

42 **BUILD GPU ACCELERATED
LLM INFERENCE LIBRARIES**

115 **RE-ARCHITECTING
FOR THE AGENTIC AI FUTURE**

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

Special Edition 4 / 2026

JAVAPRO

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

Java in the Age of AI

63 **TALK TO YOUR DATA:
NATURAL LANGUAGE DATA ACCESS IN JAVA**

68 **BUILD VECTOR DATABASE
APPS WITH PURE JAVA**

71 **HIGH-PERFORMANCE
VECTOR-SEARCH GRIDS WITH JAVA**

75 **LLMS FOR JVM CODE: WHICH MODELS TO
TRUST, WHICH BENCHMARKS TO BELIEVE**

85 **BUILDING PRODUCTION-READY
AI AGENTS WITH JAVA AND SPRING AI**

96 **THE 5 KNIGHTS
OF THE MCP APOCALYPSE**

100 **PETABYTE-SCALE AI MEMORY
WITH SERVERLESS JAVA**

Save the Date



May 31 - June 3, 2027
www.jcon.one

Java and the AI Renaissance: Late to the Race, Built to Lead

The AI landscape is shifting at a breathtaking pace, and one recent announcement has made that unmistakably clear. With TurboQuant, Google introduced a breakthrough in model efficiency by compressing neural representations into ultra-compact formats, drastically reducing memory footprint and compute requirements. The immediate impact was striking - not only in performance benchmarks, but even in financial markets, where shares of major AI hardware vendors briefly reacted to the implication that less hardware might be needed for the same workloads. TurboQuant signals a turning point: optimization is no longer incremental, it is architectural.

Yet while TurboQuant addresses short-term memory and inference efficiency, the larger challenge remains unresolved. Long-term AI memory - the persistent, growing knowledge layer behind intelligent systems - still operates on architectures that were never designed for it. This is where a new generation of Java-based technologies is beginning to redefine the landscape.

With EclipseStore 4, the Java ecosystem now has a fully native, pure Java vector database engine. For the first time, Java developers can build GenAI applications entirely within the Java runtime, without relying on external vector databases or polyglot architectures. Combined with Eclipse Data Grid, which extends this model into a distributed in-memory vector search grid, developers can scale these applications seamlessly across clusters. Both projects are fully open source, developed under the Eclipse Foundation by a team that has spent over a decade building core technologies such as Eclipse Serializer and EclipseStore. Every feature is transparent, community-driven, and built with a clear commitment to the Java ecosystem.

At the same time, Cyrock.AI introduces a complementary perspective on the problem. If TurboQuant optimizes the efficiency of models, Cyrock.AI focuses on the efficiency of knowledge itself. Its Knowledge Fabric architecture reimagines how long-term AI memory can be structured, accessed, and scaled. By applying principles similar to serverless computing - allocating resources only where data is actively used - it enables a fundamentally more efficient way to handle petabyte-scale knowledge systems. The implications are significant: not just lower costs, but the possibility of truly persistent, continuously evolving AI memory.

Taken together, these developments point to a broader transformation. Software is moving from being purely code-driven to being shaped by context. Java systems have long been built on deterministic logic, where behavior is explicitly defined and controlled. That foundation remains critical, but it is no longer sufficient on its own. Modern systems increasingly operate by interpreting situations, combining data, and dynamically determining outcomes.

This shift introduces a new paradigm: from direct control to guided autonomy. Developers are no longer only writing execution paths; they are defining boundaries, constraints, and environments in which systems can act. The focus moves from specifying every step to ensuring that decisions are made within acceptable limits.

At the center of this transformation lies context. AI systems depend on access to relevant, structured, and trustworthy information. Data is no longer passive storage - it becomes an active component of system behavior. Technologies such



Markus Kett
Editor in Chief
JAVAPRO

<https://linkedin.com/in/markuskett>
<https://twitter.com/MarkusKett>

Markus Kett and his teams have been working on IDE tools for Java and database development, as well as various open-source projects for more than 20 years. Markus is co-founder of MicroStream as well as co-founder and CEO of Cyrock.AI, the team behind the Eclipse open-source projects EclipseStore, Eclipse Serializer, and Eclipse Data Grid. At Cyrock.AI he is working on the first autonomous, petabyte-scale enterprise AI memory. He is also the editor-in-chief for the free JAVAPRO magazine and the founder and co-organizer of the JCON Java community conference series. Markus is an independent author for several magazines, and a speaker at numerous international developer conferences, user groups, and meetups.

as vector search, graph traversal, and distributed in-memory data grids are not just performance optimizations; they are the foundation of how modern systems reason.

This also raises new questions of governance and responsibility. As systems become more autonomous, ensuring transparency, traceability, and control becomes essential. The challenge is not only to build powerful systems, but to build systems that can be trusted.

Through all of this change, one constant remains: Java itself. Its stability, maturity, and performance make it uniquely suited for this new era. Rather than being replaced, the Java ecosystem is expanding - embracing AI, distributed systems, and new data paradigms while preserving its core strengths.

What we are witnessing is not just the evolution of tools, but the evolution of software itself. From code to context, from deterministic execution to intelligent systems, the role of developers is changing. The question is no longer only what systems should do, but how they should think - and what they should be allowed to decide. This issue explores that transformation.

Before closing, a brief personal note. As JAVAPRO we're neutral related to technologies, vendors and products. Exceptionally, I like to highlight projects that my team and I have been building for more than a decade: Eclipse Serializer, EclipseStore and Eclipse Data Grid. These technologies are the result of continuous iteration, shaped and refined through close feedback from the Java community. Today, all of those ideas and concepts come together into a powerful, fully open-source pure Java GenAI stack built specifically for Java developers - created with passion and a clear goal: to contribute something meaningful that helps the Java community and strengthen Java's role in the AI era. We are especially proud to introduce these frameworks at JCON EUROPE 2026, and I hope they provide real value to you in your GenAI projects.

A big thank you goes to our fantastic authors of this issue as well as to all the authors who write enthusiastically for JAVAPRO online - without you JAVAPRO would not be possible.

If you are part of a Java User Group, take advantage of the free JUG tickets, connect with your local Java User Group, and get involved. Let's strengthen the Java ecosystem together and meet in person at JCON. I wish all of you an inspiring and unforgettable event.

With the kind support of our partners. JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors



06

JCON

A Big Screen Experience for Java Developers & Architects

by JAVAPRO

50

CORE JAVA

Reactor Has Died. Long Live REACTOR!

by Marcin Chrost

08

CORE JAVA

Effective Pattern Matching 2026 Edition

by Cay Horstmann

58

API & FRAMEWORKS

The FFM API: OpenJDK Changed the Game for Native Interactions and #JavaOnRaspberryPi

by Frank Delporte

13

CORE JAVA

Data Oriented Programming with Java

by Birgit Kratz

63

API & FRAMEWORKS

Talk to Your Data: Natural Language Data Access in Java

by Marco Belladelli

20

CORE JAVA

Java 26 Adopts HTTP/3 with the Evolution of HttpClient

by Wanderson Xesquevixos

68

GENAI

Build Vector Database Apps with Pure Java

by Richard Fichtner

23

CORE JAVA

Java 26 GV: Making Data Streaming Faster and More Relatable in Containers

by Varsha Sharma & Shubham Goel

71

GENAI

High-Performance Vector-Search Grids with Java

by Gerald Kammerer

25

CORE JAVA

1BRC - The First 80% of the Journey

by René Schwietzke

75

GENAI

LLMs for JVM Code in 2026: Which Models to Trust, Which Benchmarks to Believe

by Artur Skowronski

42

CORE JAVA

Java For AI: Building GPU Accelerated LLM Inference Libraries with TornadoVM

by Michalis Papadimitriou

81

GENAI

Java 26 + Agentic Microservices: Cloud Performance Re-Imagined for FinTech

by Sibasis Padhi

85

GENAI

Building Production-Ready AI Agents with Java and Spring AI

by Yuriy Bezsonov & Sascha Möllering

115

ARCHITECTURE & MICROSERVICES

Capability Mesh: Re-Architecting for the Agentic AI Future

by Makan Sepehrifar

89

GENAI

A2A Building Interoperable AI Agents with Java

by Loïc Magnette

118

CI & CD

Bye-Bye Jenkins: Why We Migrated our Pipelines to Gitlab CI

by Lukas Pradel

96

GENAI

The 5 Knights of the MCP Apocalypse

by Jonathan Vila

121

PROJECT MANAGEMENT

AI's Impact on Team Dynamics

by Erik Pronk

100

GENAI

Petabyte-Scale AI Memory with Serverless Java

by Markus Kett

125

SECURITY

PQC - It Is Time to Act

by Sebastian Hempel

106

ARCHITECTURE & MICROSERVICES

Zero Migration Java - Staying Current Without Breaking Your App

by Grace Robinson & Laura Cowen

129

SECURITY

The Myth of Stability: Java's Software Supply Chain After Log4Shell

by Steve Poole

110

ARCHITECTURE & MICROSERVICES

Enterprise Release Evolution

by Matthias Zieger

133

JVM LANGUAGE

Kotlin Kontra Java - Part 1- Ecosystem

by Richard Gross

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) 2026
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.

A Big Screen Experience for Java Developers and Architects

JCON EUROPE 2026 | April 20–23, 2026 | Cinedom Cologne

Anyone talking about modern software development today can hardly avoid Java. Few technologies have shaped the enterprise world as sustainably, and few communities are as vibrant, diverse, and innovation-driven. This is exactly where JCON comes in: as a meeting point, a platform, and an experiential space for developers from all over the world. With participants from more than 70 countries, JCON has long since become an international gathering. **Openness, knowledge sharing, and a strong sense of community** remain at its core.

Cinematic Coding: Live Hacking on the Big Screen

One of JCON's defining characteristics in Germany is its unique setting: **a conference held in a multiplex cinema**. What may sound unusual at first quickly proves its value: live coding, architectural diagrams, and complex concepts are displayed on a massive cinema screen – crisp and clearly visible from every seat. This “cinematic coding” concept is not just a visual highlight; it reinforces JCON's ambition to make content truly tangible – not just consumable.

Author:



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.

<https://www.linkedin.com/company/javapro>

<https://www.javapro.io>

Your Individual Journey: Flexibility as a Principle

JCON does not follow a rigid structure. Instead, participants create their own experience by selecting from **parallel tracks, sessions, and workshops**. Using the central **schedule tool**, attendees can plan and prioritize content according to their interests. The result: **a conference that adapts to its participants – not the other way around.**

Deep Dives into Practice: From Core Java to Production AI

With more than 100 sessions and international top speakers, JCON covers the full spectrum of modern Java development – from Core Java and cloud-native architectures to current topics such as AI and platform engineering. The keynotes set the direction: from **Sharat Chander's** perspective on 30 years of Java, to **Johann Strauss** and **Markus Kett** positioning AI within production architectures, to **Markus Eisele** shifting the focus toward software quality in the age of AI.

The focus is on **practical solutions from real-world projects**:

- **Modernizing** existing systems
- **Performance and scalability** in cloud environments
- **Integrating AI** into existing architectures
- **Best practices** from production systems

This strong focus on real-world application is particularly evident in the workshop program.

Workshop Day: Rolling Up Your Sleeves and Writing Code

The dedicated **workshop day** is not an add-on, but the practical foundation of JCON. Across two time slots, participants work intensively on real-world challenges.

Highlights of the masterclasses include:

- **Sven Ruppert** guiding participants through building a fully Java-based, self-hosted RAG system – with a focus on security and data control.
- **Florian Habermann** demonstrating how to build GenAI architectures using JVector and EclipseStore without additional infrastructure.
- **Christian Kümmel** presenting “*Caching and Beyond* –

Smarter Data Processing with Java”, focusing on reducing system complexity.

- **Sandra Ahlgrimm** and **Bruno Borges** showing how to modernize existing Java applications using AI-assisted workflows – from analysis to migration.
- **Tim te Beek** and **Rick Ossendrijver** challenging traditional testing approaches in *“Break Your Testing Habits”*, introducing modern JUnit strategies.
- **Don Bourne** and **Michal Broz** demonstrating how AI agents can be effectively applied in Java environments.
- **Mohammad-Ali A’rabi** covering supply chain security in *“Java Supply Chain Security with Docker”*.
- **Cristian Schusztter** showing how AI tooling can be used to maintain and evolve legacy systems.

The combination of **hands-on** formats, **small groups**, and **experienced speakers** makes the workshops one of the most **effective learning formats** at JCON.

Future Focus: The Premium AI Track for Enterprise Systems

A key component of JCON is the **Premium AI Track**, spanning two days and focusing on real-world production systems. This is not about short-lived hype, but about architecture, operations, and responsibility.

Key questions include: How can AI models be meaningfully integrated into existing systems? Which architectural decisions are sustainable? And how can code remain maintainable despite AI assistance?

