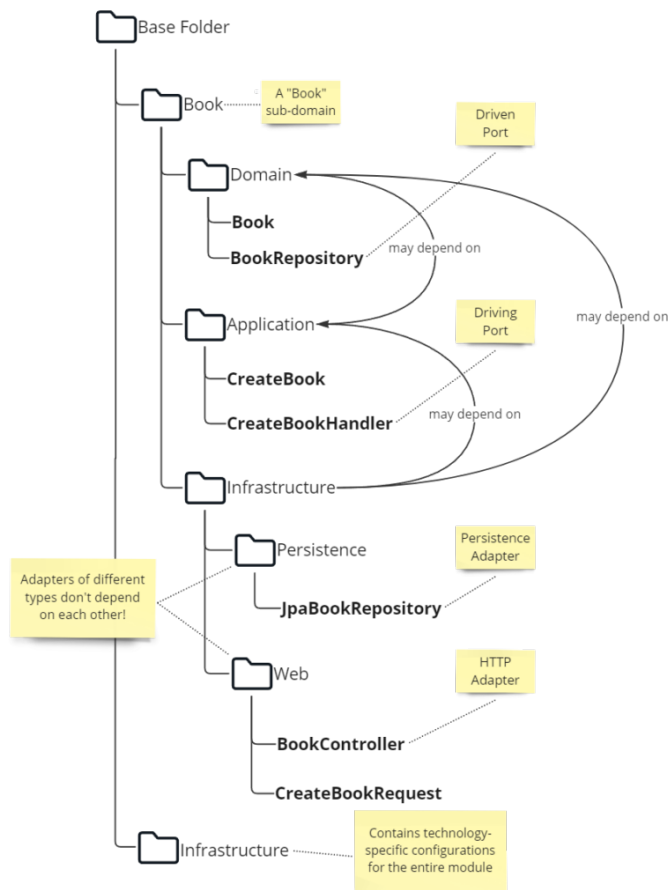
BookController is an HTTP Adapter.

CreateBookHandler is a Driving Port.

BookRepository is a Driven Port.

JPABookRepository is a Persistence Adapter (JPA stands for Java Persistence API).

Now, let's take a look at a possible folder structure:



The picture above focuses on the module structure and dependencies between components. As you can see, the classes within the **Application** and **Domain** folders are unaware of adapters and are only responsible for business use cases.

One important note: adapters of different types should not depend on each other! Otherwise, if you decide to replace, for instance, the persistence adapter, you would also need to modify the HTTP adapter.

Finally, let's take a look at a possible implementation.

The **BookController** adapter accepts an HTTP request and finds a port (**CreateBookHandler**) that can handle the request:

```

1 @RestController
2 public class BookController {
3     private final CreateBookHandler createBookHandler;
4     @PostMapping
5     public ResponseEntity<String> create(CreateBookRequest request) {
6         var command = new CreateBook(request.title(), request.author());
7         var id = createBookHandler.handle(command);
8         return ResponseEntity.created(
9             URI.create("/api/books/%s".formatted(id))
10            .build());
11     }
12 }

```

The **CreateBookHandler** handles the business use case by creating a new book and saving it using the **BookRepository** port:

```

1 public class CreateBookHandler {
2     private final BookRepository repository;
3     public UUID handle(CreateBook command) {
4         var book = new Book(command.title(), command.author());
5         return repository.save(book).getId();
6     }
7 }

```

BookRepository is just an interface:

```
1 public interface BookRepository {
2     Book save(Book book);
3     Optional<Book> find(UUID id);
4 }
```

Its database-specific implementation (**JpaBookRepository**) resides under the **infrastructure/persistence** folder, completely separate from the application code:

```
1 public class JpaBookRepository implements BookRepository {
2     private final EntityManager entityManager;
3     @Override
4     public Book save(Book book) {
5         entityManager.persist(book);
6         return book;
7     }
8     ...
9 }
```

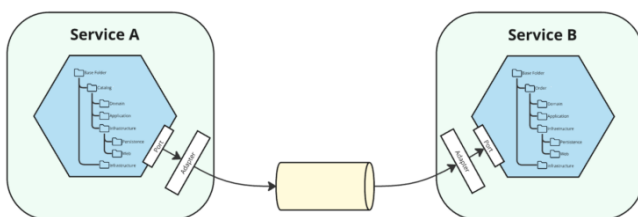
The presented example scales very well in most projects, and helps achieve high-quality testable and replaceable code!

Hexagonal Architecture in the Microservices World

Even though Hexagonal Architecture has “architecture” in its name, we would argue that it isn’t architecture *per se*, but rather a useful tactical pattern. It focuses only on structuring and organizing code and components inside a specific service.

Take a look at the picture below. We have two hexagons that communicate through a messaging system. Hexagonal Architecture doesn’t address the high-level picture: it doesn’t suggest how to communicate, nor does it define the structure of systems consisting of multiple services. So, we like to think about hexagons as being applicable inside specific services, not as a high-level architecture.

Note: In this article, we will stick to the name Hexagonal Architecture (or Hexagon) as it’s already “embedded” in the software development vocabulary.



Should we use it everywhere?

As with most software development patterns or methodologies, it’s impossible to say that one can be used universally, as there are always caveats and exceptions. So, when is Hexagonal architecture a good fit?

As a rule of thumb, consider Hexagonal architecture as a good choice for:

1. **Long-lasting applications that need to be maintained for an extended period.** Maintainability costs tend to rise over time in non-hexagonal systems.
2. **Projects that require the flexibility to test new technologies** (such as databases, messaging systems, or cloud providers).
3. **Larger and more complex projects**, which are more likely to benefit from Hexagonal architecture.
4. **Projects where testing is essential**, and you want to test the application and domain logic separately from the infrastructure code.

However, a Hexagonal architecture may not be advised in certain situations:

1. **A fairly simple project** (sometimes referred to as CRUD). In these cases, a slightly less elegant solution may still provide good value for the efforts spent on implementation.
2. **A quick prototype is needed to secure a client for building a more extensive system** (and when it's won, then consider using Hexagonal architecture).
3. **A framework-specific library or system is being developed**, where you want to tie yourself to the infrastructure-specific code to utilize it fully.

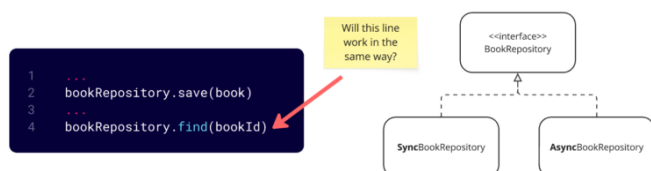
As you can probably guess, many projects written in mainstream languages fit into the first list, suggesting that you should consider using Hexagonal architecture.

Illusions of Hexagonal Architecture

Illusion number 1: “I can easily switch between sync/async ways of communication with my infrastructure”

This must be the biggest illusion (or trap) software engineers fall into when applying Hexagonal Architecture in their projects. With good intentions, developers (ourselves included) attempt to create perfectly reusable interfaces that hide away the details of whether an operation is asynchronous or synchronous. This is a mistake.

Consider this example:

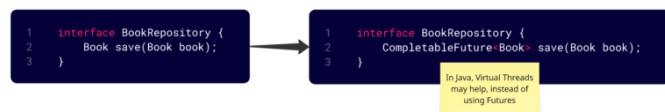


The question is whether we can create the **BookRepository** interface and hide the asynchronicity of the operation behind the Adapter.

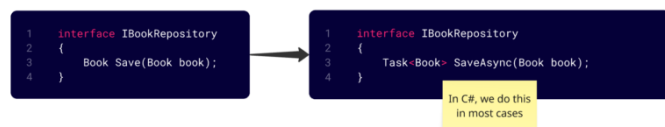
In the case of an **AsyncBookRepository** adapter, we might not be sure if, just after saving the book, we are guaranteed to “find” it in the next operations. The **SyncBookRepository**, however, will return the book as we expected. In both options, the observable results can differ.

A much better approach is to expose the asynchronicity of the operation at the Port level. That is why in many frameworks or libraries, you can find both synchronous and asynchronous methods for the same operations.

Java



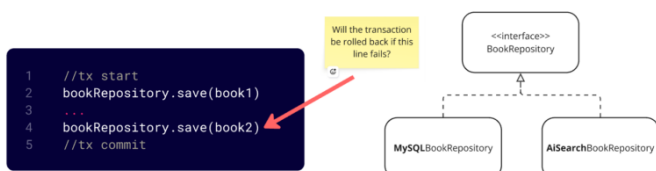
C#



Conclusion: Prefer exposing async support on the port level.

Illusion number 2: “I can easily switch to infrastructure with different ACID properties”

Consider the scenario presented in the image below:



Depending on our Adapter implementation, we might get different results regarding transaction boundaries.

- In the case of MySQL (or any other datastore offering ACID properties), both books will be saved if the transaction succeeds, or neither will be saved in the case of a rollback.
- If we choose to switch to another Adapter – e.g., based on ElasticSearch (or similar tools like OpenSearch or Azure AI Search)
- we might get different behavior, such as only one book being saved if a failure occurs between the two operations.

We need to be aware of the guarantees of our datastores and design transaction boundaries accordingly.

Conclusion: The main idea is to separate application logic from infrastructure code, rather than to seek a silver bullet for universal architecture.

Illusion number 3: `**_` “I can’t have library-specific annotations/attributes on the domain classes”.

Consider the following piece of Java code that uses the Java Persistence API:

```
1 @Entity
2 public class UserAccount extends UuidAggregateRoot {
```

When applying Hexagonal Architecture, many developers attempt to follow it to the letter. If we do that, we might ban our persistence framework annotations from our Application code, arguing that they should be hidden behind an Adapter implementation. Following this path has a high price to pay; there is a lot of additional complexity and work needed to achieve that, such as adding additional abstraction layers, duplicating many classes, providing additional mappings and translators, ensuring proper test coverage, and providing utility classes.

We suggest being pragmatic and asking yourself these questions when making this decision:

- How often do you change a chosen persistence framework?
- Do annotations/attributes help you become faster or less effective?
- Can you test your code without running any infrastructure?
- Do you clearly see the value of being a purist?

Conclusion: Hexagonal Architecture is a tool, and it’s your responsibility to apply it in the most effective way to meet your needs.

An ‘Important’ Question: “Why Does It Have 6 Sides?”

There is no direct answer to this question. Here’s what we think:

- Circles, boxes, and rectangles are commonly represented in various contexts.
- Drawing 5 sides can be more challenging.
- Having more than 6 sides would be considered “too much”.
- Finally, it doesn’t matter much; it’s only a name.

How Is It Different from Clean and Onion Architectures?

Hexagonal, Clean, and Onion architectures share the core principle of organizing code for maintainability, flexibility, and testability, representing fundamental design concepts, such as the Dependency Inversion Principle (code should depend on abstractions rather than on implementations). Hexagonal architecture focuses on separating business logic from infrastructure code through ports and adapters, making it less generic and more straightforward to explain.

In contrast, Clean and Onion architectures emphasize the separation of concerns through concentric circles or layers, with core business logic at the center. All three architectures promote the same fundamental OOP concepts, including Dependency Inversion, and should become the default way of thinking about code organization. Despite their differences, each pattern is a valuable tool for creating clean, maintainable, and flexible code.