The track features a wide range of high-quality sessions, including:

- **Jonathan Vila (Sonar)** with *“Seven Habits of Highly Effective AI Java Coding”*, focusing on quality and handling of AI-generated code.
- **Kevin Dubois (IBM)** with *“Local Development in the AI Era”*, exploring controlled local AI usage with regard to cost, privacy, and reproducibility.
- **Brian Sam-Bodden (Redis)** with *“Design Patterns for Multi-Agent Systems”*, comparing architectural patterns for agent-based AI systems.
- **Emily Jiang (IBM)** with *“Building Intelligent Enterprises: AI-Powered Java with LangChain4j & CDI”*, demonstrating AI in enterprise architectures.
- **Ronald Dehuysser** with *“Green AI with JobRunr”*, focusing on sustainable AI on the JVM.

Networking & Mentorship: Connecting with Java Champions

Beyond the sessions, JCON offers direct access to experts to discuss individual project challenges – including dedicated **1:1 speaker meetings** for more focused, in-depth conversations.

The **Mentorship Hub** – featuring Java Champions such as **Bruno Souza** – gives participants the opportunity to engage directly with experienced developers on architecture, career paths, and

real-world challenges. This is complemented by spontaneous discussions in the **expo area** and hallway track conversations. Events like the **Happy Hour** and the **VIP Party** create additional space for exchange in a more informal setting.

Additional perspectives come from formats such as the **interview booth** hosted by **Baruch Sadogursky** and podcast interviews by **Frank Delporte**, which extend and deepen conversations from the sessions and capture participants’ perspectives.

Even small details reflect the community spirit – such as the annually designed **JCON T-Shirts**, which have long become collector’s items and conversation starters.

It is this mix of structured and spontaneous interaction that makes JCON a place where discussions continue beyond the talks – and often truly begin there.

Sustained Knowledge: More Than a One-Time Event

JCON is not limited to just a few days per year. Many sessions are recorded and later published via JAVAPRO on YouTube (<https://www.youtube.com/@JAVAPRO-channel>), complemented by interviews and additional content formats.

At the same time, topics are expanded both before and after the event: on **javapro.io**, not only speakers but also developers and experts from the community regularly publish articles on key topics – from AI and architecture to concrete best practices.

JCON thus becomes a hub within a year-round ecosystem for Java professionals.

Conclusion

In a world of rapidly evolving technologies, developers need places that provide orientation. JCON is exactly such a place:

A meeting point for people who don’t just write software – but shape it.

A space for exchange on equal footing – and an event that demonstrates how vibrant and future-proof the Java community is.

Save the Date

The date for the next JCON is already set:

JCON 2027

May 31 – June 3, 2027

www.jcon.one

-> Back to the Table of Content



#JAVAPRO #COREJAVA

Effective Pattern Matching 2026 Edition



Since Java 14, the Java switch and instanceof statements have been enhanced, in multiple phases, to support pattern matching and a “data-oriented” programming style. In this article, I explore when this programming style is beneficial, and why. I describe the syntax of pattern matching, as it has evolved, and show how it will be further enhanced in upcoming versions of Java. I warn of corner cases of the switch syntax where modern usage clashes

with legacy behavior. I conclude with a set of rules when to use pattern matching and “data orientation” now, how to prepare for future enhancements, and when to stay away.

Data-Oriented Programming

Often times, when you model a business process, your data model is so simple that object-oriented programming seems overkill. Consider a fee transaction in a bank. There is a reason for the fee, and the fee amount, modeled here for simplicity as a double. (In real life, use a `BigDecimal`!)

```
record Fee(String reason, double amount) {}
```

A record is perfect here. We don’t need setters, and we don’t need encapsulation. The data representation will never change.

Ok, maybe we want *some* object-orientation. Let’s say we have two more transaction types, deposits and withdrawals. Then we want all three of them to implement a common interface:

```
sealed interface BankTransaction permits Deposit, Withdrawal,
Fee {
    double amount();
}

record Deposit(double amount) implements BankTransaction {}
record Withdrawal(double amount) implements BankTransaction {}
record Fee(String reason, double amount) {}
```

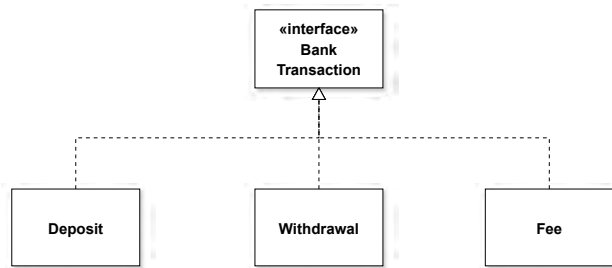
Author:



Cay Horstmann grew up in Northern Germany and attended the Christian-Albrechts-Universität in Kiel, a harbor town at the Baltic sea. I received a M.S. in computer science from Syracuse University, and a Ph.D. in mathematics from the University of Michigan in Ann Arbor. He taught computer science at San Jose State University for almost thirty years and held visiting appointments at universities in Germany, Switzerland, Vietnam, and Macao. Cay was a “serial entrepreneur” before that was a thing, including a stint as VP and CTO of a dot com startup that went from three people in a tiny office to a public company. In my copious spare time he writes books and develops online courses for beginning and professional programmers.

<https://www.linkedin.com/in/cay-horstmann-659a4b/>

Note that the interface is sealed. Only the listed types can implement it.



Once you have this setup—a sealed hierarchy of record types—you can use *pattern matching* to process it:

```
balance += switch (bankTransaction) {
    case Deposit(var amount) -> amount;
    case Withdrawal(var amount) -> -amount;
    case Fee(var amount, _) -> -amount;
};
```

Before continuing, a disclaimer. I have never written a banking application. It may well be that the problem domain has subtleties that make this data model seem naïve. A shout-out to Chris Kiehl's book. He shows you how you can find data orientation in real business scenarios.

Sometimes, sealed hierarchies of record types are called *algebraic* data types. The name comes from the fact that subtypes behave somewhat like sums, and records like products. The details are not all that illuminating, and I won't dwell on them. The important insight is that this particular structure makes pattern matching work. Each case picks up an alternative, and each record is deconstructed into its components.

The Pattern Matching Switch

Have another look at the `switch` from the preceding section. Note that it is an *expression* - it yields a value.

Each case deconstructs a record, binding variables to component values. However, the *wildcard* `_` doesn't bind a component.

The cases are *exhaustive* - they cover all possibilities of the sealed hierarchy

Finally, a bit of jargon: the value inside `switch (...)` is called the *selector*.

The `switch` syntax is easier than a sequence of `instanceof` tests, and it is *much* easier than the traditional OO approach, of adding a method to each type of the inheritance hierarchy.

Let us look at some syntactic variations. In the following example, the *guard* forbid overdrafts:

```
case Withdrawal(var amount) when amount > balance
    -> throw new BankTransactionException(...);
```

The compiler can't reason about exhaustiveness with guards. You need to add an unguarded case below.

```
case Withdrawal(var amount) -> -amount;
```

Let's add a target account to a withdrawal:

```
record Withdrawal(double amount, Account target) implements
    BankTransaction {}
```

Now we can forbid withdrawal to the same account:

```
case Withdrawal(var amount, Account(var id))
    when id.equals(currentAccount.id()) -> throw ...
```

Note the *nested* `Account` record in the pattern.

So far, you have seen *record patterns*. Another pattern, called a *type* pattern, binds a variable to entire selector:

```
case Fee f -> f.reason();
```

Patterns also work in `instanceof` expressions:

```
if (selector instanceof Fee f) ...
if (selector instanceof Fee(var _, var reason)) ...
```

At some point in the future, there will be patterns that work just like record patterns, but they can deconstruct instances of classes that are not records. Patterns will also be allowed in constructs other than `switch` and `instanceof`. Here is an example:

```
for (Map.Entry(var key, var val) : map.entrySet()) { ... }
// Not yet, but maybe in 2027?
```

Why Switch?

The `switch` syntax for pattern matching seems intuitive enough. The familiar `switch` statement has a number of cases, just like in our modern situation, where we apply various patterns.

However, there are important differences between classic and modern switches that you should keep in mind, so that you can use the modern feature effectively.

Java 1.0 adopted the classic `switch` statement from the C programming language. It is intended to effectively produce a jump table. Depending on the selector value, the code jumps to one of the cases. This example is from Kernighan and Ritchie's C programming book:

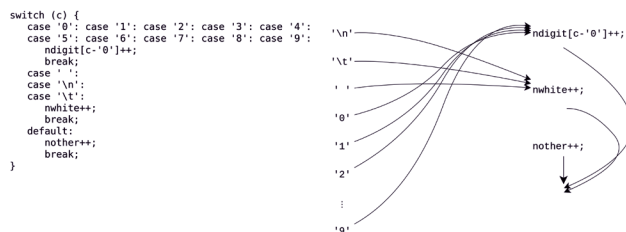

```

switch (c) {
    case ,0': case ,1': case ,2': case ,3': case ,4':
    case ,5': case ,6': case ,7': case ,8': case ,9':
        ndigit[c-'0']++;
        break;
    case , ,:
    case ,\n':
    case ,\t':
        nwhite++;
        break;
    default:
        nother++;
        break;
}

```

Since there are only a handful of cases in a narrow range, one can put the starting address of the code in each case into an array, making the jumps very efficient.

That is why the `break` statements are necessary. They cause jumps to the end of the statement. Without them, execution “falls through” to the next case, which is rarely desirable.



Note that here, the cases are unordered. If you reorder them,

the jump addresses are just rearranged in the table. In contrast, pattern matching is strictly sequential, looking at all cases from the top to the bottom until one of them matches.

The classic `switch` is a statement, whereas pattern matching is usually used as an expression. To bridge this gap, Java 14 introduced two other kinds of `switch`, for a total of four:

I suggest that you stay away from the “fall through” form and favor expressions over statements.

The Java 1 switch only works with types `int`, `short`, `char`, and `byte`. Java 5 introduced `String` and `enum` selectors. But what if they are `null`? Then an exception is thrown.

Nowadays, you can add a `case null`.

You have to be a bit careful about `null` in nested patterns. Consider again our `Fee` with a nested `Account` record:

```
case Withdrawal(var amount, Account(var id)) -> ...
```

If the selector is a new `Withdrawal(100, null)`, then a `MatchError` is thrown because the `id()` method cannot be applied to a `null` account.

In practice, try to avoid using `null` in nested records. It makes pattern matching quite tedious. You don’t want to add checks such as

```
case Withdrawal(_, Account a) when a == null -> ...
```

	EXPRESSION	STATEMENT
NO FALLTHROUGH	<pre> int numLetters = switch (seasonName) { case „Spring“ -> { IO.println(„spring time!“); yield 6; } case „Summer“, „Winter“ -> 6; case „Fall“ -> 4; default -> -1; }; </pre>	<pre> switch (seasonName) { case „Spring“ -> { IO.println(„spring time!“); numLetters = 6; } case „Summer“, „Winter“ -> numLetters = 6; case „Fall“ -> numLetters = 4; default -> numLetters = -1; } </pre>
FALLTHROUGH	<pre> int numLetters = switch (seasonName) { case „Spring“: IO.println(„spring time!“); case „Summer“, „Winter“: yield 6; case „Fall“: yield 4; default: yield -1; }; </pre>	<pre> switch (seasonName) { case „Spring“: IO.println(„spring time!“); case „Summer“, „Winter“: numLetters = 6; break; case „Fall“: numLetters = 4; break; default: numLetters = -1; } </pre>

Here Be Dragons

The rules for classic `switch` are more intricate than many programmers realize, and the pattern matching form tries to respect them as much as possible, while also trying to steer users to modern usage patterns.

Here is an example. Does this code compile?

```
enum Color { RED, YELLOW, GREEN };
boolean go(Color c) {
    switch (c) {
        case RED: return false;
        case YELLOW, GREEN: return true;
    }
}
```

No—the compiler complains about a “missing return statement.” It has been so since Java 5, since someone might add another `enum` constant, without recompiling this code.

Does this compile?

```
Boolean go(Color c) {
    switch (c) {
        case RED: return false;
        case YELLOW, GREEN: return true;
        case null: return null;
    }
}
```

Yup. `case null` makes it “enhanced.” And the rules are subtly different for enhanced switches.

Such finer points can be bewildering even for experts, and you can find quite a few “puzzlers” in blogs and presentations. These puzzlers can be fun for mastering those finer points, if indeed you are interested in such mastery.

But don’t let them scare you from using pattern matching! You won’t run into grief as long as you stay away from some pathological situations.

Record and type patterns are fine, as long as you stay away from nested `null` values. That applies to both `switch` and `instanceof` expressions.

You can safely use enumerations when they are a part of a sealed hierarchy. For example, one might define a part of a JSON hierarchy as follows:

```
public interface JSONObject {}
public interface JSONPrimitive extends JSONObject {}
public record JSONString(String value) extends JSONPrimitive {}
```

```
public enum JSONBoolean { FALSE, TRUE } implements
JSONPrimitive {}
...
```

As always, an `enum` is appropriate when there are only a finite number of values.

Now you can write expressions such as

```
boolean truthiness = switch (jsonPrimitive) {
    case JSONString(String v) -> return v.equals(",") ||
        v.equals(",false");
    case JSONBoolean.TRUE -> true;
    case JSONBoolean.FALSE -> false;
    ...
};
```

Here, the selector value belongs to a sealed hierarchy, and there is no possibility of conflict with classic rules. Things get murkier when the selector type is `Object` or a wrapper such as `Integer`.

```
Object obj = ...;
switch (obj) {
    case 0 ... // <cite>Error</cite>
    case JSONBoolean.TRUE ... // <cite>Ok</cite>
    ...
}

Integer n = ...;
switch (n) {
    case Integer i when i >= 0 ... // <cite>Doesn't dominate
        <code>case 0</code></cite>
    default ... // <cite>Doesn't dominate <code>case 0</code></cite>
    case 0 ... // <cite>Ok</cite>
    case null ... // <cite>Error: dominated by <code>default</code>
        <code> (even though not covered)</cite>
    ...
}
```

Fortunately, you are unlikely to run into examples of this kind in the wild.

Java 26 introduces primitive type patterns that check whether a value can be losslessly represented.

```
int n = ...;
if (n instanceof <code>byte b</code>) // <cite><code>b</code>
    set to <code>n</code> if -128 ≤ <code>n</code> ≤ 127</cite>
```

This can lead to an unhappy pitfall in a nested pattern:

```
sealed interface Shape {}
record Rect(double x, double y) implements Shape {}
record Square(double x) implements Shape {}
record Circle(double x) implements Shape {}

public List<Shape> dareToBeSquare(List<Shape> shapes) {
    return shapes.stream()
        .map(s -> switch(s) {
            case Rect(int x, _) -> new Square(x);
            default -> s;
        })
        .toList();
}
```

Did you notice the `case Rect(int x, _)`?

This case only matches rectangles whose x-coordinate is a `double` that can be safely cast to an `int`. With 0.5 or 2233445566.0, the default case is used.

The remedy is to use `var` inside patterns: `case Rect(var x, _)`.

Conclusion

Pattern matching works great with sealed hierarchies of records. You don't have to plan the perfect inheritance hierarchy, but simply use a switch whenever you need to extract values.

The `syntax` is generally regular and easy to use. The consistent use of pattern in switch and `instanceof` is a plus.

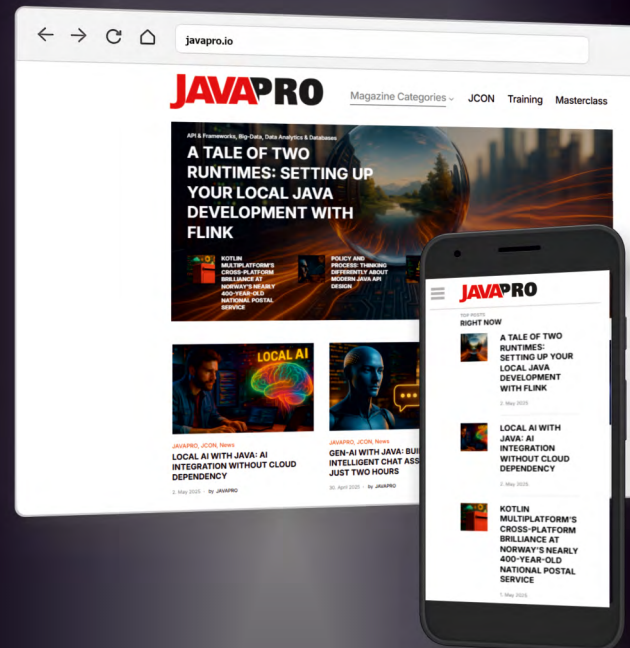
However, there are conflicts between modern and classic `switch` usage that can give rise to "sharp edges". These are a delight for puzzler authors, and mostly occur with very generic type tests and conversions among primitive types. Fortunately they don't usually apply to the context of data-oriented programming.

Stay away from the troublesome cases and enjoy the convenience of pattern matching! For now, think about where "algebraic data types" lurk inside your domain models. Future versions of Java will bring more opportunities.

-> Back to the Table of Content

JAVAPRO

**DISCOVER MORE GREAT
ARTICLES ONLINE:**



**BECOME A JAVAPRO
AUTHOR & BLOGGER !**



#JAVAPRO #COREJAVA

Data Oriented Programming with Java

Java and Object-Oriented Programming

Java has been with us for more than 30 years now. It was designed from the ground up as a purely object-oriented language. The mantra “everything is an object” (well, almost—primitives aside) has been deeply ingrained in Java’s DNA from day one. This object-oriented foundation helped Java become one of the most popular programming languages in the world, powering everything from enterprise applications to Android mobile apps.

Author:



Birgit Kratz is a freelance software developer and consultant with more than 25 years of experience in the Java ecosystem. Her focus and passion lie in applying agile development methods and promoting the software craftsmanship mindset. For many years, she has been a co-organizer of the Softwerkskammern in Cologne and Düsseldorf and helps organize the SoCraTes conference. In her free time, she is a passionate road cyclist.

<https://www.linkedin.com/in/birgitkratz/>

The OO paradigm has served us well, enabling us to build complex, maintainable systems.

But the language has matured and evolved over the last few years. We’ve learned that while object-oriented programming is powerful, it’s not always the best tool for every job.

Object-oriented programming in Java is built on fundamental principles: encapsulation, inheritance, polymorphism, and abstraction.

These principles allow us to model complex systems by bundling data with the methods that operate on that data, creating reusable and maintainable code through class hierarchies.

Object-oriented programming offers significant advantages: code reusability through inheritance, intuitive modeling of real-world entities, and clear separation of concerns through encapsulation. However, it also comes with drawbacks: deep inheritance hierarchies can become fragile and difficult to maintain, mutability can lead to unexpected side effects, and the tight coupling between data and behavior can make code harder to reason about and test.

Is there a way to avoid dealing with these disadvantages in certain cases?

To answer this question, we need to examine features that have been added to the JDK over the last few years. By combining these new features, we may be able to mitigate some of these traditional OO disadvantages.

Algebraic Data Types (ADT)

Oh... there you go. Algebra!? Seriously?!

Well, it's simpler than you might think, at least the parts that are relevant for us here.

In computer programming, an algebraic data type is a composite data type — a type formed by combining other types.

An algebraic data type is defined by two key constructions: a sum and a product.

These are sometimes referred to as “Or” and “And” types.

- Wikipedia

OK — that already sounds more like our daily business, right?

So let's take a closer look at what exactly **sum types** and **product types** are and how they can be represented in Java.

Sum Types - “OR”

A sum type represents a **choice** between different possibilities. It can hold a value that matches one of several defined variants. For example, a shape can be either a circle, a rectangle, or a triangle. Of course, there are many more shapes to choose from — if we define them as choices.

An example of a sum type in Java is an **Enum**:

```
enum Shape {
    RECTANGLE, CIRCLE, TRIANGLE;
}
```

Another, more powerful example is **Sealed Types**.

Sealed Types

Sealed types were introduced to Java with JDK 17. We can have sealed classes and sealed interfaces.

A sealed class or interface can be extended or implemented only by those classes and interfaces permitted to do so.

- JEP 409: Sealed Classes

```
sealed interface Shape permits Circle, Rectangle, Triangle {}
```

Here, only the types `Circle`, `Rectangle`, and `Triangle` are allowed to implement the `Shape` interface. If we had other types like `Hexagon` or `Oval`, they would not be permitted to implement the `Shape` interface.

This means we have a **controlled, finite set of types, known at compile time**, that can implement `Shape`. This is valuable information that the compiler can leverage.

Product Types — “And”

A product type combines multiple types. A value of a product type contains a value for each of its component types.

According to this definition, any immutable class that only holds data fields but provides no methods to alter that data can be considered a product type.

Since JDK 16, we even have a specific name for such a product type: **Record**.

Records

JDK 16 introduced records to Java.

Records are

carriers of immutable data and can be thought of as *nominal tuples*

- JEP 395: Records

which are simply tuples with names.

```
record Rectangle(int a, int b) {}
record Circle(int radius) {}
record Triangle(int a, int b, int angle) {}
```

Once a record is instantiated, the data it contains cannot be modified (not even by reflection!). This characteristic makes records perfect for modeling data as data — nothing more, nothing less.

If we want to change the data, we must create a new record instance with the modified data.

JEP 468: Derived Record Creation (also known as “Withers”) will make creating new records with data derived from existing records even easier.

The data in a record could be validated in the record's constructor, making it impossible to create invalid record instances.

Why Are Algebraic Data Types Interesting for Us?

The answer is: **Pattern Matching**.

Pattern Matching

Pattern matching is a programming mechanism that tests whether a value's structure and content match a specified pattern, then deconstructs (binds/extracts) its components for use if the match is successful.

Key behaviors of pattern matching include:

- **Testing:** Checking the exact structure match (e.g., is it a `Shape`?)
- **Binding:** Named captures of parts (e.g., `Circle c` binds `c`, allowing access to `c.radius`)
- **Exhaustiveness:** Ensuring all cases are covered (compiler-

checked for ADTs)

- **Guards:** Adding conditions (e.g., `Circle c when c.radius > 0`)

Since JDK 16, we have Pattern Matching for `instanceof`. Instead of writing:

```
if (o instanceof Shape) {
    Shape s = (Shape) o;
    ...
}
```

we can now write it in a much shorter, more concise way:

```
if (o instanceof Shape s) {
    ...
}
```

When we combine this with Pattern Matching for `switch`, which became final in JDK 21, we can handle different `Shape` implementations in a `switch` statement. Since our `Shape` type is a sealed interface, the compiler already knows all possible implementing types for the `switch` statement. The compiler can therefore check whether the `switch` statement is exhaustive (covers all possible cases). This removes the need for a default clause, which is considered **good practice**.

```
double area(Object o) {
    if (o instanceof Shape s) {
        return switch (s) {
            case Rectangle r -> r.a() * r.b();
            case Circle c -> Math.PI * Math.pow(c.radius(),
                2);
            case Triangle t -> 0.5 * t.a() * t.b() *
                Math.sin(t.angle());
        };
    }
    throw new IllegalArgumentException("Object is not a known
        shape.");
}
```

If we additionally combine this with JEP 440: Record Patterns, which was also finalized in JDK 21, our `switch` statement can be written even more concisely:

```
double area(Object o) {
    if (o instanceof Shape s) {
        return switch (s) {
            case Rectangle r -> r.a() * r.b();
            case Circle c -> Math.PI * Math.pow(c.radius(),
                2);
            case Triangle t -> 0.5 * t.a() * t.b() *
                Math.sin(t.angle());
        };
    }
}
```

```
throw new IllegalArgumentException("Object is not a known
    shape.");
}
```

Here the records are deconstructed, and its elements are bound to variables, which can then be used to calculate the `switch` result. If we would add a new shape (like `Hexagon`) to the allowed shapes, then the compiler would warn us that the `switch` expression does not cover all possible values.

This is only possible because we do not have a default branch in the `switch` statement.

Another important observation is the separation of the definition of the data from the operation on the data. The `area` method is not part of the sealed interface, but can reside somewhere else.

Additional enhancements for pattern matching include JEP 456: Unnamed Variables & Patterns in JDK 22 and Primitive Types in Patterns, `instanceof`, and `switch`, which is still in preview.

What Is Data-Oriented Programming (DOP)?

So why is all this talk about algebraic data types, pattern matching, records, sealed classes, and so on?

Perhaps we can combine these features in such a way that their sum adds up to more than its individual parts. In fact, when we combine sealed classes, records, pattern matching for `switch`, and record patterns, we have all the ingredients for **Data-Oriented Programming**.

Data-Oriented Programming should follow specific principles that shift our focus from objects to data.

The Four Principles of Data-Oriented Programming

According to Brian Goetz's seminal work on this topic [1], Data-Oriented Programming follows four key principles:

1. Model the data, the whole data, and nothing but the data
2. Data is immutable
3. Validate at the boundary
4. Make illegal states unrepresentable

Let's examine how Java's modern features support each of these principles.

Principle 1: Model the Data, the Whole Data, and Nothing but the Data

This principle emphasizes separating data modeling from behavior. Instead of bundling data with methods that operate on it (as in traditional OOP), we model data structures as pure data carriers.

This can be achieved through a combination of sealed classes and records. Sealed classes define the set of possible data variants, while records define the structure of each variant. This separation allows us to add new operations on the data without modifying the data definitions themselves.

```
sealed interface Shape permits Circle, Rectangle, Triangle {}
record Circle(int radius) implements Shape {}
record Rectangle(int a, int b) implements Shape {}
record Triangle(int a, int b, int angle) implements Shape {}

// When the permitted subclasses are small in size and number,
// it may be convenient to declare them in the same source file as
// the sealed class.
// When they are declared in this way, the sealed class may
// omit the permits clause and the Java compiler will infer the
// permitted subclasses from the declarations in the source file.
sealed interface Shape {
    record Circle(int radius) implements Shape {}
    record Rectangle(int a, int b) implements Shape {}
    record Triangle(int a, int b, int angle) implements Shape
    {}
}
```

Principle 2: Data Is Immutable

Records are immutable by design. Once created, their component values cannot be changed. This eliminates entire categories of bugs related to unexpected state mutations and makes code easier to reason about, especially in concurrent contexts. The immutability of records ensures that data flows through our programs predictably, without hidden side effects.