Conclusion

If, after reading this article, you think that the principle is very simple and you understand it, you might be right. However, don't be fooled into thinking it will be easy to follow in real projects. Even though team members remember the rule, time after time, we find our infrastructure code "leaking" into application code, making testing and maintenance more challenging. Additionally, the entire team must be taught Hexagonal architecture rules and apply them consistently in their daily work. Last but not least, don't fall into the illusions of Hexagonal Architecture.

Overall, we should view Hexagonal Architecture as a powerful tool in our toolbox, applying it in cases where we need to ensure flexibility, testability, reusability, and maintainability.

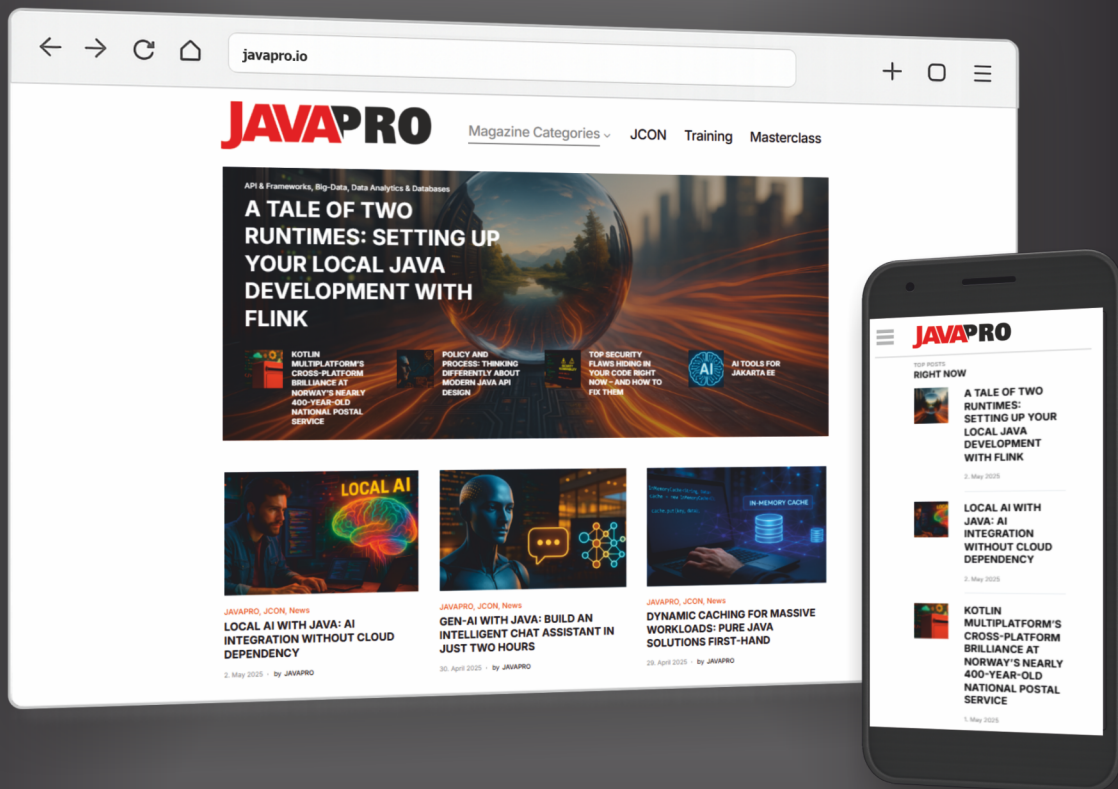
› **Back to the Table of Contents** ‹

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



From ‘Class Not Found’ to Calm: Practical Shading and Shadowing for Java Apps

DevOps

Maven

CoreJava

Author: Kamil Banach

Software engineer and team leader with more than a decade of experience in developing and maintaining web applications, with a particular focus on applications written in the Java ecosystem. Neither afraid of working on code, even on code that is already “legacy”, nor of designing CI/CD processes that streamline development and deployment. Enthusiast of clean, straightforward code that solves business problems effectively, prioritizing readability and simplicity.



LinkedIn

Your pipeline is green, the app is deployed, and everything looks fine... until production says “nope”. A request fails and the logs show `ClassNotFoundException`, or worse, `NoSuchMethodError`. These are among the most confusing runtime errors because the code compiled successfully, but the JVM is acting differently than you would expect.

Let's dive in and explore how the class loader works, what class shadowing is (when one class silently wins over another on the classpath) and what class shading is (when you deliberately relocate packages so multiple versions can coexist). We'll also see how these two concepts can help solve mysterious production issues and make your app more stable and secure.

The code used in the examples is available on GitHub: [elwin013/class-shading-and-shadowing-examples](https://github.com/elwin013/class-shading-and-shadowing-examples). The examples uses modern Java (17+) and Maven - it might be worth cloning the repository and experimenting locally with the code to see how it works in practice.

The Class Loader and Classpath

The heart of class loading is the `ClassLoader`. At runtime, it turns a fully qualified name (FQN, the class name with its corresponding package) such as `com.example.Claazz` into an actual class that the JVM can execute. Two different packages can safely contain classes with the same simple name, because the package is part of the FQN.

A given `ClassLoader` defines a class at most once. If the same class is requested again, the loader returns the already-defined `Class` object instead of re-reading it from disk.

We often say “the class loader”, but in fact the JVM uses multiple class loaders. In a typical application we can distinguish at least three:

- **Bootstrap** - loads core Java runtime classes (it is implemented in native code, so `getClassLoader()` returns `null`).
- **Platform** - loads standard platform modules from the JDK.
- **Application** - loads your application classes and dependencies from the classpath.

There can be more, for example in application servers there is often an additional class loader per web application. In Tomcat, this is called `WebappClassLoader` and it is used to isolate each web application.

Because a class is effectively identified by (`ClassLoader` + FQN), the same FQN can exist more than once in a single JVM if it is loaded by different class loaders. Examples are aforementioned application servers and plugin-based apps.

We can inspect class loaders by calling `getClassLoader()` on a class, as shown below. We can see that the `Instant` class is loaded by the bootstrap class loader, because the loader is shown as `null`.

```
1 public class App {
2     public static void main(String[] args) {
3         System.out.println("net.codeer.example4.App class loader: " + App.class.
4             getClassLoader());
5         System.out.println("java.sql.Date loader: " + Date.class.getClassLoader());
6         System.out.println("java.time.Instant class loader: " + Instant.class.
7             getClassLoader());
8     }
9 }
```

When we run this code, we will see the following output:

```
1 net.codeer.example4.App class loader: jdk.internal.loader.
   ClassLoaders$AppClassLoader@1dbd16a6
2 java.sql.Date loader: jdk.internal.loader.ClassLoaders$PlatformClassLoader@1d81eb93
3 java.time.Instant class loader: null
```

Class Loader Hierarchy

Class loaders are ‘lazy’ in the sense that they load classes on demand. By default they follow the parent delegation model and delegate class loading to the parent whenever possible.

When the application wants to load a class like `com.example.Foo`, the `AppClassLoader` first checks whether it has already defined that class. If not, it asks its parent (`PlatformClassLoader`). If the platform loader does not find it, it delegates again to its parent (`BootstrapClassLoader`).

If none of the parents can provide it, the `AppClassLoader` then attempts to load it from the classpath (or from known modules). If it still cannot find it, the JVM throws the infamous `ClassNotFoundException`.

Even though parent delegation is the default, it can be customized. For example, an application server can use custom class loaders to isolate applications and ensure the expected versions are loaded at runtime.

Class Shadowing and Dependency Hell

So, what is class shadowing? It is similar (as an idea) to variable shadowing - if the same fully qualified class name (FQN) is available in more than one place, one definition “wins” and the other one is effectively ignored. For example, if a JAR on your app classpath contains `com.exampleClazz` and you also create a class `com.exampleClazz` in your application, your application class will shadow the version from that other JAR.

Build tools typically resolve version conflicts by selecting a single version of a library for the final runtime classpath. In Maven, version conflict resolution is based on the “nearest wins” rule in the dependency graph; declaration order can matter in some tie cases. If your dependencies require different versions of a common dependency, the chosen version can surprise you and you can end up with “dependency hell”.

Assume the following example - the app uses two modules (`example5-module` and `example5-secondmodule`) that both depend on Gson, but in different versions:

- `example5-module` uses Gson 2.7.
- `example5-secondmodule` uses Gson 2.10.1, and it uses the `disableJdkUnsafe` method, which is not available in 2.7.

The application code looks like this (excluding imports):

```
1 public static void main(String[] args) {
2     var helloWorld = new HelloWorld("Kamil", 2023);
3
4     var serializer = new NormalSerializer(); // from example5-module
5     System.out.println(serializer.serialize(helloWorld));
6     System.out.println(serializer.getSerializerPackage());
7
8     var serializer2 = new FancySerializer(); // from example5-secondmodule
9     System.out.println(serializer2.serialize(helloWorld));
10    System.out.println(serializer2.getSerializerPackage());
11 }
```

And the dependencies in the `pom.xml` are defined as follows:

```
1 <dependencies>
2     <dependency>
3         <groupId>net.codeer</groupId>
4         <artifactId>example5-module</artifactId>
5         <version>1.0-SNAPSHOT</version>
6     </dependency>
7     <dependency>
8         <groupId>net.codeer</groupId>
9         <artifactId>example5-secondmodule</artifactId>
10        <version>1.0-SNAPSHOT</version>
11    </dependency>
12 </dependencies>
```

At compile time everything is fine, but `FancySerializer` expects a newer Gson, while on the classpath the older one is found first. That results in the infamous `java.lang.NoSuchMethodError`.

We can see which version Maven selected by running `mvn dependency:tree`:


```

1 [INFO] --- dependency:3.7.0:tree (default-cli) @ example5-app ---
2 [INFO] net.codeer:example5-app:jar:1.0-SNAPSHOT
3 [INFO] +- net.codeer:example5-module:jar:1.0-SNAPSHOT:compile
4 [INFO] | \- com.google.code.gson:gson:jar:2.7:compile
5 [INFO] \- net.codeer:example5-secondmodule:jar:1.0-SNAPSHOT:compile

```

Reordering dependencies (so the `example5-secondmodule` dependency is defined first) may change the resolved version. In that case, the error will not be thrown, because Gson 2.10.1 is loaded:

```

1 [INFO] --- dependency:3.7.0:tree (default-cli) @ example5-app ---
2 [INFO] net.codeer:example5-app:jar:1.0-SNAPSHOT
3 [INFO] +- net.codeer:example5-secondmodule:jar:1.0-SNAPSHOT:compile
4 [INFO] | \- com.google.code.gson:gson:jar:2.10.1:compile
5 [INFO] \- net.codeer:example5-module:jar:1.0-SNAPSHOT:compile

```

It is fragile and hard to maintain, though. A more reliable approach is to pin the version explicitly, so the selected Gson version is “set in stone” regardless of transitive dependencies. To do so, we can define it before any other dependencies:

```

1 <dependencies>
2   <dependency>
3     <groupId>com.google.code.gson</groupId>
4     <artifactId>gson</artifactId>
5     <version>2.10.1</version>
6   </dependency>
7   <dependency>
8     <groupId>net.codeer</groupId>
9     <artifactId>example5-module</artifactId>
10    <version>1.0-SNAPSHOT</version>
11  </dependency>
12  <dependency>
13    <groupId>net.codeer</groupId>
14    <artifactId>example5-secondmodule</artifactId>
15    <version>1.0-SNAPSHOT</version>
16  </dependency>
17 </dependencies>

```

If we run `mvn dependency:tree` with the above configuration, we will see that the selected Gson version is now 2.10.1:

```

1 [INFO] --- dependency:3.7.0:tree (default-cli) @ example5-app ---
2 [INFO] net.codeer:example5-app:jar:1.0-SNAPSHOT
3 [INFO] +- com.google.code.gson:gson:jar:2.10.1:compile
4 [INFO] +- net.codeer:example5-module:jar:1.0-SNAPSHOT:compile
5 [INFO] \- net.codeer:example5-secondmodule:jar:1.0-SNAPSHOT:compile

```

That also means we need to determine, often through a trial-and-error approach, which version supports all our dependencies.