Principle 3: Validate at the Boundary

Data should be validated when it enters the system (at the boundary), not throughout the codebase. This can be done before records are instantiated or within a record's compact constructor:

```
record Circle(int radius) implements Shape {
    public Circle {
        if (radius <= 0) {
            throw new IllegalArgumentException("Radius must be
            positive");
        }
    }
}
```

By validating in the constructor, we ensure that invalid Circle instances can never exist in our system.

Principle 4: Make Illegal States Unrepresentable

This principle encourages us to design our data types so that invalid states simply cannot be represented in code.

Through validation in record constructors, we prevent the creation of invalid data.

Additionally, pattern matching with switch on sealed types ensures exhaustiveness—the compiler won't allow us to miss cases.

When we avoid using a default clause, the compiler will warn

us if we add a new variant to a sealed type without handling it in existing switch statements. This way we don't have to search through the code ourselves.

```
// Compiler ensures all Shape variants are handled
String stringify (Shape s) {
    return switch (s) {
        case Circle c -> „It's a circle“;
        case Rectangle r -> „It's a rectangle“;
        case Triangle t -> „it's a triangle“;
        // No default needed - compiler knows this is
        exhaustive
    };
}
```

When Is Data-Oriented Programming Useful?

Data-Oriented Programming shines in several scenarios:

Domain Modeling: When modeling complex domains with well-defined data structures and operations, DOP provides clarity and type safety. Financial systems, medical records, and configuration data are excellent candidates.

Data Transformation Pipelines: Systems that primarily transform data from one form to another benefit greatly from immutable data structures and pattern matching. ETL processes, compilers, and data processing frameworks are natural fits.

APIs and Data Transfer: When designing APIs or working with external data sources, immutable records provide clear contracts and prevent accidental mutations.

Concurrent Systems: The immutability guarantee of DOP makes it easier to write correct concurrent code without complex synchronization.

Algebraic Problem Domains:

Any domain that naturally maps to algebraic data types—expression evaluators, abstract syntax trees, state machines, or game logic—becomes more expressive with DOP. In many cases, DOP can even replace traditional design patterns like Visitor or eliminate the need for complex recursive methods.

Let's look at two concrete examples that demonstrate these advantages.

Example 1: State Machine for Order Processing

Consider an order processing system where orders transition through various states. With DOP, we can model this state machine clearly and handle state transitions safely:

```
sealed interface OrderState permits Pending, Confirmed,
    Shipped, Delivered, Cancelled {}
```

```

record Pending(String orderId, LocalDateTime createdAt)
implements OrderState {}
record Confirmed(String orderId, LocalDateTime confirmedAt,
String paymentId) implements OrderState {}
record Shipped(String orderId, LocalDateTime shippedAt, String
trackingNumber) implements OrderState {}
record Delivered(String orderId, LocalDateTime deliveredAt,
String signature) implements OrderState {}
record Cancelled(String orderId, LocalDateTime cancelledAt,
String reason) implements OrderState {}

class OrderProcessor {
    // Process order based on current state - exhaustive and
    type-safe

    static String getStatusMessage(OrderState state) {
        return switch (state) {
            case Pending(var id, var created) ->
                „Order „ + id + „ is pending (created: „ +
                created + „)“;
            case Confirmed(var id, var confirmed, var paymentId)
                ->
                „Order „ + id + „ confirmed with payment „ +
                paymentId;
            case Shipped(var id, var shipped, var tracking) ->
                „Order „ + id + „ shipped - tracking: „ +
                tracking;
            case Delivered(var id, var delivered, var
                signature) ->
                „Order „ + id + „ delivered and signed by „ +
                signature;
            case Cancelled(var id, var cancelled, var reason)
                ->
                „Order „ + id + „ cancelled: „ + reason;
        };
    }

    // State transitions with validation
    static OrderState confirmOrder(OrderState state, String
    paymentId) {
        return switch (state) {
            case Pending(var id, var created) ->
                new Confirmed(id, LocalDateTime.now(),
                paymentId);
            case Confirmed c -> c; // Already confirmed
            case Shipped _, Delivered _, Cancelled _ ->
                throw new IllegalStateException(„Cannot confirm
                order in state: „ + state);
        };
    }

    // Check if order can be cancelled
    static boolean canCancel(OrderState state) {
        return switch (state) {
            case Pending _, Confirmed _ -> true;
            case Shipped _, Delivered _, Cancelled _ -> false;
        };
    }
}

```

Compare this to a traditional OOP approach using the State pattern, which would require:

- An abstract OrderState class with abstract methods for each operation
- Multiple concrete state classes, each implementing all operations
- Complex state transition logic scattered across multiple classes
- Potential for runtime errors if a state doesn't handle a transition properly
- Many code points to fix when a new state is added, but no help from the compiler

With DOP, the compiler ensures we handle all states exhaustively, and all the logic for a particular operation is in one place, making it easier to understand and maintain.

Example 2: JSON Parsing Without Visitor Pattern or Recursion

JSON parsing is a classic use case for the Visitor pattern or recursive methods. With DOP and pattern matching, we can achieve the same result more elegantly:

```

// Model JSON structure as an ADT
sealed interface JsonValue {}

record JsonString(String value) implements JsonValue {}
record JsonNumber(double value) implements JsonValue {}
record JsonBoolean(boolean value) implements JsonValue {}
record JsonNull() implements JsonValue {}
record JsonArray(List<JsonValue> elements) implements
JsonValue {}
record JsonObject(Map<String, JsonValue> fields) implements
JsonValue {}

class JsonProcessor {
    // Convert JSON to string representation - no recursion
    needed!
    public String stringify(JsonValue json) {
        return switch (json) {
            case JsonString(var s) -> „““ + s + „““;
            case JsonNumber(var n) -> String.valueOf(n);
            case JsonBoolean(var b) -> String.valueOf(b);
            case JsonNull() -> „null“;
            case JsonArray(var elements) -> „[„ +
                elements.stream()
                    .map(this::stringify)
                    .collect(Collectors.joining(„, „)) +
                „]“;
            case JsonObject(var fields) -> „{„ +
                fields.entrySet().stream()
                    .map(e -> „““ + e.getKey() + „“: „ +
                    stringify(e.getValue()))
                    .collect(Collectors.joining(„, „)) +
                „}“;
        };
    }
}

```

```
// Find value at a path (e.g., „user.address.city“)
public JsonValue findAtPath(JsonValue json, String path) {
    String[] parts = path.split("\\.");
    JsonValue current = json;

    for (String part : parts) {
        current = switch (current) {
            case JsonObject(var fields) ->
                fields.getOrDefault(part, new JsonNull());
            case JsonArray(var elements) -> {
                try {
                    int index = Integer.parseInt(part);
                    yield index < elements.size() ?
                        elements.get(index) : new JsonNull();
                } catch (NumberFormatException e) {
                    yield new JsonNull();
                }
            }
            default -> new JsonNull();
        };
    }
    return current;
}

// Example usage
class JsonExample {
    public static void main(String[] args) {
        JsonValue user = new JsonObject(Map.of(
            „name“, new JsonString(„Alice“,),
            „age“, new JsonNumber(30),
            „address“, new JsonObject(Map.of(
                „city“, new JsonString(„Berlin“,),
                „zipCode“, new JsonString(„10115“)
            )),
            „hobbies“, new JsonArray(List.of(
                new JsonString(„reading“,),
                new JsonString(„cycling“)
            ))
        ));

        JsonProcessor processor = new JsonProcessor();
        System.out.println(processor.stringify(user));
        // Output: {„name“: „Alice“, „age“: 30.0, „address“:
        // {„city“: „Berlin“, ...}, ...}

        JsonValue city = processor.findAtPath(user, „address.
        city“);
        System.out.println(city); // JsonString[value=Berlin]
    }
}
```

Advantages over Traditional Approaches:

Traditional approaches would require either:

- Visitor Pattern: Creating a `JsonVisitor` interface with methods for each type, then implementing concrete visitors for each operation. This spreads the logic across multiple classes and makes adding new operations cumbersome.
- Recursive Methods with instanceof: Writing deeply nested if-else chains with manual type checks and casts, which is error-prone and verbose.

With DOP:

- All logic for an operation is in one switch expression
- The compiler ensures exhaustiveness—we can't forget a case
- Pattern matching eliminates boilerplate casting and null checks
- Adding new operations is straightforward—just write a new method with a switch
- The code clearly expresses the structure and intent

Is Data-Oriented Programming The New Hammer For All Our Problems?

No, data-oriented programming is not a silver bullet, and it's not meant to replace object-oriented programming entirely.

When to Stick with OOP:

Traditional OOP remains appropriate for systems where behavior and state are tightly coupled, such as UI frameworks, game engines with complex entity behaviors, or systems that model stateful processes. If your domain naturally involves objects that change state over time and have rich behavioral interfaces, OOP is still the right choice.

When to Use DOP:

Data-Oriented programming excels when your primary concern is modeling and transforming data structures. It's particularly valuable in functional-style pipelines, parsers, compilers, configuration systems, and data-centric business logic, usually in smaller systems.

The Hybrid Approach:

In practice, most modern Java applications benefit from a hybrid approach. Use DOP for data modeling and transformation logic, while employing traditional OOP for stateful components, frameworks, and infrastructure code. The key is choosing the right tool for each problem domain.

Java's evolution with records, sealed classes, and pattern matching gives us the flexibility to choose the most appropriate paradigm for each part of our system. This isn't about replacing one approach with another—it's about expanding our toolkit and making more informed design decisions.

Conclusion

Data-oriented programming represents a significant evolution in how we can write Java code. By combining sealed classes, records, and pattern matching, we gain powerful tools for modeling data clearly and safely. The four principles—modeling pure data, ensuring immutability, validating at boundaries, and making illegal states unrepresentable—guide us toward more robust and maintainable systems.

As Java continues to evolve, features like derived record creation (withers) and enhanced pattern matching will make Data-Oriented programming even more ergonomic. The language is giving us new ways to think about old problems, and it's worth exploring how these techniques can improve your codebase.

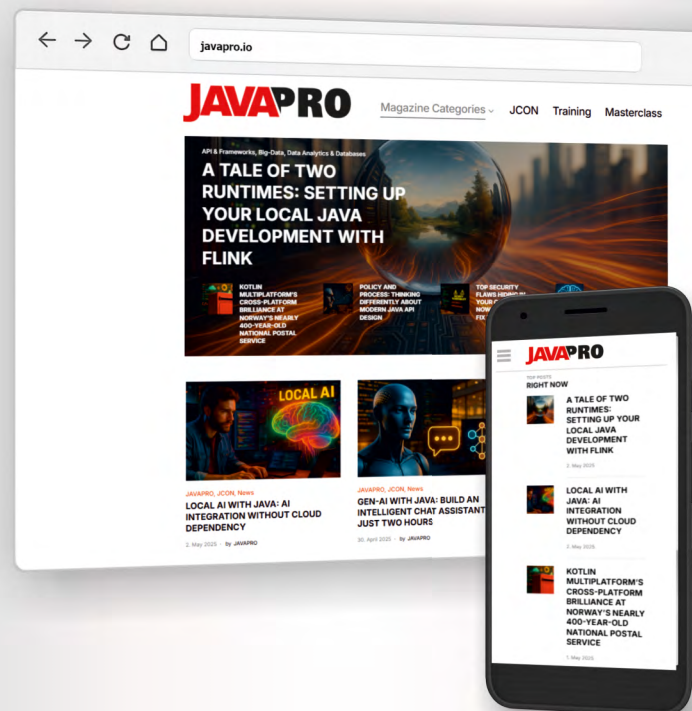
Resources

1. Data Oriented Programming in Java by Brian Goetz and Daniel Bryant
2. Data-Oriented Programming in Java by Gavin Bierman
3. Data-Oriented Programming in Java – Version 1.1 by Nikolai Parlog
4. Harnessing Java 21 for Data Oriented Programming by Ken Kousen
5. Data Oriented Programming in Java 21 by Nicolai Parlog

-> Back to the Table of Content

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

www.javapro.io



#JAVAPRO #COREJAVA

Java 26 Adopts HTTP/3 with the Evolution of the HttpClient

The Future of Communication Between Microservices

Modern distributed architectures, such as microservices, are deeply intertwined with the evolution of network protocols. As

microservice architectures have become the dominant approach for building scalable and resilient applications, service-to-service communication efficiency has become a significant concern.

Although HTTP/1.1 and HTTP/2 provided significant enhancements, such as persistent connections and request multiplexing, they were still inherently bound by their reliance on TCP. In high-latency or unstable networks, these limitations can still result in degraded performance, increased tail latency, and reduced resiliency under load.

With JEP 517, Java 26 introduces HTTP/3 support in HttpClient, enabling QUIC-based communication. Let's understand what changes with HTTP/3.

What Changes with HTTP/3

HTTP has historically been optimized at the application layer while relying on TCP as its transport protocol for decades. Despite improvements such as persistent connections, HTTP/1.1 achieved scalability mainly by opening multiple parallel connections, increasing overhead and congestion.

HTTP/2 took a significant step forward when it introduced a binary protocol, request multiplexing, HPACK header compression, and server push. Multiple logical streams can now coexist over the same connection, reducing connection churn and increasing throughput. But despite these improvements, HTTP/2 is still mostly bottlenecked by TCP.

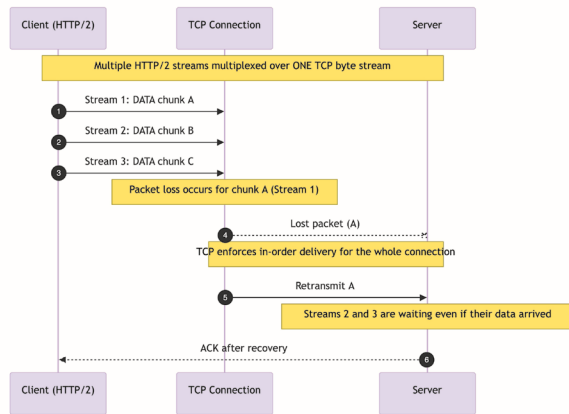
Author:



Wanderson Xesquevixos is a passionate software engineer with over 20 years of experience and author of *Software Architecture with Spring* (Packt, 2025). Working with Java since 2001 and Spring since 2006, he holds Java and AWS certifications. He specializes in software architecture, microservices, and cloud-native systems, helping teams build scalable solutions while applying a philosophical lens to technical decision-making, constantly questioning the "why" behind each choice and its trade-offs. Wanderson holds a degree in Computer Science and MBAs from USP in Data Science and Software Engineering. He is currently preparing to pursue a master's in Computer Science focused on Distributed Systems and Cloud Computing

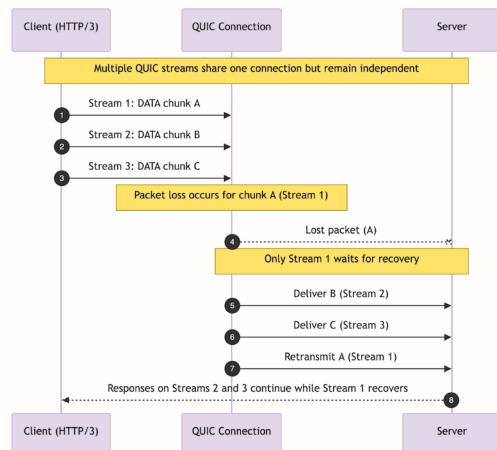
<https://www.linkedin.com/in/wandersonxs/>

Figure 1 illustrates **Head-of-Line (HoL) blocking** in HTTP/2 due to TCP's in-order delivery guarantees.



- **Steps 1, 2, and 3:** The client sends multiple HTTP/2 streams (Stream 1, 2, and 3). Each stream carries a chunk of data (A, B, C), and all of them are multiplexed over a single TCP connection. Although HTTP/2 treats streams as logically independent, TCP sees everything as one ordered sequence of bytes.
- **Step 4:** After a packet loss occurred (a TCP packet containing DATA chunk A (Stream 1) is lost somewhere in the network), TCP enforces in-order delivery, because TCP guarantees ordered delivery, the receiver cannot deliver any data that comes after the missing packet, even if chunks B and C (Streams 2 and 3) arrive successfully.
- **Step 5:** Retransmission blocks other streams. TCP retransmits the lost packet A. During this time, Streams 2 and 3 are blocked, even though their data is already available. **This is the Head-of-Line blocking effect.**
- **Step 6:** Recovery and acknowledgment. Once packet A is retransmitted and acknowledged, TCP can resume delivery, and all streams continue.

HTTP/2 solves application-layer blocking (from HTTP/1.1), but cannot avoid transport-layer blocking caused by TCP. A single lost packet affects all multiplexed streams, which is one of the core motivations for HTTP/3 over QUIC, where streams are independent at the transport layer, as illustrated in **Figure 2**.



- **Steps 1, 2, and 3:** The client sends multiple HTTP/3 streams (Stream 1, 2, and 3). Each stream carries a chunk of data (A, B, C), and all of them are multiplexed over a single QUIC connection. Although the streams share the same connection, each QUIC stream is independently ordered and flow-controlled.
- **Step 4:** A packet loss occurs for DATA chunk A (Stream 1). Unlike TCP, QUIC does not enforce global in-order delivery across all streams. Only the affected stream (Stream 1) is impacted by the loss.
- **Step 5:** Streams 2 and 3 continue delivering data. DATA chunks B and C are successfully delivered to the server without waiting for the recovery of Stream 1, because QUIC provides stream-level independence.
- **Step 6:** The lost packet for DATA chunk A is retransmitted, but this retransmission is scoped only to Stream 1, not to the entire connection.
- **Step 7:** Stream 1 resumes once its missing data is recovered, while Streams 2 and 3 have already progressed.
- **Step 8:** Responses on Streams 2 and 3 continue uninterrupted while Stream 1 recovers, demonstrating that **Head-of-Line blocking is eliminated at the transport layer**.

HTTP/3, by running over QUIC, **removes transport-layer Head-of-Line blocking by ensuring that packet loss affects only the stream involved**, not the entire connection. This is the fundamental difference from HTTP/2 and TCP, and the core motivation behind HTTP/3. Having understood HTTP/3, its advantages, and its evolution, let's explore JEP 517.

Understanding JEP 517 — HTTP/3 for the HTTP Client API

JEP 517 adds native HTTP/3 support to the Java platform by extending the existing `java.net.http.HttpClient`. Rather than adding a new API, JEP 517 updates the HTTP client introduced in Java 11.

This approach follows a long-standing Java design principle: delivering significant runtime improvements without breaking existing applications.

In a technical sense, JEP 517 incorporates an HTTP/3 stack that uses QUIC directly in the JDK. In other words, the `HttpClient` becomes protocol-aware, using HTTP/1.1, HTTP/2, or HTTP/3 through the same abstractions.

The selection of protocols is made on the fly based on server availability, network conditions, and client configuration, to prevent applications from becoming protocol-blind by implementing newer transports when available.

The client automatically attempts to use the highest-level protocol supported by both endpoints per request, reverting to HTTP/2 or HTTP/1.1 as needed. This negotiation happens without requiring explicit branching logic in the application code.

As a result, existing applications continue to use the same request and response APIs, while benefiting from improved packet-loss

resilience, faster connection establishment, and lower tail latency in HTTP/3-enabled environments (which will be very useful to microservices and cloud-native apps).

Finally, JEP 517 provides full transparency by including the negotiated protocol in the `HttpResponse`, enabling logging, traceability, and metric collection. Such visibility further enables performance analysis and control of implementation, notably in heterogeneous environments where HTTP/3 availability varies.

Using HTTP/3 with HttpClient

First, let's certify that our JDK supports HTTP/3. At the time writing this article I was using the following JDK: `java 26.ea.28-open`. The following command installs it using the SDKMAN.

```
sdk install java 26.ea.28-open
```

Check whether you've set the installed JDK with `java --version`, then check whether it supports HTTP/3 typing:

```
jshell
import java.net.http.*;
System.out.println(java.util.Arrays.toString(HttpClient.
Version.values()));
```

You should get the response:

```
[HTTP_1_1, HTTP_2, HTTP_3]
```

The following code is straightforward and presents the use of HTTP/3 with `HttpClient`.

```
static void main(String[] args) throws IOException,
InterruptedException {
    // Prefer HTTP/3; automatically fall back to HTTP/2 or
    HTTP/1.1
    // if unavailable
    HttpClient client = HttpClient.newBuilder()
        .version(HttpClient.Version.HTTP_3)
        .build();

    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://jsonplaceholder.
        typicode.com/todos/1"))
        .header("Accept", "application/json")
        .version(HttpClient.Version.HTTP_3)
        .GET()
        .build();

    HttpResponse<String> response = client.send(request,
        HttpResponse.BodyHandlers.ofString());

    IO.println("Status: ", + response.statusCode());
    IO.println("Negotiated protocol: ", + response.version());
    IO.println("Body:\n" + response.body());
```

This code configures an `HttpClient` to prefer HTTP/3 by setting `version(HttpClient.Version.HTTP_3)`. At runtime, the client

attempts to establish an HTTP/3 connection and transparently falls back to HTTP/2 or HTTP/1.1 if QUIC is unavailable.

The request reiterates the HTTP/3 preference at the request level, illustrating that protocol selection can be applied selectively. After execution, the negotiated protocol is obtained from the response, allowing verification of whether HTTP/3 was actually used.

Let's run the code with the following command:

```
java \
    -Djdk.httpclient.HttpClient.log=all \
    src/main/java/com/Main.java
```

After the code execution, you can find in the log something such as:

```
HTTP3: HTTP/3 connection created for QuicClientConnection(1)
QUIC: QuicClientConnection(1) handshake completed successfully
HEADERS: H3 HEADERS FRAME (stream=0)
... alt-svc: h3=":443"; ma=86400
Negotiated protocol: HTTP_3
```

The log output confirms that the request was executed over HTTP/3. The creation of a QUIC-based connection and the successful QUIC handshake indicate that the client established an HTTP/3 transport rather than falling back to TCP.

The presence of HTTP/3 headers frames shows that the request was encoded using the HTTP/3 protocol, and the `alt-svc` header confirms that the server advertises HTTP/3 support.

Finally, the negotiated protocol reported as `HTTP_3` provides application-level confirmation that the exchange completed over HTTP/3.

Conclusion

HTTP/3 support in Java 26 represents a significant evolution of the `HttpClient` API, aligning the JVM with modern transport mechanisms already adopted by browsers, CDNs, and edge platforms.

By transparently integrating QUIC into the existing HTTP client, Java enables applications to benefit from reduced latency, improved resilience to packet loss, and the elimination of head-of-line blocking without invasive code changes.

The ability to prefer HTTP/3 while retaining automatic fallback allows safe adoption in heterogeneous environments, enabling microservices, API integrations, and client-server interactions to progressively leverage HTTP/3's performance characteristics while preserving compatibility and operational stability.

References and Source Code

JEP 517: <https://openjdk.org/jeps/517>

Source code: <https://github.com/wandersonxs/JEP-517-HTTP3>

OSI Protocol: <https://aws.amazon.com/what-is/osi-model/>

-> Back to the Table of Content



#JAVAPRO #COREJAVA

Java 26 GC: Making Data Streaming Faster and More Reliable in Containers

Authors:



Varsha Sharma is an Engineer with experience in data streaming and cloud-native platform. She spends her time investigating how decentralized AI and real-time data can empower developers.

<https://www.linkedin.com/in/varsha-sharma-2553b9162/>



Shubham Goel is a data industry expert with 10 years of experience, specializing in data streaming for the past four years. As part of Confluent's Global Technical Support team, he helps organizations optimize and troubleshoot real-time data streaming solutions. With expertise in Kafka, Kubernetes, and cloud-native architectures, Shubham supports enterprises in scaling and modernizing their streaming platforms.

<https://www.linkedin.com/in/shubham-goel-a4232021/>

Imagine running a Java application that processes massive amounts of data in real time.

Example financial transactions flowing through Kafka or analytics pipelines in Flink. These apps need to be fast and consistent, but there's a built-in challenge: **Garbage Collection (GC)**.

What Is Garbage Collection?

Java automatically cleans up unused memory. However the application sometimes has to pause completely. These are called "Stop-The-World" pauses. Even a pause of just a few milliseconds can cause:

- Delayed data processing
- Timeouts and dropped connections
- Missed deadlines for real-time operations

For systems that handle thousands of messages per second, these pauses are a serious problem.

What Java 26 Brings to the Table

Java 26 introduces major improvements to the G1 Garbage Collector (the default cleanup mechanism in Java). This is under-the-hood optimizations that make the garbage collection process

smarter and less disruptive.

Four Key Benefits for Data Streaming Apps

1. Smoother Performance (Lower Tail Latency)

Those occasional slow responses that make your 99th percentile latency look bad? They happen less often. Your application responds more consistently, which matters when you're processing time-sensitive data.

2. More Work Gets Done (Higher Throughput)

Because the application spends less time paused for garbage collection, it can process more messages in the same amount of time. Think of it like reducing coffee breaks at work, more gets accomplished overall.

3. Predictable Behavior

Instead of wondering "when will the next GC pause hit?", you get more predictable performance patterns. This makes it easier to plan capacity, set realistic SLAs, and avoid surprises during peak traffic.

4. Better Performance in Containers

This is particularly important for modern deployments. When running in Kubernetes pods with limited CPU and memory, the improved GC means:

- Fewer health check failures (Kubernetes won't think your app is dead during a GC pause)
- Lower CPU usage for the same workload
- More efficient use of container resources

Real-World Example

Let's say you're running a Kafka Streams microservice in Kubernetes that processes stock market trades:

On Java 21:

- Occasional GC pauses cause the service to miss Kubernetes readiness checks
- CPU spikes during garbage collection
- Some trades get processed late

On Java 26:

- Readiness probe failures drop noticeably
- CPU usage stays more stable
- Trades flow through more consistently

You can measure this difference directly by comparing the same application running on both versions in identical container environments.