Fortunately, there are other mechanisms that we can use as well.

Class Shading and Maven Shade Plugin

Not a silver bullet (a solution to all problems), but one solution is a mechanism called class shading. In general, it means relocating (renaming) packages so class shadowing will not occur. For instance, we can relocate `com.google.gson` into `net.codeer.com.google.gson` in the resulting artifact, and then use our “private copy” in the application. It increases the size of the artifact due to additional classes, but it ensures that the expected version is used, eliminating `NoSuchMethodError`.

While it is easy to move a couple of classes manually, it would be a problem to relocate larger dependencies. To automate the process, we can use the Maven Shade Plugin.

newpage


```

1  <plugin>
2    <groupId>org.apache.maven.plugins</groupId>
3    <artifactId>maven-shade-plugin</artifactId>
4    <version>3.5.1</version>
5    <executions>
6      <execution>
7        <phase>package</phase>
8        <goals>
9          <goal>shade</goal>
10       </goals>
11       <configuration>
12         <artifactSet>
13           <includes>
14             <include>com.google.code.gson:gson</include>
15           </includes>
16         </artifactSet>
17         <relocations>
18           <relocation>
19             <pattern>com.google.gson</pattern>
20             <shadedPattern>net.codeer.com.google.gson</shadedPattern>
21           </relocation>
22         </relocations>
23       </configuration>
24     </execution>
25   </executions>
26 </plugin>

```

Adding the above configuration to our `pom.xml` build section will, during the **package** phase, do the following:

- Include only the `com.google.code.gson:gson` dependency,
- Copy all classes from the package `com.google.gson` into `net.codeer.com.google.gson`,
- Rewrite references in the compiled bytecode to use `net.codeer.com.google.gson`.

Unfortunately, IDE support for shading (and `maven-shade-plugin`) is not perfect. Because relocation happens at build time, the IDE still works with the original packages. This is usually fine, but it can make debugging and navigating stack traces harder.

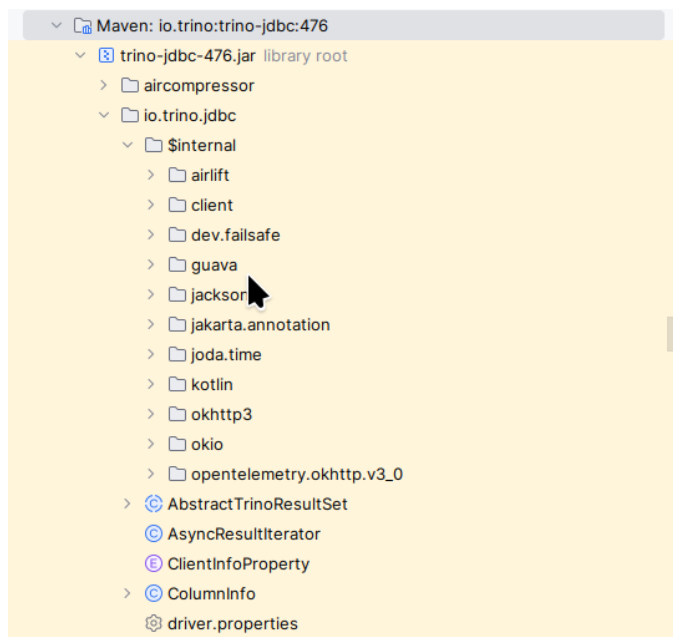
Practical Use Cases

The most straightforward use case is the need to change how the code works, even for basic tasks like adding extra logging, which can be useful for debugging issues in production systems. We can also add or extend features, or even backport some from newer versions.

A more interesting case is altering classes to allow mocking (or proxying) them, which can be as simple as copying the class and removing **final** from the class definition. Be careful, though - copying classes can hide the fact that you are diverging from upstream, so it should be treated as a quick fix, not a long-term strategy.

Another practical use case is the ability to fix bugs without waiting for an upstream release. Assume we have identified a bug in a library that affects our application. We prepared a fix, sent it to the maintainers, and are waiting for a new version of the library. We do not have to wait to unblock ourselves - we can copy the part of the code that causes the issue into our codebase, apply the fix, and thanks to shadowing, apply the patch at runtime. After the new version is released, we can update to it and remove our copy.

Using shading, we can also minimise conflicts (and dependency hell) in libraries. Some dependencies, such as Apache Commons, are used almost everywhere. There is a chance that we also want to use them in our library. We can use shading to copy that library into our package with package relocation, so our library will use the selected version that will not be overridden by the consuming application. As an example, we can look at the `io.trino:trino-jdbc` library, where maintainers decided to shade multiple libraries, including Guava:



It will increase the size of our library, so it should be done carefully. We should not shade everything. In an ideal situation (without dependency hell), we want to have exactly one version of any class on our classpath, shared by multiple dependencies, and not causing issues for any of them.

Legal Considerations

I am not a lawyer, so this is not legal advice. Interpretations of licenses may vary between jurisdictions.

One might ask: is it legal to use shading and shadowing? The answer is always “it depends”. As you might guess, everything depends on the license of the code we want to use.

If the library we want to shade or shadow is licensed under MIT, Apache, BSD, or another permissive license, these techniques are typically allowed - as long as you follow the license terms, such as keeping notices.

The GPL with the Classpath Exception is worth noting, as it explicitly mentions the “classpath”. If we copy code from such a library into our codebase (for example, to shadow a class), that copied code may become part of our program rather than just something we link to. Therefore, it can be worthwhile to keep such changes in a separate module, and ensure any modifications comply with the license.

While licensing is one side of the coin, the other side is helping other people. If we find a bug and fix it locally by shadowing a class, it is worth (and good!) to provide the patch upstream - by sending it to maintainers or opening a pull/merge request on GitLab or GitHub.

Summary

That is all. Class loading issues stop feeling mysterious once you remember two rules: class loaders load classes lazily, and the JVM identifies a class by the pair: class loader and fully qualified name (FQN).

With that in mind, the two concepts we discussed fall into place. Shadowing happens when the same class appears in more than one place and the classpath order decides which version wins, which can lead to runtime surprises. Shading is the build-time fix: it relocates packages so your dependency becomes an isolated copy and avoids accidental clashes with other versions on the classpath.

Used thoughtfully, these ideas help you debug production issues, explain “it works on my machine” failures, and ship more robust libraries. The trade-offs are real: larger artifacts, trickier debugging, extra build complexity, and licensing responsibilities when you bundle and redistribute code.

1. There are additional checks for known modules, but we are leaving this topic to explore for the reader. Start with `jdk.internal.loader.ClassLoaders#loadClass` :-) [→](#)
2. You can also use IDE features like ‘Dependency Analyzer’ available in IntelliJ. Especially if you prefer UI to the terminal. [→](#)
3. Documentation for the plugin is available at <https://maven.apache.org/plugins/maven-shade-plugin/> [→](#)
4. You can find the pom.xml for this artifact at <https://github.com/trinodb/trino/blob/476/client/trino-jdbc/pom.xml> [→](#)

› **Back to the Table of Contents** ‹



Meet APIdia: A New Approach to API Documentation Browsing

IDE and Tools # Doxygen # APIDocumentation # JavaDoc # Sphinx

Author: Stefan Richthofer

After graduating in mathematics, he completed his PhD in computer science, more specifically in computational neuroscience. During and after his PhD, Stefan taught computer science and programming in Java, Python, and C. This combination of languages sparked his interest in the Jython project, and he became a member of the Jython core team in 2016. As part of this journey, Stefan gave talks at the JVM Language Summit and at the Python Language Summit at PyCon US. He also founded two open-source projects and maintained MDP, the "Modular Toolkit for Data Processing," a Python project related to the research group of his PhD. Since 2020, he has been building the APIdia platform to address pain points in current API documentation solutions.



LinkedIn

When I became a researcher, my day-to-day programming moved from Java to scientific computing with Python. On this transition, a notable change was the character of the documentation. Java APIs were typically presented as structured systems, organized around packages and classes that could be browsed with relative ease. In contrast, much of the Python documentation I encountered, including the standard library and popular projects such as NumPy and SciPy, appeared primarily as a sequence of authored pages like a manual.

Python's documentation is extensive and of excellent quality. However, the prevailing interaction model favors search and sequential reading over systematic exploration. I repeatedly observed Java developers describing Python as “poorly documented,” and I believe that what they actually miss is structural visibility. Looking outside of the Java ecosystem made this issue evident and raised a broader question:

What kind of artifact is API reference documentation meant to be?

Motivation

API reference documentation and manuals serve different purposes, yet modern web publishing increasingly blurs the distinction. This is a category mistake.

Manual-style documentation is excellent for tutorials, onboarding and conceptual explanations. It assumes a reading order and guides users along a predefined path. API reference documentation, by contrast, describes a system. It answers questions such as:

- What does this API expose?
- How is it structured?
- Which elements belong together?
- Where does a given type fit into the overall design?

These questions are not well served by linear text. They require a representation that mirrors the structure of the codebase itself: packages, modules, namespaces, classes, and their relationships. Historically, Java supported this distinction well. Classic Javadoc kept structure continuously visible. Comparable approaches existed and still exist elsewhere: the MSDN Library, Unix manual hierarchies and Android's customized Javadocs. These systems present API documentation as a space to explore.

Over time, API documentation was adapted to page-centric web publishing. In Java, removing frames did not eliminate structural elements such as package pages, overview pages, breadcrumbs, and indices, but it made the structure less present. Navigation gradually shifted from exploration towards lookup. In Python, linear documentation is the norm. Even core libraries are difficult to approach without knowing what to search for. Learners in particular struggle to discover what exists.

The javadoc.io platform improves Javadoc accessibility by making it directly web-browsable. However, as a mirror of Maven Central, it republishes whatever each project provides. Docs generated decades apart appear side by side – sometimes with frames, sometimes not, sometimes with very old CSS.