Here's a sample demo application deployed in k8s :Kafka Streams Performance Comparison.

Hands on environment: <https://killercode.com/varsharma/scenario/java26-kafka-demo>

Source code for demo application : <https://github.com/varsharma-12/scenario-examples/blob/main/java26-kafka-demo/kafka-streams-demo.tar.gz>

Why This Matters

For teams running data streaming platforms, this translates to:

- **More reliable systems** without code changes
- **Better resource utilization** in cloud environments
- **Easier compliance** with strict latency requirements
- **Lower operational costs** from more efficient container usage

Java 26's GC improvements solve a real pain point for anyone running high-throughput, latency-sensitive Java applications in containers. You get faster, more predictable performance without having to rewrite your code or become a GC tuning expert.

To see these improvements in action, try the Kafka Streams Performance Comparison demo.

-> Back to the Table of Content



#JAVAPRO #COREJAVA

1BRC – The First 80% of the Journey



The One Billion Row Challenge (1BRC) started as a way to kill time during the 2023/2024 holidays, asking a simple question: How fast can Java crunch temperature values from a text file and calculate the min, mean, and max per weather station? The catch is that the file contains one billion rows, totaling approximately 13.8 GB. Thomas Wuerthinger, Quan Anh Mai, and Alfonso Peterssen achieved the best result of 1.5 seconds on 8 cores. That was insanely and unexpectedly fast. But when you take a closer look at the code, you will see that it is beyond your daily dose of Java. It is a complex, single-use-case-oriented implementation, skipping safety features, and most of the tunings applied cannot be easily reused.

This article will show you how to get 80% of the result with simpler and more maintainable code. It will also focus on a single-threaded approach to keep us focused. You can use most of the ideas in your own code easily if you need such tuning. But

yes, we will also make decisions that a clean code coach would consider big no-nos.

The Challenge

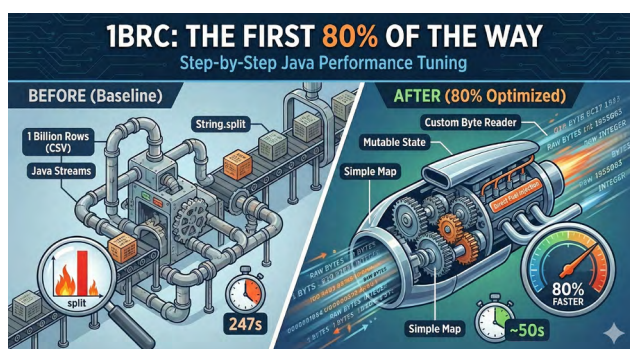
At the end of 2023, when everyone was busy with the holidays, Gunnar Morling got bored. So he came up with a fun little project to read one billion rows from a CSV file as fast as possible.

Author:

René Schwietzke is a seasoned IT professional with over two decades of experience in performance tuning and performance measurements, having started his journey with Java version 1.0. During his tenure at various e-commerce and quality assurance companies, he honed his expertise in optimizing and fine-tuning large-scale applications. His academic background includes a Master's degree in Computer Science, and he actively shares his knowledge by giving lectures at universities and presenting at international software development conferences. He is also a sought-after speaker for training sessions on Java, garbage collection, memory management, quality assurance, and testing.



<https://www.linkedin.com/in/reneschwietzke/>



Recognizing that performance challenges are best tackled by the community, he turned it into a competition. He wanted to see how fast Java code could become with the help of other Java nerds.

Imagine a data file with one billion rows, each row containing the name of a weather station, aka city and a temperature value. The goal is to calculate the min, mean, and max temperature per weather station as fast as possible.

```
Zagreb;19.7
Tromsø;-3.9
Gangtok;-1.7
Houston;32.4
Zürich;13.8
```

The final output was supposed to look like this. Out of pure laziness, Gunnar went with `TreeMap::toString`. All numbers are properly rounded and formatted to one decimal place.

```
# This is TreeMap::toString
# Data shown here is formatted for screen
{
  Abha=5.0/18.0/27.4, Abidjan=15.7/26.0/34.1,
  Abéché=12.1/29.4/35.6, Accra=14.7/26.4/33.1,
  Addis Ababa=2.1/16.0/24.3, Adelaide=4.1/17.3/29.7,
  ...
  Willemstad=-7.9,28.2,57.7, Winnipeg=-28.1,3.0,39.4,
  Wrocław=-27.5,9.6,50.2, Xi'an=-22.5,14.0,49.1,
  Yakutsk=-42.2,-8.8,26.4, Yangon=-2.6,27.6,59.0,
  Yaoundé=-10.1,23.6,58.3, Yellowknife=-38.4,-4.0,40.0,
  Zagreb=-25.3,10.6,48.4, Zanzibar City=-5.0,26.1,61.2,
  Zürich=-20.9,9.4,43.5, Ürümqi=-26.8,7.4,43.7,
  İzmir=-15.8,18.0,50.1
}
```

Several things are important to note.

- The file is encoded in UTF-8 and contains characters outside the ASCII range.
- The file is fully valid. No BOM, no invalid characters, no trailing spaces, no empty lines.
- We can have up to 10,000 stations, but the base version of the file has only 413.
- All line endings are `\n`.
- All temperatures are within the -99.9 to 99.9 range. Always formatted with one decimal place.
- No station exceeds a length of 100 bytes.

For the competition itself, one could use up to 8-cores. The file was in the file cache because the measurement was done five times and the best three were averaged. But this is not important for our 80% journey.

The following results have been achieved. Once again, these are my measurements and not the official results. Also, as a reminder, we will avoid multi-threading for now to keep things as simple as possible. The baseline, suggested by Gunnar Morling, is straightforward Java code that does the job.

The Results

TEST	CORES	TIME DO INTEL ¹	TIME HETZNER AMD ²
Baseline	1	247.6 s	152.1 s
Baseline	8	111.3 s	81.4 s
Thomas Wuethinger	1	8.9 s	7.3 s
Thomas Wuethinger	8	1.2 s	1.6 s
Today's 20% Goal	1	49.5 s	30.4 s

The Remeasured 1BRC Results

1. Digital Ocean, Amsterdam, Dedicated Machine, Intel CPU-optimized, 16 core, 32 GB, Jan 2025
2. Hetzner, Finland, Dedicated Machine, AMD, 8 core, 32 GB, Jan 2025

You can already see where we want to go today: our 80% goal or, rather, our 80% tuning. Certainly, this is not a record-breaking number, but most of the tuning and profiling can be applied to your daily work.

You might not want to brute-force apply the tunings from here to your code. You have to have a problem first. Just because you can, doesn't mean you should. Clean and maintainable code always comes first.

All runtime numbers have been measured by me on 8-core cloud machines at Hetzner (AMD CPUs) and Digital Ocean (Intel CPUs). This means your mileage might vary, and because machines and setups change, I will likely not be able to reproduce the exact numbers when I run that today. You always have to measure the entire series to get a good picture of the runtimes.

When you look at the winning solution, you can see that it uses `Unsafe`, GraalVM native builds, a ton of bit magic, and many other things that you wouldn't use in your everyday code. Impressive and interesting, but with limited practical reuse. GraalVM native images excluded.

If you want to dive into the final solution, try to understand the branchless parsing (also known as SWAR – SIMD Within A Register) because it is the key driver for performance compared to other solutions as well as ours today.

Our Goal: 80% Faster

We don't want to be as fast as the winning solution (which is about 9 seconds for a single thread), we just want to end up with a fifth of the runtime, shaving off 80% of the runtime.

You can find all the code at <https://github.com/rschwietzke/1brc-the-first-80-meters>. I will either link to the implementation or just mention the class by name.

Just to detour quickly. The original 1BRC uses single Java files and always runs them several times. To make measurements easier and also allow for better learning regarding Java JIT compilation, I wrote a small framework for executing each solution, including measurement, warming, and checksumming, not to forget batch execution.

Step 0: The Baseline

The baseline is the original solution by Gunnar Morling. As you will see, we have a plain stream implementation that uses features Java provides, such as collecting and grouping. I leave the interpretation to you or just read about that in all the other 1BRC articles out there.

This and any following code has been modified to fit the article. Please always refer to the repository for the full code. Formatting is likely inconsistent too.

```
Collector<Measurement, MeasurementAggregator, ResultRow>
collector =
    Collector.of(MeasurementAggregator::new,
        (agg, m) -> {
            agg.min = Math.min(agg.min, m.value);
            agg.max = Math.max(agg.max, m.value);
            agg.sum += m.value;
            agg.count++;
        }, (agg1, agg2) -> {
            var res = new MeasurementAggregator();
            res.min = Math.min(agg1.min, agg2.min);
            res.max = Math.max(agg1.max, agg2.max);
            res.sum = agg1.sum + agg2.sum;
            res.count = agg1.count + agg2.count;

            return res;
        }, agg -> {
            return new ResultRow(agg.min, (Math.round(agg.sum
                * 10.0) / 10.0) / agg.count, agg.max);
        });

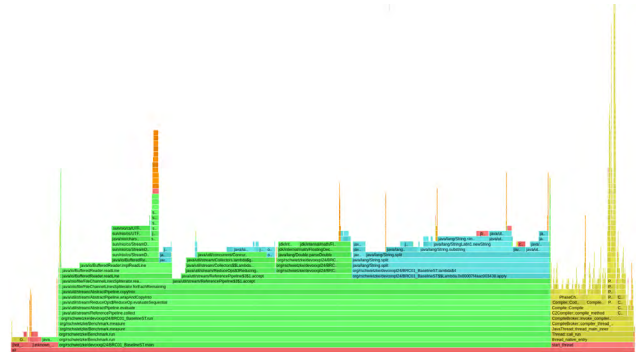
// Measurement is a record
// MeasurementAggregator is a regular data class
// ResultRow is a class that can print to the proper output
format
var result = Files.lines(Paths.get(fileName))
    .map(l -> l.split(",;"))
    .map(l -> new Measurement(l))
    .collect(groupingBy(m -> m.station(), collector));

return new TreeMap<>(result).toString();
```

The above code runs for a moment, but we don't know yet where it might spend most of its time. We don't want to guess what to optimize; we want to see. For that, we look into our toolbox and find the AsyncProfiler – perhaps the most versatile and powerful open-source profiler for Java.

With this command line, you can create this flamegraph and browse it yourself.

```
java \
-agentpath:libasyncProfiler.
so=start,event=cpu,flamegraph,file=flamegraph.html \
-cp target/classes/ \
org.rschwietzke.devvoxp124.BRC01_BaselineST measurements.
txt 2 2
```



A flamegraph of the baseline solution

That is the last flamegraph we will see in this article. You can browse a lot more when you page through the Jfokus 2025 slides. For the rest of the article, we will go with a small table that shows us the most important parts and the percentage of time spent there.

METHOD	%
run	78.4%
readLine	17.7%
split	31.7%
parseDouble	11.7%
store/reduce	14.4%

BRC01_BaselineST

Step 1: No Streams

Streams cannot be tuned easily, so let's turn it into a classic piece of Java code. Our approach is just an unrolled stream operation, and the collect and groupBy are done manually.

```
final Map<String, Temperatures> cities = new HashMap<>();

try (var reader = Files.newBufferedReader(Paths.get(fileName)))
{
    String line;
    while ((line = reader.readLine()) != null)
    {
        // split the line
        final String[] cols = line.split(",;");

        // get us the data to store
        final String city = cols[0];
        final double temperature = Double.parseDouble(cols[1]);

        // store and sum up the data
```

```
// Map::merge(key, value, remappingFunction)
cities.merge(city, new Temperatures(temperature), (t1,
t2) -> t1.merge(t2));
}
}

// ok, we got everything, now we need to order it
return new TreeMap<String, Temperatures>(cities).toString();
```

Let's look at the flamegraph data. DO is our DigitalOcean Intel Server and HE is our AMD 8-core at Hetzner.

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
80% Goal	49.5 s	20.0%	30.4 s	20.0%

Runtimes BRC03_NoStreamST

So, we gained speed by getting rid of the stream code. That was easy.

Important: This does not mean you should abandon streams. Streams are a powerful tool and should be used when appropriate. They might not be perfect for performance-critical code, but they are great for everyday use.

METHOD	01	03
run	78.4%	80.4%
readLine	17.7%	17.9%
split	31.7%	26.8%
parseDouble	11.7%	12.2%
store/reduce	14.4%	21.0%

Profiling Data BRC03_NoStreamST

Things moved a little, but the overall distribution is nearly the same.

Step 2: Split

Split consumes the majority of our time, so let's get rid of that first.

Split is a very powerful method that normally uses regular expressions to split a string. When you look at the JDK code of split, you can see that it tries to be smart. It features a fastpath when the split char is a single char and not a regex. The JIT (Just-In-Time) compiler might or might not be able to "constant fold" it (calculating the value once and reusing it). Even when this is folded, it still leaves us with a call to a larger method that creates objects and might not be inlined (where the compiler copies the method's code into the caller to avoid the overhead of a method jump)

```
// our cities with temperatures
final Map<String, Temperatures> cities = new HashMap<>();

try (var reader = Files.newBufferedReader(Paths.get(fileName)))
{
    String line;
    while ((line = reader.readLine()) != null)
    {
        // split the line
        final int pos = line.indexOf(';'); // <--

        // get us the city
        final String city = line.substring(0, pos); // <--
        final String temperatureAsString = line.substring(pos
+ 1); // <--

        // parse our temperature
        final double temperature = Double.
parseDouble(temperatureAsString);

        // merge the data into the captured data
        cities.merge(city, new Temperatures(temperature), (t1,
t2) -> t1.merge(t2));
    }
}

// ok, we got everything, now we need to order it and print it
return new TreeMap<String, Temperatures>(cities).toString()
```

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%

BRC05_ReplaceSplitST

That is impressive. Almost no effort, and we gained more than 20%. It is interesting to see that AMD and Intel are not seeing the same speedup. By the way, we will see that pattern again and again.

No, you should not replace all `String::split` in your codebase. Most code is not performance-critical and might be more readable with `split`, possibly even less error-prone.

If it runs faster on your machine, don't assume you will see the same on your production server or on another machine of a different type or architecture. Always measure and profile where you really need the speedup.

AREA	01	03	05
Total	78.4%	80.4%	81.3%
readLine	17.7%	17.9%	31.6%
split	31.7%	26.8%	-
indexOf	-	-	2.2%
subString	-	9.2%	7.6%
parseDouble	11.7%	12.2%	14.8%
merge	14.4%	21.0%	23.9%

Profiling Data BRC05_ReplaceSplitST

Our split disappeared into `indexOf` and `subString`.

Step 3: Double Parsing

When you check the `Double::parseDouble` code in the JDK, you find code that is more than 200 lines long and covers almost all possible edge cases. But we don't have edge cases here. We know that our input is always a valid double with one decimal place, so let's use that knowledge to our advantage.

```
public static double parseDouble(final String s) {
    return parseDouble(s, 0, s.length() - 1);
}

private static final double[] multipliers = {
    1, 1, 0.1, 0.01, 0.001, 0.000_1, 0.000_01, 0.000_001,
    0.000_000_1, 0.000_000_01,
    0.000_000_001, 0.000_000_000_1, 0.000_000_000_01,
    0.000_000_000_001, 0.000_000_000_000_1,
    0.000_000_000_000_001, 0.000_000_000_000_001,
    0.000_000_000_000_000_1, 0.000_000_000_000_000_01};

public static double parseDouble(final String s, final int
offset, final int end) {
    final int negative = s.charAt(offset) == '-' ? offset + 1
: offset;

    long value = 0;
    int decimalPos = end;

    for (int i = negative; i <= end; i++) {
        final int d = s.charAt(i);
        if (d == ',' || d == '.') {
            decimalPos = i;
            continue;
        }
        final int v = d - DIGITOFFSET;
        value = ((value << 3) + (value << 1));
        value += v;
    }
}
```

```
// adjust the decimal places
value = negative != offset ? -value : value;
return value * multipliers[end - decimalPos + 1];
}
```

I had some code available that I used in another project, so I used that here too. The code is already prepared to cover the next steps as well.

This code is not 100% correct due to the inherent imprecision of floating-point calculations on CPUs (see IEEE 754), but it works well enough in our case due to the limitation to one decimal place.

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%

BRC06_NewDoubleParsingST

We gained a lot on the Intel CPU. For the AMD CPU, we got it 10 seconds faster.

AREA	01	03	05	06
Total	78.4%	80.4%	81.3%	76.2%
readLine	17.7%	17.9%	31.6%	32.8%
split	31.7%	26.8%	-	-
indexOf	-	-	2.2%	2.2%
subString	-	9.2%	7.6%	9.9%
parseDouble	11.7%	12.2%	14.8%	4.4%
merge	14.4%	21.0%	23.9%	25.8%

Profiling Data BRC06_NewDoubleParsingST

Step 4: One Less String

If you look at our current code, you can easily see that we create a new String only to pass it to the `Double::parseDouble` method. Since it's only being read, we don't actually need that instance at all.

So, let's continue by parsing the double inline, based on the position of the semicolon in the original data set without splitting it for real.


```

while ((line = reader.readLine()) != null)
{
    // split the line
    final int pos = line.indexOf(',');

    // get us the city
    final String city = line.substring(0, pos);
    // OLD: final String temperatureAsString = line.
    substring(pos + 1);

    // parse our temperature inline without an instance of a
    string for temperature
    // OLD: final double temperature = Double.
    parseDouble(temperatureAsString);
    final double temperature = ParseDouble.parseDouble(line,
    pos + 1, line.length() - 1);

    // merge the data into the captured data
    cities.merge(city, new Temperatures(temperature), (t1, t2)
    -> t1.merge(t2));
}

```

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_ NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%

BRC07_NoCopyForDoubleST

That was simple and very effective.

AREA	01	03	05	06	07
Total	78.4%	80.4%	81.3%	76.2%	76.4%
readLine	17.7%	17.9%	31.6%	32.8%	29.6%
split	31.7%	26.8%	-	-	-
indexOf	-	-	2.2%	2.2%	2.6%
subString	-	9.2%	7.6%	9.9%	5.8%
parseDouble	11.7%	12.2%	14.8%	4.4%	6.0%
merge	14.4%	21.0%	23.9%	25.8%	28.7%

Profiling Data BRC07_NoCopyForDoubleST

Step 5: Double Is Double the Work

In case you have worked with old computers before, something like 486 or even the first Pentiums, you know that double is extra work and slower than int. We also know that our double here is something like XX.X, so we can easily deal with all numbers as int and only fix that up at the end when we print out the data.

By the way, double is a lot of overhead at the end of the day. That is why AI operates on far smaller floating-point units, such as FP16, BF16 and FP8, even FP4 is used.

```

/**
 * Parses a double but ends up with an int, only because we
 * know
 * the format of the results -99.9 to 99.9
 */
public static int parseInteger(final String s, final int offset,
final int end)
{
    final int negative = s.charAt(offset) == '-' ? offset + 1
    : offset;

    int value = 0;

    for (int i = negative; i <= end; i++)
    {
        final int d = s.charAt(i);
        if (d == ',')
        {
            continue;
        }
        final int v = d - DIGITOFFSET;
        // multiply by 10 = multiply by 8 + multiply by 2
        value = ((value << 3) + (value << 1));
        value += v;
    }

    value = negative != offset ? -value : value;
    return value;
}

```

Our new double parsing is actually only int parsing now and we silently drop the decimal information. Essentially, we postpone working with doubles until we print the result data. That is not yet the maximum tuning for that code, by the way.

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_ NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%

BRC08_GoIntST

We did not gain that much—just a little. You have to keep one thing in mind: a modern CPU still has more execution units for integer operations than for double-precision operations. Modern CPUs run instructions in parallel, so we're enabling them to utilize more execution units where possible. Do we really profit from that? It's difficult to tell.

AREA	01	03	05	06	07	08
Total	78.4%	80.4%	81.3%	76.2%	76.4%	79.3%
readLine	17.7%	17.9%	31.6%	32.8%	29.6%	34.4%
split	31.7%	26.8%	-	-	-	-
indexOf	-	-	2.2%	2.2%	2.6%	2.5%
substring	-	9.2%	7.6%	9.9%	5.8%	6.0%
parseDouble	11.7%	12.2%	14.8%	4.4%	6.0%	8.2%
merge	14.4%	21.0%	23.9%	25.8%	28.7%	24.2%

Profiling Data BRC08_GoIntST

Step 6: Look, No Lambda

At the moment, we utilize a lambda expression to merge the data. Because we plan some more optimizations later, we might have to change that.

```
while ((line = reader.readLine()) != null)
{
    // split the line
    final int pos = line.indexOf(',');

    // get us the city
    final String city = line.substring(0, pos);

    // parse our temperature inline without an instance of a
    // string for temperature
    final int temperature = ParseDouble.parseInteger(line, pos
        + 1, line.length() - 1);

    // get city and update
```

```
// OLD: cities.merge(city, new Temperatures(temperature),
// (t1, t2) -> t1.merge(t2));
final var v = cities.get(city);
final var t = new Temperatures(temperature);
cities.put(city, v != null ? v.merge(t) : t);
}
```

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_ NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%

BRC09_NoMergeST

Wow, that is unexpected. We are slower now. But why? This is not obvious but easily explained.

When we get our data, we need the hashCode first and we retrieve the data from the calculated position. If the data is not there, we return null. For the later `put`, we do the same again. Our lambda expression was able to combine that; hence it was only one method call, and it only needed to calculate the hashCode once and access the backing datastructure once. But we will get there again, promised.

Step 7: The Anti Pattern – Mutability

Before we try to restore the original speed again, we should look at our code and see if there is something more obvious we can tackle.

Our code, even though I have not shown that here, still uses immutable classes for storing the temperature data. That is a modern practice and perfect for concurrent code because it makes things less error-prone, but it is a silent performance killer. Nothing is free: not memory, not CPU cycles, and not memory bandwidth. Every time you create a new instance to update a value, the JVM must allocate memory and later clean it up via Garbage Collection. So, let's mutate the `Temperature` instead. By the way, the GC cleanup is not our enemy; the allocation is.

```
while ((line = reader.readLine()) != null)
{
    ...
    // get city
    final Temperatures v = cities.get(city);
    if (v != null)
    {
        // know it, put both together
```

```

        v.add(temperature);
    }
    else
    {
        // we have not seen that city yet, create a container
        and store it
        cities.put(city, new Temperatures(temperature));
    }
}

```

We only create a new `Temperatures` instance if we have not seen the city before. If we have seen the city before, we just add the temperature to the existing instance.

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_ NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%

BRC10_MutateST

That helped a lot. Let's take a look at the current state of our code. I took the liberty of removing some of the data we have seen before to keep things compact.

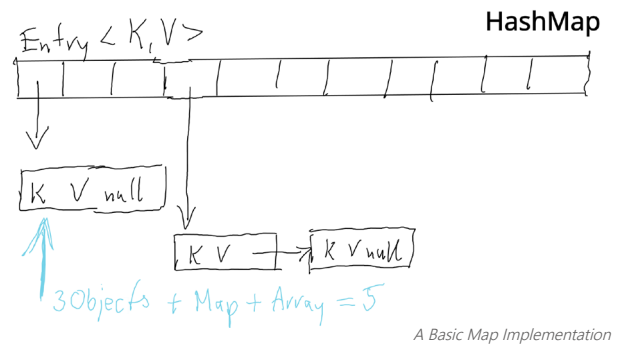
AREA	01	07	08	09	10
Total	78.4%	76.4%	79.3%	80.3%	79.3%
readLine	17.7%	29.6%	34.4%	39.1%	34.4%
split	31.7%	-	-	-	-
indexOf	-	2.6%	2.5%	2.8%	2.5%
substring	-	9.2%	6.0%	7.9%	6.2%
parseDouble	11.7%	6.0%	6.3%	6.6%	0.0%
get	-	-	-	10.9%	15.7%
merge	14.4%	28.7%	2.3%	5.2%	0.0%
put	-	-	-	5.2%	0.0%

Profiling Data BRC10_MutateST

Mutating data removes the `merge` and makes the `put` very rare, so we don't even see that in our profiling data anymore. The `get` is not slower, it just sees more share of the overall runtime. Also our `add` is so fast and likely inlined that we cannot see it in the flamegraph.

Step 8: A Simpler Map

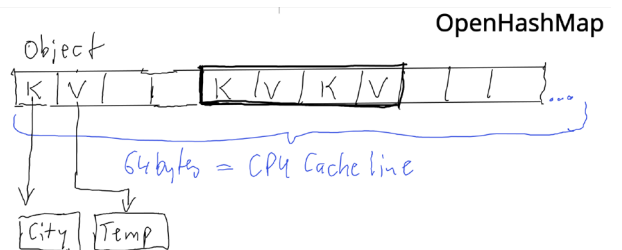
When we removed the lambda expression, we got slower. So, let's address that now. We use the default `HashMap` from the JDK, and that is a good and very versatile implementation. But we don't need that versatility here. We also know, when looking at our profiling data, that there is something like `getNode` visible. If you check the `HashMap`, you will see that it stores key and value in a wrapper object, the node. We can do better.



A Basic Map Implementation

Our map has backing arrays with wrapper objects that hold key and value. In addition, in case of collisions, we build a linked list with the wrapper objects.

A simplified `Map` implementation might look like that.



A Simple Open Hashing Map

Our open map uses a backing array as well but stores key and value next to each other. Yes, this is less perfect in terms of code beauty. When we have collisions, we store the other data simply in the next position in the array. I used `ObjObjMap` as the starting point.

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%

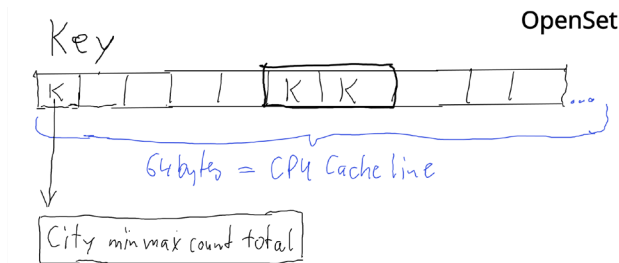
CODE	TIME DO	%	TIME HE	%
BRC07_NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%

BRC12_NewMapST

So, some gain for AMD CPUs, but not much for Intel.