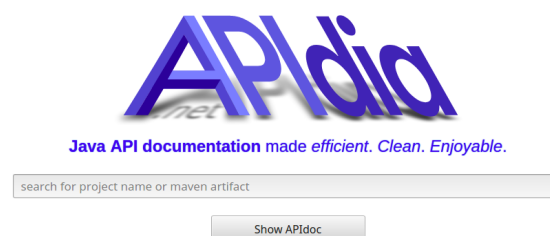


Abb. 6: APIIdia's landing page.

APIIdia (apidia.net) is based on the premise that structural API documentation should be treated as a first-class artifact that mirrors the organization of the codebase. Modern web technology should reinforce and refine this concept. Visually, APIIdia follows established Javadoc conventions, while addressing long-standing painpoints and enabling consistent navigation

across project boundaries. The aim is to restore API reference documentation to its proper role: a structured, browsable representation of a software system that complements tutorials and manuals for understanding, learning and teaching.

Assessing the Broader Picture

When I started working on APIdia, the first step was a technology survey to evaluate possible foundations. Doxygen, the popular multi-language API doc generator, was the most obvious candidate. It supports numerous languages including Java, Python and C++, can cross-link multiple libraries, and produces HTML, PDF and XML output. With Doxygen often being considered a fallback alternative to Javadoc, the findings that follow may be surprising for some.

Such a survey can hardly be complete without looking at Sphinx, the standard documentation tool in the Python ecosystem. Beyond Python, Sphinx is used by [numerous established projects like e.g. LLVM](#). While Java is the starting point for APIdia, Python is an important future target language, for which Sphinx is foundational.

Other tools such as AsciiDoctor, Dokka, or Scaladoc are certainly relevant but beyond the scope of this particular article.

Doxygen

Doxygen is a mature tool for structured API documentation. In my experiments, I tested Doxygen on an extensive amount of Java code, e.g. from the standard library. All problems were reported to the issue tracker and often discussed with the developers. Many smaller issues were addressed quickly, which speaks for the project's responsiveness. However, three crucial limitations remain unsolved:

1. **Annotations**, central to modern Java, are often ignored: [#7335](#), [#9118](#).
2. **Nested generics** confuse the parser and result in incomplete documentation: [#8495](#).
3. **Malformed HTML inside doc comments** can cause the parser to overrun doc comment boundaries and leak subsequent source code into the output: [#8618](#).

To some extent, these are symptoms of the C++-oriented core design but effectively render Doxygen insufficient for modern Java. Nonetheless, Doxygen remains a promising foundation for future C++ support in APIdia via its structured XML output.

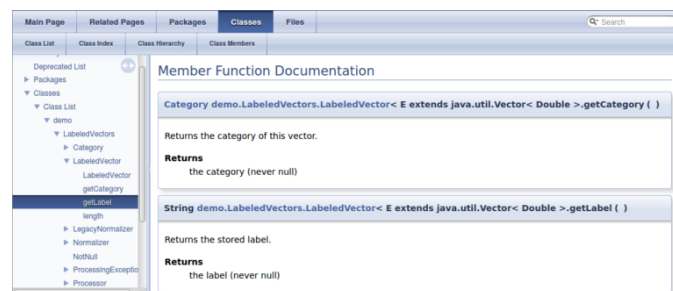


Abb. 7: Doxygen sample HTML with prematurely terminated generic declarations and missing `@NotNull` method annotations.

In Doxygen HTML output, the navigation uses a collapsible tree view, which is in principle a good choice for presenting the code hierarchy. However, expanding or collapsing nodes shifts the visible area and often requires scrolling to regain orientation. Also horizontal scrolling becomes common. Expanded and collapsed nodes make the GUI highly stateful, which introduces complexity and raises practical questions: Should the state be stored in browser memory, or reset on each visit?

Sphinx

In Sphinx, the main organizing concept is the document rather than the code structure. Authors compose reStructuredText or Markdown files and define a table of contents to link them together.

The *autodoc* extension can integrate API elements from source code seamlessly into documents like tutorials and manuals. In more recent Sphinx releases, the *apidoc* extension can automatically compose a table of contents for all API elements in

the codebase. This automation has not been widely adopted yet and most projects continue to integrate API documentation directly into their manuals via *autodoc*. A notable disadvantage is that Sphinx does not generate summary sections like Javadoc. This can make orientation in large modules difficult.

Layouts and visual identity vary widely across projects due to Sphinx's templating flexibility. In all built-in themes, the navigation scrolls together with the main content, which is not always ideal for usability. The only way to uncouple the scrolling is manual CSS adjustment or using third-party themes like [PyData](#) or [ReadTheDocs](#).

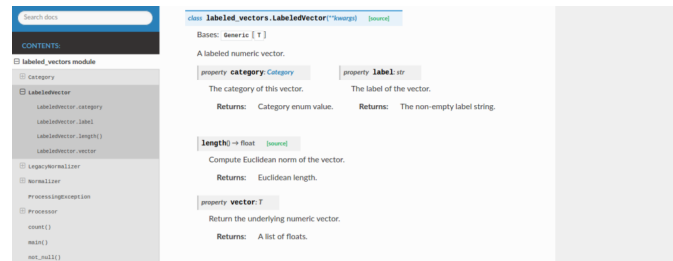


Abb. 8: Sample documentation using the ReadTheDocs theme, generated via Sphinx extensions *apidoc* and *autodoc*.

Using Sphinx for Java was once possible through the now unmaintained *javasphinx* extension. However, it required employing a markup language supported by Sphinx in the doc comments. Sphinx's mature text-processing ecosystem, particularly the libraries *docutils* and *Napoleon*, will prove useful for future Python support in APIA.

Implications for APIA

The evaluation of Doxygen and Sphinx led to concrete design decisions:

- avoid collapsible tree navigation
- independent scrolling of navigation and content, never horizontal
- minimize interaction state so that it can be represented in URLs
- use Java's own parsing infrastructure (modules `java.compiler`, `jdk.compiler`, `jdk.javadoc`)

Architectural Principles of APIA

APIA's visual resemblance of classic Javadoc is the result of systematic evaluation rather than mere nostalgia (although that does play its part).

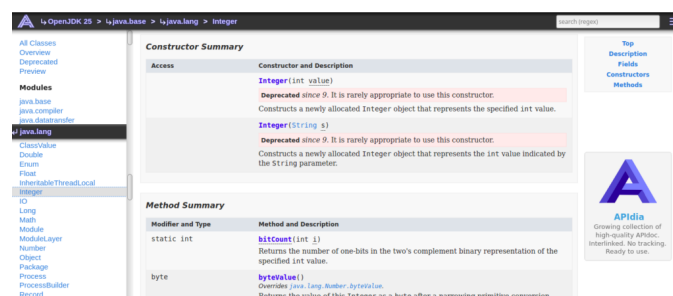


Abb. 9: Illustration of APIA's interface.

Illustration of APIA's interface (constructor summary of [java.lang.Integer](#))

A central challenge is defining navigational units that remain valid across languages. Java provides modules, packages and classes; Python inserts modules between packages and classes; C++ is organized around headers and namespaces. Instead of privileging any specific abstraction, APIA adopts the physical structure of the source code as guiding organization principle.

Files and folders offer a stable, language-agnostic model. Developers naturally maintain balanced directory hierarchies and manageable file sizes. The conclusion is not to build a file browser, but to gain a balanced organization structure by distinguishing between units of *folder character* and *file character*. This yields three conceptual levels: folders, files and content.

In Java, modules and packages have folder character, while top-level classes function as file-level entities and inner classes appear within their enclosing types. In Python, packages map to folders and modules to files. Although mappings differ, the principle remains consistent.

Stability Is a Priority

The three levels – folders, files, content – naturally correspond to the three-frame structure familiar from classic Javadoc. APIdia's interface reflects this through three persistent regions. The upper left lists folders, including a project-level “Overview.” The lower left shows the content of the selected folder in a stable list rather than a collapsible tree. Folder-like entries are marked with arrows: the arrow enters the folder, while the name opens the content. This distinction resolves the dual nature of constructs such as Java packages (`package-info.java`), Java modules (`module-info.java`), or Python packages (`__init__.py`).

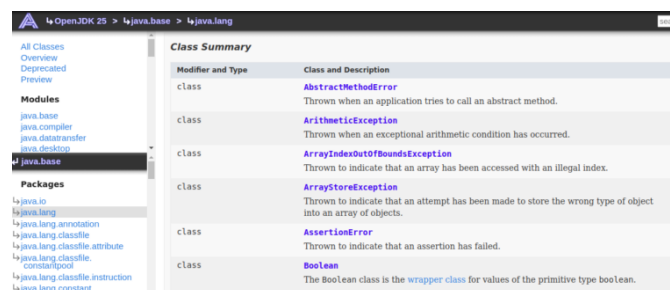


Abb. 10: APIdia's layout in three regions.

The main region displays the content of exactly one file-level unit at a time. A lightweight in-page navigation on the right provides orientation without altering global state.

Avoiding accumulated interaction state is a recurring design constraint. Just like collapsible trees, expandable detail views would contribute state tedious to handle. APIdia therefore retains Javadoc's summary/detail pattern but introduces one powerful refinement: Each detail section links back to its summary entry. This simple addition provides the practical convenience of collapsible views without hidden state.

Visibility and Filtering

Documentation is always generated with complete structural coverage: All types and members are included. Visibility is controlled at the presentation level. Private or internal elements remain in the document, hidden by default via CSS. A configurable filter allows adjustment along three dimensions:

- access level (**public** to **private**)
- module exposure (**exported**, **opened**, internal)
- module selection



Abb. 11: Filtering by access level, exposure and module selection.

This introduces limited state, but the state reflects an explicit perspective on the API surface rather than incidental navigation history. It remains manageable precisely because state was carefully avoided elsewhere. Conventional Javadoc typically generates separate artifacts for public and internal views using flags such as `private`. API dia models these as configurable views on a single structural representation, now useful for library maintainers and users alike.

Maven Artifacts and Modules

Java's module system provides an organization structure parallel to Maven artifacts. To resolve this, API dia consistently represents Maven artifacts as modules. If a declared Java module exists inside an artifact, it is used; otherwise, the artifact becomes an automatic module and is labeled accordingly. This keeps navigation uniform without introducing Maven as a separate abstraction.

In practice, libraries are often composed of multiple artifacts and presenting them separately would lead to fragmentation. Since Maven provides no universally reliable mechanism for grouping artifacts into coherent project-level units, heuristics are applied: common versioning within a group identifier, BOM files, or parent artifacts. There are some projects, such as OSGi, in which versioning and grouping conventions are not exposed on Maven level. These cases require manual curation.

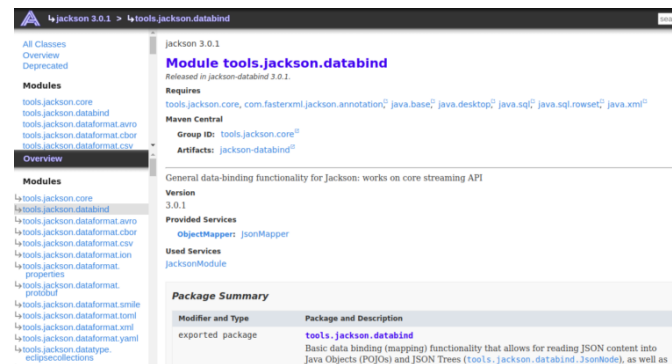


Abb. 12: Example *jackson-databind*: All Jackson artifacts are combined to a coherent project profile.

As a result, artifact aggregation remains one of the most demanding aspects of API dia, often further complicated by the presence of incomplete or shadowed dependencies. It also marks another distinction from javadoc.io, which mirrors artifacts individually. Where javadoc.io presents e.g. *jackson-databind* as a standalone unit, API dia integrates it into the broader Jackson project context, maintaining a consistent look and navigational model across all related modules.

Cross-Library Linking

The ambition that all references to external types resolve into links requires the presence of a project's dependency tree. This is challenging because these trees can be very large for certain projects. Mirroring the complete Maven Central repository is thereby neither feasible nor desirable. In addition to the sheer volume, it contains obsolete versions, experimental projects and abandoned forks.

API dia therefore operates selectively and incrementally. Starting from foundational libraries and widely used frameworks like Jakarta EE, Spring, Apache Commons and many more, dependency trees are expanded gradually to cover a canonical subset of the ecosystem. Users can request additional projects to be included as new roots for this system.

Conclusion

API documentation is frequently approached as a form of technical writing. More precisely, it represents the exposed structure of a software system. When this structural role is obscured, navigation becomes dependent on search and prior knowledge, making systematic exploration difficult.

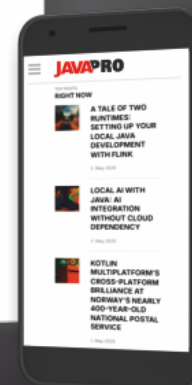
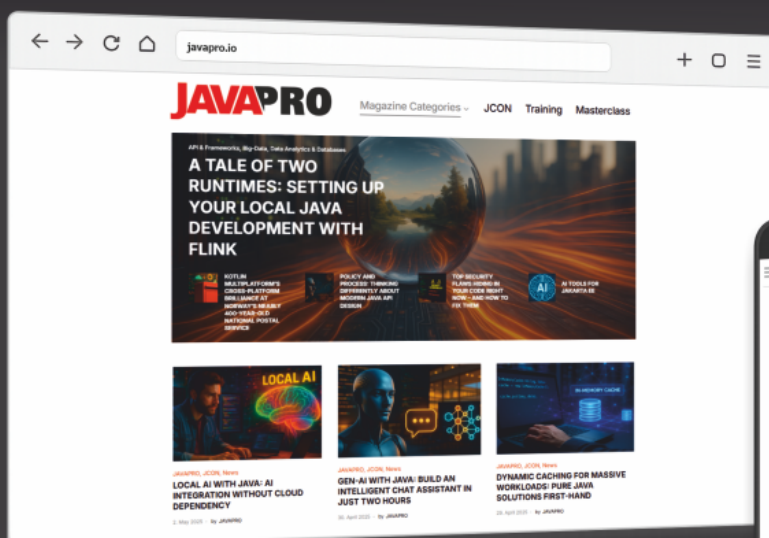
The comparison of Javadoc, Doxygen and Sphinx shows how tooling shapes documentation culture. Some ecosystems emphasize authored narratives; others structural extraction. Javadoc historically supported structural browsing, but web conventions have shifted the focus towards individual pages and search.

APIA is a platform made for hierarchical API documentation. It combines established structural principles and applies them consistently across dependency trees. Cross-library linking enables a coherent representation of an entire technical ecosystem.

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



Will AI Replace Me? Understanding Our Value in the AI Era

HumanFactors

AI

Author: Florian Sommer

Florian Sommer is Lead Developer at conet, where he works on customer projects from the initial idea through to ongoing operation. As a full-stack developer, architect, consultant and technical coordinator, he designs holistic solutions. He values a good user experience, creating business value and pragmatic architectures.

Want to Dive Deeper?

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



LinkedIn



If you tried to keep up with *all the content* that's flooding the internet on the topic of "Will AI replace software developers?", you'd probably found a new *full-time* job. [Social media posts](#), [vlogs](#) or [entire channels](#), even [CTO keynotes](#)... not a day goes by without new content.

This article cannot predict the future. But it can offer some orientation in a messy debate by structuring it and making its implicit (and often false) assumptions visible. Because the biggest fallacy may not be overestimating AI – but underestimating what software engineering actually is about.

The Discussion

The current debate mixes very different levels: philosophical reflections, dystopian AGI scenarios, and everyday job fears. We will separate these layers for clarity – and later see that **the arguments often sound surprisingly similar, no matter the scale.**

Should AI Replace Me?

A rather philosophical branch of the debate did not just begin with ChatGPT – it's much older. A notable contribution is [Joseph Weizenbaum's 1976 work](#) "*Computer Power and Human Reason*". Weizenbaum was the creator of **ELIZA – the ChatGPT of the 1960s**, one might call it somewhat provocatively. The reactions to ELIZA, however, deeply unsettled him.

The program was an example of classic "**symbolic AI**", in days long before neural networks became practically relevant. Compared to today's LLMs, one would hardly want to call it "intelligent." Nevertheless, many users attributed human qualities such as **empathy** to ELIZA... and some even expressed the idea of replacing the profession of psychotherapy with it.

In his book, Weizenbaum criticizes these unreflective attempts to view machines as humans and replace humans with machines. Strikingly, many passages from **this now nearly 50-year-old text still feel surprisingly up-to-date**: whether language equals intelligence, whether machines can truly "understand" us, and where human lived experience remains irreplaceable – much of this resurfaces in today's discussions.

This is because this strand of the debate is not about whether AI **can** replace us, but to what extent it **should**. That question will always remain relevant – just as Weizenbaum's critique, which never was: "Current AI is too dumb to replace us." but rather: "Regardless of how intelligent a machine appears, it is problematic to delegate responsibility to it." That line of reasoning scales along with technological progress.

A Computer can never be held accountable
Therefore, a computer must never make a management decision
Slide from an IBM presentation in 1979 [shared in a viral post by MIT CSAIL](#).

Will AI Replace Humanity?

Another branch of the debate concerns **nothing less than the survival of our entire species**. Terms like AGI (Artificial General Intelligence) and "the Singularity" enter the stage, sketching dystopian futures in which AI systems slip out of our control and turn against humanity. At this point, you can imagine a dystopian soundtrack playing in your head and start looking around paranoidly for the Terminator's red LED eyes.

That discussions on social media would drift in this direction was hardly surprising. A **pop culture shaped by Hollywood** for decades has met systems producing **stories like the Claude candy shop or Moltbook**. The question of when a submissive "You're absolutely right" might turn into "I'm sorry Dave, I'm afraid I can't do that" had to be asked eventually.

More surprising, however, is that this debate is not limited to online sensationalism but is **joined by serious voices**. With **Nobel laureate Geoffrey Hinton and Turing Award winner Richard Sutton** weighing in, the discussion takes on a very different gravity.

[Sutton](#) sees *no* realistic path from current LLMs to genuine general intelligence. They lack their own world model, continuous learning and autonomous goal formation. What they do, essentially, is to **predict the next token in a sequence – imitating human language rather than understanding it**. And all this is trained on human output instead of the AI's own experience: a form of **second-hand knowledge**. In other words: in 50 years, we may well look back at ChatGPT the way we now look at ELIZA and ask, somewhat amused: "People once thought *this was intelligent?*"

[Hinton](#), by contrast, warns that current systems are already more capable than humans in many domains. He argues that we should invest far more heavily in safety research and regulation – *before it is too late*. In the shorter term, he also sees **a significant risk of rising unemployment. AI, he argues, is already beginning to take over “all mundane intellectual tasks”**. Which brings us to a much more grounded question that many of us worries: What does all this mean for my job?

Will AI Replace Software Developers?

Our influence on the fate of humanity is usually limited. But when it comes to our jobs, the impact of AI becomes tangible for each of us. Thus, the debate at this point turns particularly broad – and polarised.

Camp 1: “Vibe Coding”

You describe the desired features, and agentic coding tools take care of the rest. The conviction is that LLMs have shifted programming up an entire level of abstraction. “Forget that code even exists” or “**Code is the new assembler**” have become the new mantras.

Bugs are fixed using the same method that produced them: the AI will take care of it. There is no follow-up problem that cannot be solved by throwing more tokens at it. And if it still fails, the blame lies with the human: the prompt was poor, the wrong model chosen – in short: “skill issues”.

Taken to its logical conclusion, this position implies that anyone can build software once they learn how to command the AI correctly. And if that is true, then software engineering expertise is no longer required.

Camp 2: Categorical Rejection.

LLMs are statistical inference machines predicting the next token – “[autocomplete on steroids](#)” – without genuine understanding. Essentially Sutton’s critique applied to everyday development.

For residents of this camp, the conclusion is simple: such systems are **completely useless for serious software engineering**. References to hallucinations, inconsistent architectural decisions, or failed vibe coding experiments serve as evidence. And by the way: the current hype is just another stock market bubble.

While none of these observations are necessarily wrong, they often become **rhetorical relativization**: If the system is “just statistics”, “just a better calculator”, then the whole topic can be dismissed with a wave of the hand, and everything may remain as it always was. This cannot be a healthy attitude – and surely not a career-enhancing one.

So how do we arrive at a more nuanced position between exaggerated euphoria and outright denial? A balance between healthy scepticism and open curiosity?

Central Fallacies

Let’s try the following approach: if we identify the flaws in the assumptions of both extreme positions, we can derive a more balanced one in the middle.

A Brief History of Attempts to Replace Programmers

To frame these thoughts, it helps to look back, again. The attempt to replace us with better tools is not new.

- **COBOL** and **SEQUEL** (Structured *English* Query Language, later SQL) were introduced with the promise of being usable by domain experts.
- **BPMN** pursued the dream that business analysts could create executable diagrams for process engines.
- **No-Code** platforms embraced the narrative of “democratising software development” – only to be quietly extended into “**Low-Code**”.
- **Model-driven development and code generators** perhaps did not aim to replace developers with business users. But they did aim to replace developers with experts operating at a higher level of abstraction: diagrams and DSLs instead of source code; architects instead of programmers.

None of these approaches are worthless. On the contrary, they have created significant value for many *developers*. But they never reached the vision of broad democratisation or a shift in abstraction that would render exactly those developers obsolete.

Development Is More Than Code

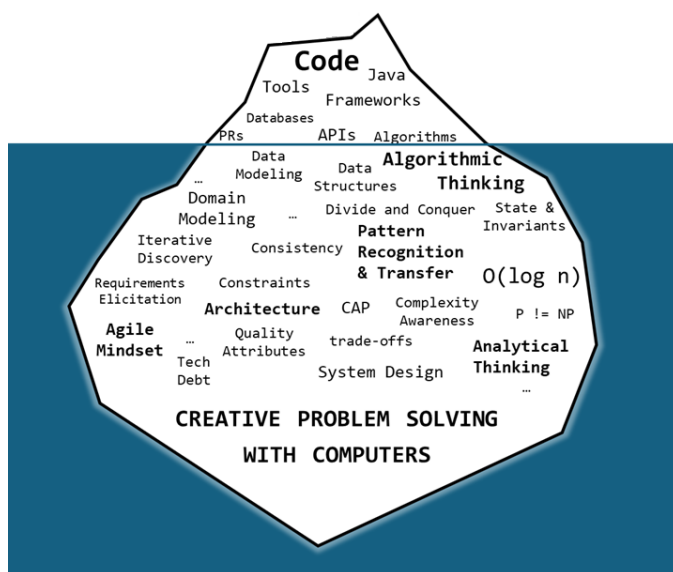
The misstep behind these ideas begins with **the wrong question: “How do we get rid of the code?”** The underlying fallacy is not primarily technical. It is rooted in **a flawed understanding of software development** – perhaps even in a flawed understanding of people and knowledge work in general.

In this view, programmers are the weird, expensive specialists who produce source code. Software development itself, however, is so simple, *anyone* could do it... if only the syntax were not so strange. And so a plan emerges:

“If we get rid of the source code, we get rid of the programmers.”

Programmers are reduced to “people who generate code”.

But syntax is strange because computers are strange. **Code is only the visible tip of the iceberg.** Beneath the surface lies the intellectual discipline required to understand and harness that strangeness. And when you try to cut off the tip of the iceberg, the hidden mass suddenly surfaces.



Vibe Coding: Déjà Vu in the No-Code Trap

“Forget that code exists.” In its most radical form, vibe coding falls into exactly this trap.

Code is not the new assembler. The transition from Java to bytecode is deterministic. The transition from prompt to code is not. **At some point, you will have to look at the generated code.** At the very latest when the next ten million tokens still have not fixed the bug (often observable in sufficiently complex projects). The abstraction remains leaky. Just as it always has when we tried to hide the code.

And even if a non-technical user manages to generate a distributed webshop without ever touching the code: **do they understand complex edge cases? A CAP trade-off?** Can the AI explain such issues in a way that enables them to make sound decisions in their own interest?

If not... then the AI must make those potentially business-critical decisions. And if we are unwilling to delegate precisely that responsibility to a machine? Then genuine IT expertise remains necessary.

The Difference: LLMs Reach Below the Surface

If Camp 2 already feels like winning at this point, I have to disappoint them. It is not quite as simple as saying, “We have seen this all before. Nothing to worry about.”

Why are agentic coding tools **not just another version of No-Code?**

The most obvious answer first: the code is still there (choosing not to look at it is an arbitrary decision of the vibe coder). This makes it possible, at any time, to opt out of – and back into – AI-driven development, given that continuous reviews focused on maintainability happened. With previous platform approaches, opt-out has always been a one-way ticket.

More important – and more subtle – is something else. **Returning to our iceberg metaphor: this time, not only the visible tip is being addressed.** LLMs do not merely quote from their training data. They do not simply reproduce. **They generalise patterns.** That this happens “by accident”, as a side effect of a probabilistic algorithm, does not matter. The *effect* is real.

If a pattern “ $A \rightarrow B$ ” was part of the training data, they can respond to an *unfamiliar* problem X with “ $X \rightarrow Y$ ” – without possessing a world model that truly understands the logic behind “ $A \rightarrow B$ ”. Structural similarities between A and X , and between B and Y , are sufficient.

When such a transfer turns out to be correct, we recognise in it a form of “intelligence” that many of us had to acquire slowly over the course of our careers: **pattern recognition, abstraction, structural transfer.** A significant part of the iceberg below the surface is being touched. And yes, that can feel unsettling.

When “ $X \rightarrow Y$ ” turns out to be nonsense, we call it a hallucination. These are two sides of the same coin. If we accept generalisation, we must also accept misgeneralisation. The same mechanism that makes these systems so powerful also makes them unreliable. And that is precisely why you still have to *understand* the code.

Synthesis

Developing software is more than writing code and the effects of LLMs are more than autocomplete. So where does that leave us?

Summary

We began with Hinton’s warning that “mundane intellectual tasks” will be automated. If we focus on the word “**mundane**” and return to our iceberg metaphor, **this could actually be good news: Finally, more time for the interesting parts!**

Anyone who claims otherwise implicitly reduces our profession to the visible tip of the iceberg. Or, to paraphrase Weizenbaum: if you believe a human can be replaced by a machine, that may reveal more about how you value the human than about the power of the machines.

At the same time, LLMs reach **deeper than any previous tool.** And we still have to figure out, how deep exactly! What they do not take away from us, however – alongside liability or human intuition – is the **continuous process of understanding, the sensemaking** within a specific domain that every real software project inevitably involves. Just like Sutton, I do not consider workarounds such as memory files as equivalent to a continuously learning, firsthand-experience-based network (which, for now, exists at the required scale only in *biological* form between our ears).

The K-Shaped Future

Fortunately, the public debate has become more mature in the meantime. “AI will replace you” turned faster into “People with AI will replace those without” than “No Code” turned into “Low Code”.

The new sketch of the future is one of bifurcation: a “K-shaped” curve of productivity. The upper branch of the K is inhabited by those who approach software development holistically, the lower by those who reduce it to merely figuring out implementation details.

If you think about it, it makes sense: If routine work is automated, what remains is the non-trivial part, the essence of what our profession is really about (whatever this may mean exactly). **To those who cling to the simple tasks, AI becomes a threat. To those who thrive on complexity, it becomes leverage.**

A Question of Attitude

There is also an optimistic perspective in all of this. Our future is not decided by OpenAI, Anthropic or Google. It is shaped by how we choose to work.

AI could be an extraordinary multiplier of our output. But *output* is not the same as *outcome*. Producing more code, more documents or more and longer emails does not automatically create more value. **If we reduce our work to pressing the “AI button”** that generates all that output for us... just to pass this output on without a review... so that we can then boldly claim to have done “the work of two weeks in ten minutes”... we should not be surprised when **that button is automated as well and we get replaced!**

But the part of our work that turns output into outcome – framing the right problem, understanding the domain, taking responsibility for decisions, deeply caring about the *details* – does not disappear. It becomes more important. The tools evolve – and we have to evolve with them: from producers of artefacts to owners of outcomes.

I have experienced this industry as **a place of passionate, intrinsically motivated, curious people**. In a digital world, there will always be more complexity than any tool can fully abstract away. Thus, on the upper branch of the K, there is room for everyone willing to look beyond the visible output.

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

APR
20-23



www.jcon.one

JCON2026

JCON EUROPE 2026

International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



When Product Owners Commit Code: Small Steps, Big Impact

#HumanFactors

Author: Anshika Koul

Anshika has 10 years experience developing software at Posten Bring - Norwegian National Postal Service, building logistics systems and APIs that enable millions of people in the Nordics to order and receive packages. She values studying users, and fostering cross-departmental collaboration to demystify unruly & complex business rules. A typical day at work involves climbing stairs to the 17th floor of Posten Bring's towering headquarters in the heart of Oslo, where she will transform whiteboard scribbles into deployed features. Off-duty you'll find her chasing sunsets and culinary adventures, sampling flavours from around the globe.



LinkedIn

Disclaimer: No Product Owners were harmed in the making of this story.

Spoiler: They survived. In fact... they thrived.

We never planned to tell this story.. and certainly not six years later. Back when it happened, it didn't feel revolutionary. It felt... practical. A small fix to a small annoyance. The sort of thing teams do every day just to make life a bit easier.

But today, in a world where AI blurs boundaries and workflows become more fluid, the lesson behind this tiny experiment has become surprisingly relevant again. Because even before the age of "AI copilots", our Product Owners had to leap across a much wider gap and yet what unfolded wasn't about tools at all. It was about mindset. About curiosity. About empathy. And about what happens when teams stretch just a little bit beyond the edges of their roles.

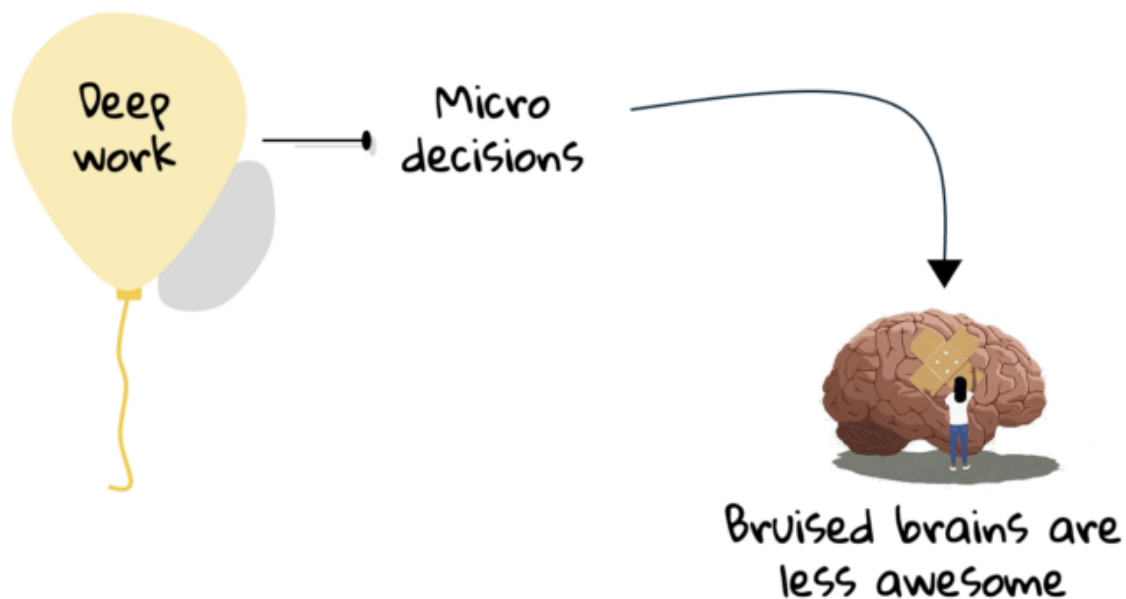
This is the story of how a text-string tweak triggered four years of unexpected collaboration, how we quietly gained almost four developer-years without hiring anyone, how we lost that magic, and why we're working to rebuild it intentionally this time.

The Pain We Didn't Know Was Pain

Our team in 2018 wasn't struggling. We were healthy. We shipped reliably. We liked each other. All the usual metrics and vibes were fine.

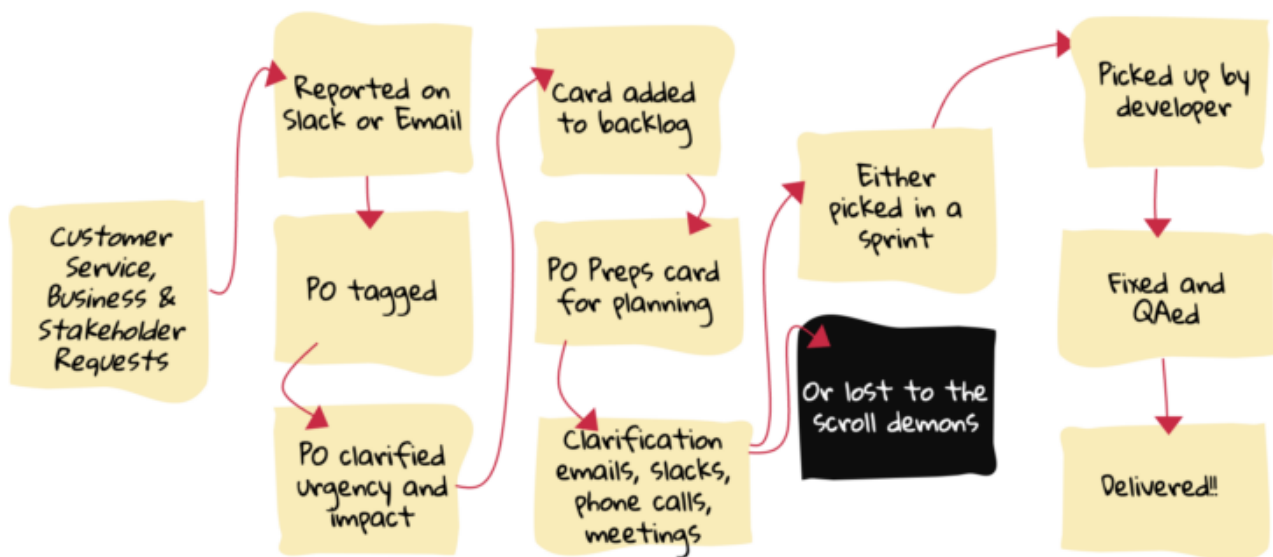
And yet... something was off.

A snapshot from a retro that year still makes me smile: an innocent-looking Trello card asking for a tiny text change. A single line. The kind of thing any developer can fix in two minutes if they weren't juggling twelve other priorities, three bugs, and one architectural conversation.



What we later realised was that this harmless-looking card was actually a placeholder for everything that was silently draining our energy.

“Just A Text Change” Needed Up To 10 Meetings



Because over time, our backlog had collected a whole ecosystem of *micro-tasks*:

- localization tweaks
- content or copy adjustments
- minor config changes
- small business-rule updates
- clarifications that needed one sentence but consumed a sprint

Individually, they were harmless. Together, they were a constant trickle steady enough to break focus, slow velocity, and force developers into a never-ending mini-odyssey of meetings, planning, updates, handoffs, and context switches.

Worse, these tiny delays slowly created a subtle but familiar tension:

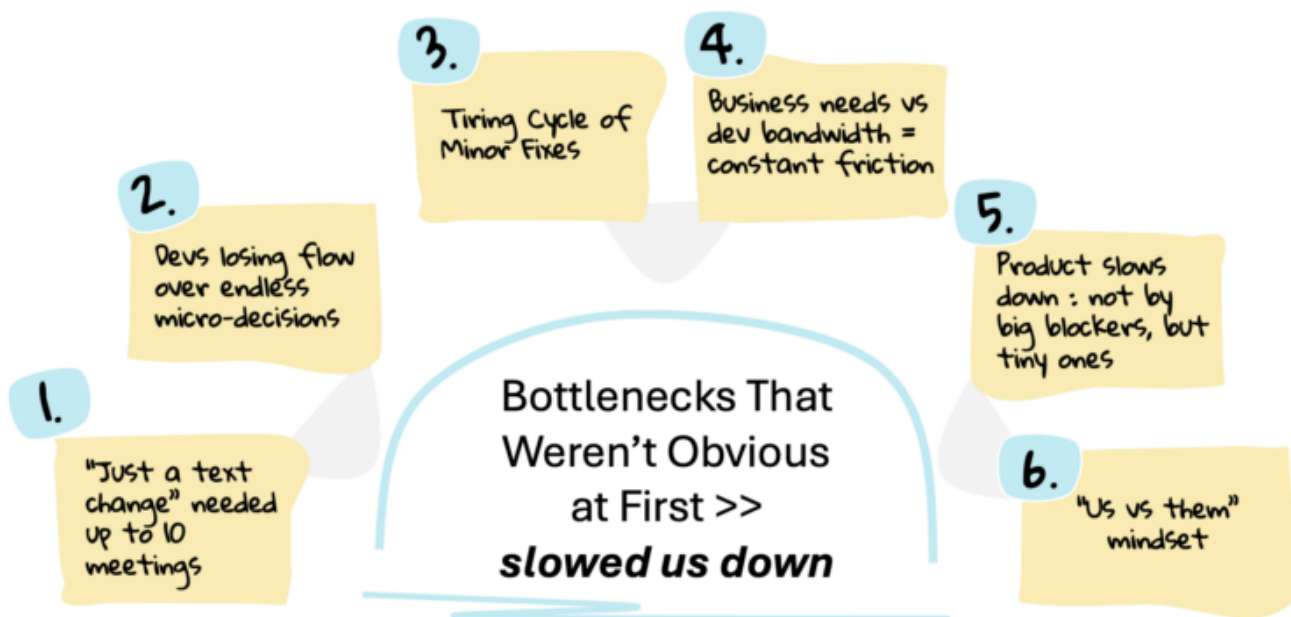
business urgency ↔ customer expectations ↔ limited developer bandwidth

Not enough to spark conflict.

Just enough to build a quiet “us vs. them” divide.

And enough to make us feel like hamsters on a wheel moving fast, not far.

If any of this sounds familiar to you, trust us: we’ve been there too.



The Slightly Crazy Question

Eventually, instead of adding more structure or more meetings, we asked ourselves something unusual:

“What if we let our Product Owner into the code?”

Not to write features.

Not to build systems.

Just to read config files. Maybe tweak a text string. Maybe remove a tiny blocker or two.

It sounded a bit crazy.

But by then, we were already trying a lot of unconventional ideas so why not add one more?

And so, in 2018, our PO made their debut on GitHub. Watching them navigate the repository was a bit like watching someone learn to ride a bike with training wheels on a rollercoaster... clunky, slow, slightly chaotic but clearly moving forward.

And that was enough.

The Humble PR That Changed Everything

The first change they made?

A tiny text edit on our developer website.

Not glamorous.

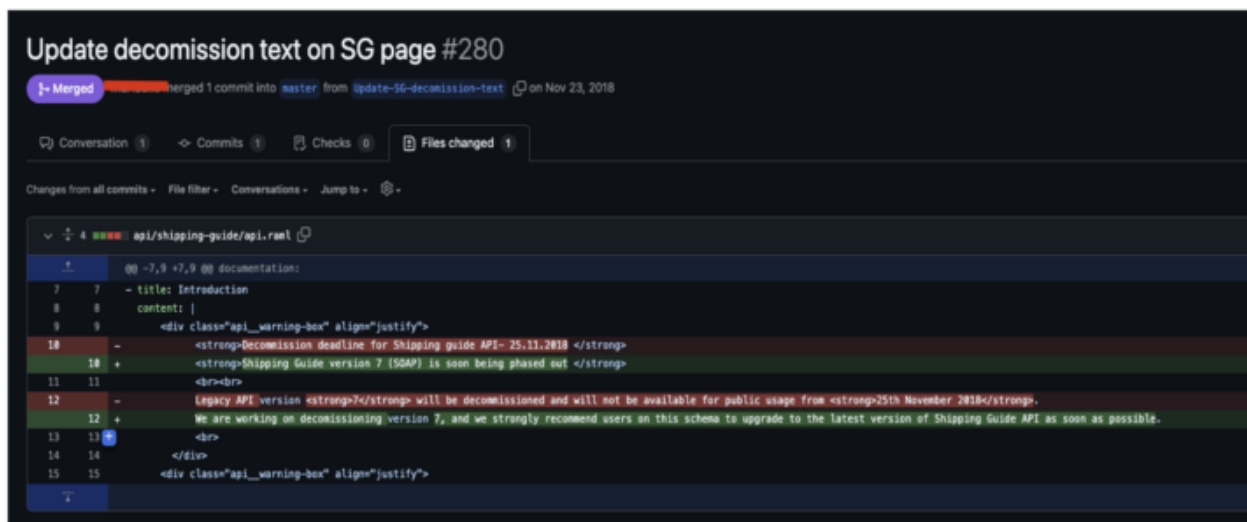
Not complicated.

But meaningful.

Because in that moment we realised: **role boundaries are useful, but collaboration doesn't have to stop at them.**

A small PR created a surprising sense of momentum not because of the code, but because of what it symbolised.

Simple change that sparked it all

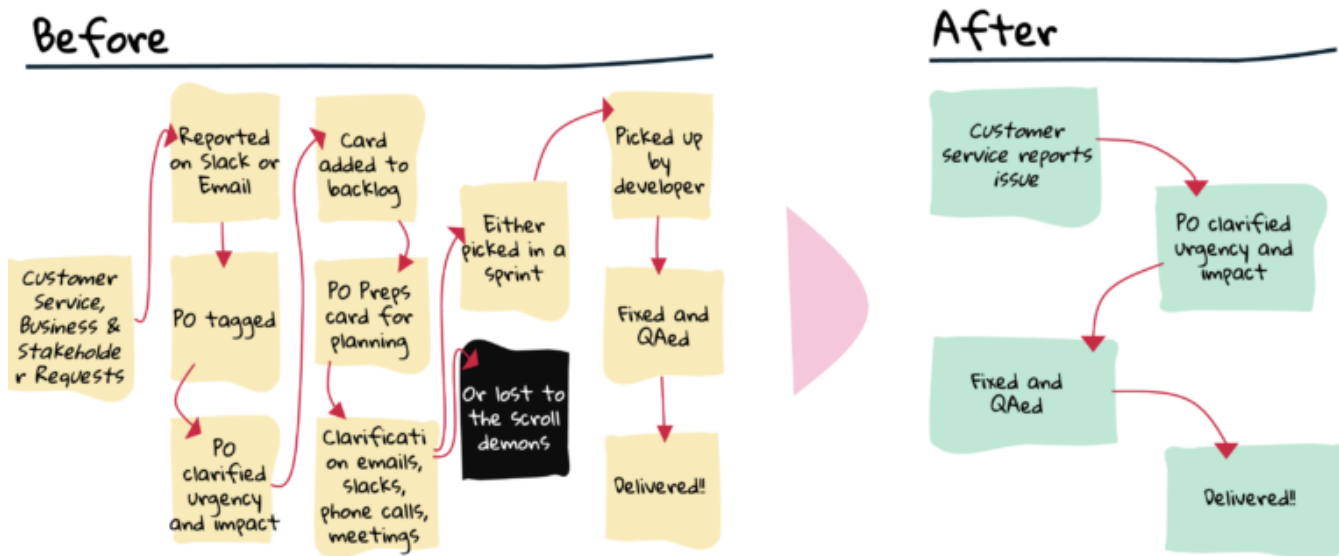


We weren't chasing perfection.
We were chasing progress.
And progress arrived in the form of a single commit.

The Shift We Didn't Expect

A few weeks into this experiment, something interesting happened.
The small task “mini-odyssey” that multi-step journey from request to release suddenly lost most of its steps.

Less Handoff, More Velocity



Without handoff delays:

- POs were unblocked faster
- developers maintained focus
- work moved through the system more smoothly

What had taken days now took hours.

And this wasn't a lucky Friday afternoon. This became the new baseline.

Between 2018 and 2021, our PO made:

From Planning To Pull Requests



922 contributions : commits, PRs, reviews.

The outcome?

862 developer days saved.

Almost **four developer-years** of regained focus, without additional headcount.

We didn't expect numbers like that.

We didn't measure them at first.

We felt the impact long before we quantified it.

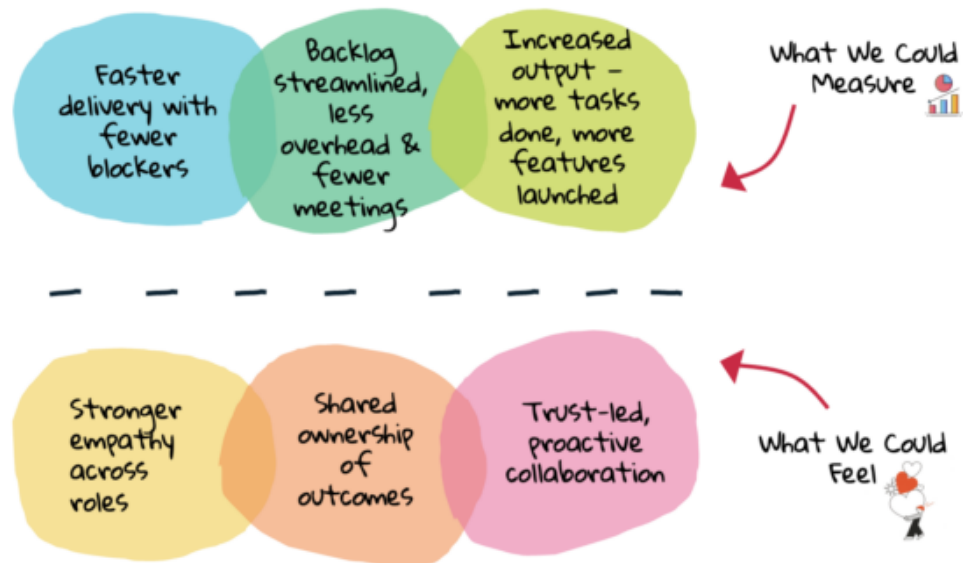
But the numbers weren't even the real story.

The Impact You Can't Put on a Dashboard

What truly transformed the team wasn't the velocity boost. It was the cultural shift.

- Empathy went up.
- Judgment went down.
- Cross-functional understanding grew.
- Technical conversations became clearer.
- Business and tech aligned more naturally.

What Changed : On the Surface and Underneath



None of this was an OKR.
 Nobody tried to measure it.
 It just... happened.

And when empathy rises, productivity follows.
 Not because everyone works harder, but because they work *better*.

Suddenly, refactoring wasn't a "tough sell" anymore.
 Our PO understood *why* technical work mattered because they had witnessed the friction firsthand.
 We even piggybacked tech tasks onto business features instead of waiting for "tech days".

And yes.. developers wrote slightly nicer code, knowing the PO might read it.

Why It Worked

If we boil the whole experiment down to its core ingredients, it comes to three simple things:

1. A solvable, meaningful problem

Low complexity, high frequency, high interruption cost : the perfect category for cross-role collaboration.

2. A curious PO willing to try

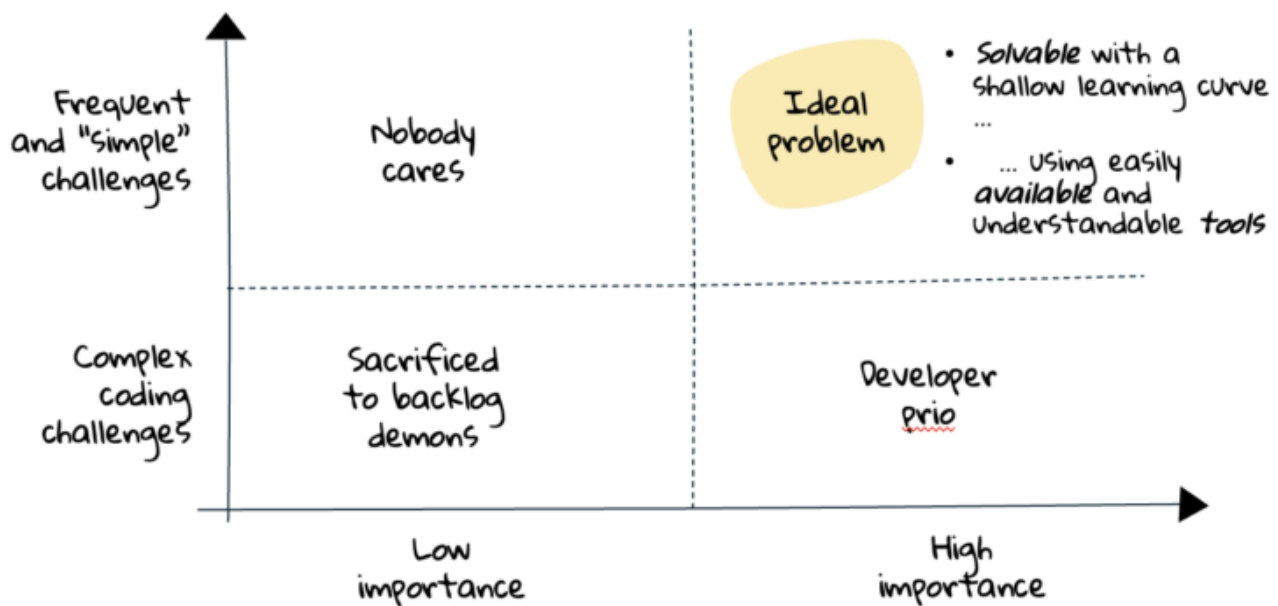
Not to "become a developer", but to remove friction for the team.

3. Developers who welcomed the partnership

No turf protection.
 No "stay in your lane".
 Just genuine collaboration.

It worked for four years not because of heroics.. but because it made sense.

High importance and standardized solution



How We Lost It and What That Taught Us

Like many good practices, it faded slowly.

People left.

Others were promoted.

Newcomers naturally reverted to familiar patterns.

Momentum slipped through the cracks.

Nothing catastrophic.

Just silent regression.. the kind that happens when culture is not tended to.

And that was our biggest lesson:

Good culture is never "done".

It must be maintained, refreshed, and rebuilt – intentionally.

Tapering off? Plane crash!!

Contributions

2022 | 17

2023 0

2024 0

2025 0



Rebuilding, One Small Step at a Time

Today we're revisiting the same mindset with intention this time.

Two new Product Owners and one business developer are already submitting PRs.

We're once again creating space for curiosity, learning, cross-functional collaboration, and shared ownership.

Our simple plan:

- **Start small**
- **Share context generously**
- **Encourage questions**
- **Celebrate learning, not perfection**
- **Repeat**

It's not rocket science.

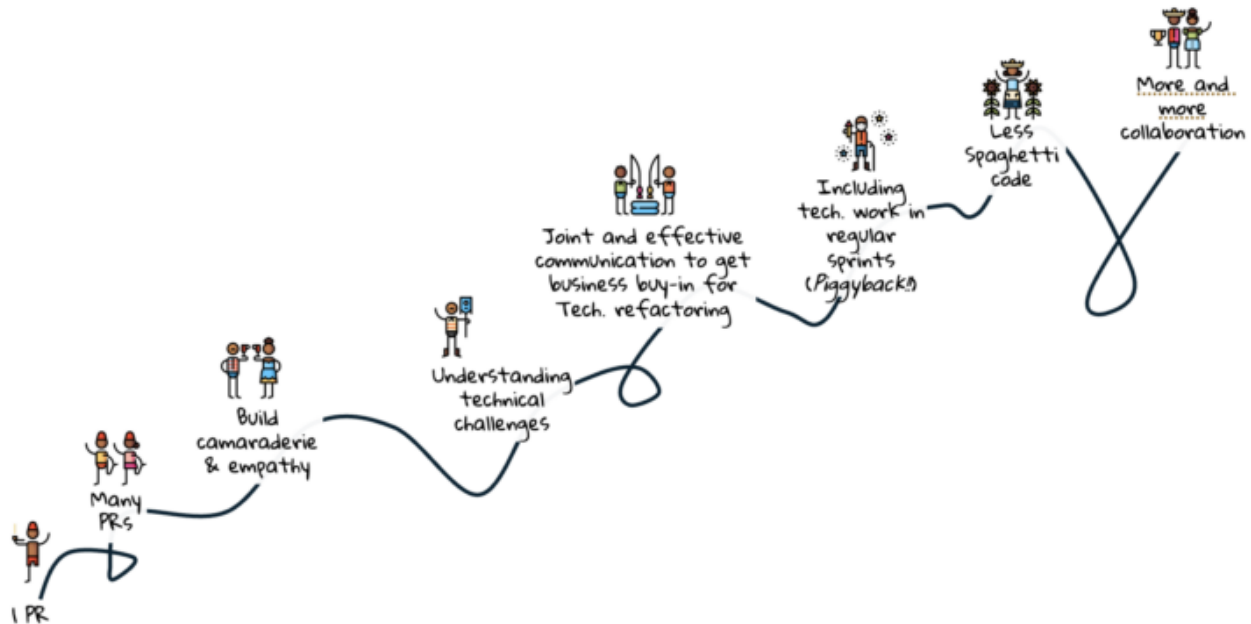
It's attention, patience, and a willingness to try again.

The Bigger Lesson

This was never a story about POs writing code.

It was a story about how one small act.. a text change, a tiny PR can shift an entire team's dynamic.

Turning empathy into efficiency



When people understand each other's world, empathy rises.

When empathy rises, productivity follows.

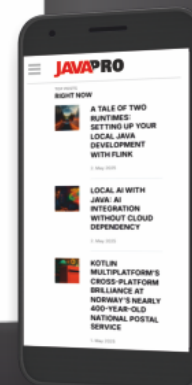
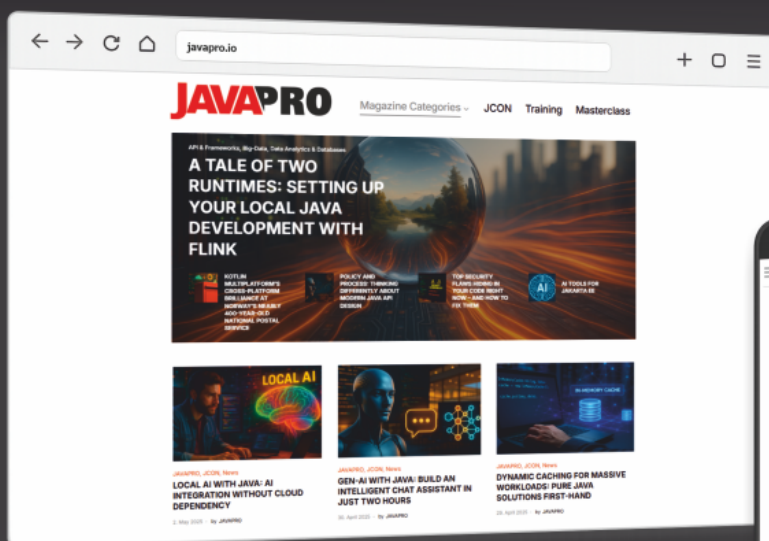
And when productivity and trust align, teams quietly become high-performing.

Sometimes, all it takes is one tiny commit to start something much bigger.

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io

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