Step 9: Turn the Map Into a Set

Our current map has a drawback. It needs two slots and it needs a wrapper for the values. Why not combine that into a single entry? And so we get a set.



A Simple Open Hashing Set

We can fit twice as many object references into a CPU cache line (a small, ultra-fast memory area inside the processor). Because we have already our wrapper in our hands, we don't need to look up a second object either, which reduces memory access and speeds things up.

As part of this change, we also will remove the need for a separate put method and return to the concept we abandoned when removing the lambda expression. Call the method once and let it decide if this is a new entry or just an update. In addition, we also move the `Temperatures` handling into the `Set` class. Yes, I know: reuse, architecture, clean code, and so on. Sometimes you have to trade a bit of elegance for performance. It is no longer a generic `Set`, but we're giving everything for performance, aren't we?

```
private Temperatures[] m_data;

public void getPutOrUpdate( final String city, int value ) {
    final int hash = city.hashCode();
    int ptr = hash & m_mask;
    Temperatures k = m_data[ ptr ];

    if ( k == FREE_KEY ) {
        put(new Temperatures(city, value));
        return;
    }

    if ( k.equals( city ) ) {
        k.add(value);
    }
}
```

```
return;
}

while ( true ) {
    ptr = (ptr + 1) & m_mask; //that's next index
    k = m_data[ ptr ];
    if ( k == FREE_KEY ) {
        put(new Temperatures(city, value));
        return;
    }
    if ( k.equals( city ) ) {
        k.add(value);
        return;
    }
}
}
```

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%
BRC13_HardcodedSetST	105.3 s	42.6%	76.5 s	50.3%

BRC13_HardcodedSetST

Ok, we gained about 2%. Not much, but we are making progress.

AREA	01	07	08	09	10	12	13
Total	78.4%	76.4%	79.3%	80.3%	79.3%	74.1%	72.1%
readLine	17.7%	29.6%	34.4%	39.1%	34.4%	37.9%	38.2%
split	31.7%	-	-	-	-	-	-
indexOf	-	2.6%	2.5%	2.8%	2.5%	2.0%	3.2%
subString	-	9.2%	6.0%	7.9%	6.2%	7.7%	4.5%
parseDouble	11.7%	6.0%	8.2%	6.3%	6.6%	8.3%	6.7%
get/ put(OrUpdate)	-	-	-	10.9%	15.7%	15.4%	16.1%
merge	14.4%	28.7%	24.2%	2.3%	0.0%	0.0%	-
put	-	-	-	5.2%	0.0%	0.0%	-

Profiling Data BRC13_HardcodedSetST

Step 10: Heavy Lifting – I/O

Okay, we've addressed all the "cheap" optimizations. Now what? We have to deal with the I/O side of life. How can we make `readLine` faster?

Sit back, relax, and let's dive in. Let's have another look at our data and its constraints.

Our data is well-formatted and controlled by definition. Everything is valid UTF-8, and UTF-8 includes ASCII characters. These are the first 127 bytes (not chars).

Since all our numbers, semicolons, and newlines are ASCII, why bother converting everything to char when reading from disk (which stores just bytes)? Instead, let's delay the conversion to char until we actually need a char. This is similar to our double-to-int approach: delay repeated actions that aren't necessary for immediate processing.

```
try (var raf = new RandomAccessFile(fileName, "r");
    var channel = raf.getChannel();)
{
    final Line line = new Line(channel);

    while (true)
    {
        // get us a range of bytes marked in our buffer
        line.readFromChannel();

        if (line.hasNewLine())
        {
            // parse our temperature inline without an
            // instance of a string for temperature
            final int temperature = ParseDouble.
                parseInt(line.data,
                    line.semicolonPos + 1, line.newlinePos - 1);

            // find and update
            cities.getPutOrUpdate(line, temperature);
        }
        if (line.EOF)
        {
            break;
        }
    }
}

private ByteBuffer buffer = ByteBuffer.allocate(MIN_
    BUFFERSIZE);
private byte[] data = buffer.array();

private void readFromChannel()
{
    hasNewLine = false;

    try
    {
```

```
// are we nearing the end of the buffer?
if (end - pos < REMAINING_MIN_BUFFERSIZE)
{
    // we move the buffer indirectly, because
    // the ByteBuffer just wraps our array,
    // not for the faint of heart
    System.arraycopy(data, pos, data, 0, data.length
        - pos);
    end = end - pos;
    pos = 0;
    buffer.position(end);

    // fill the buffer up
    final int readBytes = channel.read(buffer);
    if (readBytes == -1)
    {
        EOF = true;
    }

    end = buffer.position();
}
}
catch (IOException e)
{
    ...
}
lineStartPos = pos;

// look for semicolon and new line
int i = pos;
for (; i < end; i++)
{
    final byte b = data[i];
    if (b == ';' || '\n')
    {
        semicolonPos = i;
        break;
    }
}

i++;
for (; i < end; i++)
{
    final byte b = data[i];
    if (b == '\n')
    {
        newlinePos = i;
        pos = i + 1;
        hasNewLine = true;
        return;
    }
}
}
```

We could use the `ByteBuffer` directly and read bytes from it, but repeatedly calling methods that check the range of the backing array introduces significant overhead. Instead, we will access the backing array directly to gain speed.

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%
BRC13_HardcodedSetST	105.3 s	42.6%	76.5 s	50.3%
BRC14b_ReadBytesFixed	57.3 s	23.2%	46.1 s	30.4%

BRC14b_ReadBytesFixed

What a leap! Let's check our profiling data. Because the code is so different now, we will limit the profiling to our last two measurements.

AREA	13	14
Total	72.1%	80.2%
readLine	38.2%	-
readFromChannel	-	20.5%
> self	-	18.3%
> read	-	2.2%
indexOf	3.2%	-
substring	4.5%	-
parseDouble	6.7%	8.3%
putOrUpdate	16.1%	48.6%
> self	2.7%	6.5%
> add	0.8%	1.1%
> hashCode	5.1%	26.6%
> equals	7.5%	14.4%

Profiling Data BRC14b_ReadBytesFixed

Our `readLine` contains a method call to `read` and that is 2.2% of the runtime. The rest is used for our data processing. Yes, it could have been extracted more neatly into its own method. Next time!

The Unexpected Bonus

When we run our baseline code with `EpsilonGC`, we get about 3.2 GB of heap usage for 10 million lines. That sums up to a staggering 320 GB for 1 billion rows. That is memory churn at the end of the day. Request memory, clean it, use it, investigate if still in use, add it to the free list or move the content around when still in use. By the way, we are not really allocating and deallocating against the OS here, because the JVM does its own memory segment management (ask for a chunk of memory from the OS and own that as long as possible).

`EpsilonGC` (see JEP 318) is a GC that does not do any GC at all. It just reserves or assigns memory, but never frees it. Hence, a program might crash after a while if it needs more memory than what the heap can hold. Perfect for tuning and investigations because all background GC processes are offline.

Warning: Because `EpsilonGC` does not free memory anymore, it also does not compact it. Hence, live objects don't align nicely next to each other after a while, but rather are interrupted by dead objects. This hurts the cache efficiency.

```
# Run with max heap 4 GB for 10 M lines
java -Xlog:gc*=debug \
  -Xms4g -Xmx4g \
  -XX:+UnlockExperimentalVMOptions \
  -XX:+UseEpsilonGC \
  -cp target/classes/ \
  org.rschwietzke.devvoxp124.BRC01_BaselineST \
  data-10m.txt 0 1

[3.726s][info][gc] Heap: 4096M reserved, 3204M (78.22%)
committed, 3199M (78.11%) used
```

When we repeat that for our version 13, before we changed to byte reading, we see 1.04 GB memory used for 10 million lines. That is a total of 104 GB for 1 billion rows.

```
# Run with max heap 4 GB for 10 M lines
java ... \
  org.rschwietzke.devvoxp124.BRC13_HardcodedSetST \
  data-10m.txt 0 1

[2.035s][info][gc] Heap: 4096M reserved, 1129M (27.57%)
committed, 1037M (25.33%) used
```

If we repeat that with our version 14 and byte reading, we will consume 3.7 MB for 1 billion rows. You read that right: **less than 4 MB for 1 billion rows**.

```
# Run with max heap 4 MB(!) for 1000 M lines
java ... \
  -Xms4m -Xmx4m \
  org.rschwietzke.devboxpl24.BRC14_ReadBytesST \
  data-1000m.txt 0 1

[55.401s][info][gc] Heap: 4096K reserved, 4096K (100.0%)
committed, 3634K (88.73%) used
```

This is a rather unexpected bonus: we are now effectively GC-less. We are using almost no memory, and the data will likely fit into the CPU cache. One important note: we still transfer 13 GB of data from the file into our program, but because this isn't handled by the heap, it doesn't involve constant allocation and garbage collection. While the CPU cache might see some "pollution" (where reading new data from disk pushes useful data out of the cache), the overall efficiency is massive.

Step 11: Improve Number Parsing

We've made a huge leap, but we're still not quite at our 80% reduction goal. While we're very close on the Intel system, the AMD results are lagging slightly behind.

Let's try some "cheaper" tuning first. Let's revisit our number parsing.

```
/**
 * Parses a double but ends up with an int, only because we
 * know
 * the format of the results -99.9 to 99.9
 */
public static int parseIntegerFixed(final byte[] b, final int
offset, final int end) {
    final int length = end - offset; // one is missing, we care
    for that later

    // we know the first three pieces already 9.9
    int p0 = b[end];
    int p1 = b[end - 2] * 10;
    int value = p0 + p1 - (DIGITOFFSET + DIGITOFFSET * 10);

    // we are 9.9
    if (length == 2) {
        return value;
    }

    // ok, we are either -9.9 or 99.9 or -99.9
    if (b[offset] != (byte) '-') {
        // we are 99.9
        value += b[end - 3] * 100 - DIGITOFFSET * 100;
        return value;
    }

    // we are either -99.9 or -9.9
    if (length == 3) {
        // -9.9
        return -value;
    }
}
```

```
// -99.9
value += b[end - 3] * 100 - DIGITOFFSET * 100;
return -value;
}
```

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_ NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%
BRC13_HardcodedSetST	105.3 s	42.6%	76.5 s	50.3%
BRC14b_ReadBytesFixed	57.3 s	23.2%	46.1 s	30.4%
BRC15_ ParseDoubleFixedST	55.0 s	22.2%	42.7 s	28.1%

BRC15_ParseDoubleFixedST

There was still some juice to be squeezed out of this performance lemon. Eliminating the loop makes things easier for the CPU. While there are still several if-conditions, they keep the code readable. The CPU does not like to jump, so if-statements and loops always have a performance impact. The CPU tries to guess the likely branch and executes that code already in its pipeline (if possible). When the if-conditions turn out to be different, it throws away the temporary result and continues on the other branch. The fewer misses we have, the faster we will be because modern CPUs do not execute one command at a time, but many. If they cannot do that, they become slow.

Step 12: Min/Max Unrolled

Let's look at our code more closely and start with this part that updates the total values for a city.

```
public void add(final int value)
{
    this.min = Math.min(this.min, value);
    this.max = Math.max(this.max, value);
    this.total += value;
    this.count++;
}
```

This code always compares the new value with the minimum and writes to memory even if the value hasn't changed. It does the same for the maximum, even if the minimum was already updated. While `Math.min` and `Math.max` are likely efficient intrinsics,

unnecessary writes are best avoided. In most cases, neither the minimum nor the maximum values need to be updated.

```
public void add(final int value)
{
    if (value < this.min)
    {
        this.min = value;
    }
    else if (value > this.max)
    {
        this.max = value;
    }
    this.total += value;
    this.count++;
}
```

The following list shows one more improvement I haven't explained in detail yet (BRC20_UseArrayNoBufferST): we refer to the direct reference of our backing array instead of asking the `ByteBuffer` for that array every time. Since it never changes, why ask for it repeatedly?

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	323.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_ NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_ NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%
BRC13_HardcodedSetST	105.3 s	42.6%	76.5 s	50.3%
BRC14b_ReadBytesFixed	57.3 s	23.2%	46.1 s	30.4%
BRC15_ ParseDoubleFixedST	55.0 s	22.2%	42.7 s	28.1%
BRC20_ UseArrayNoBufferST	52.5 s	21.2%	42.9 s	28.2%
BRC21_ManualMinMaxST	51.5 s	20.9%	42.7 s	28.1%

BRC21_ManualMinMaxST

It is interesting to notice that the buffer trick speeds up Intel but not AMD, while min/max has a similar effect on both. Regardless, we are still not quite at our goal.

Step 13: One Less Loop

Let's carefully check our code and also look at the latest list of hot methods. We can see that we spend a lot of time on the `hashCode` calculation. How to make that cheaper?

That is our current code.

```
// OLD: look for semicolon and new line
int i = pos;
for (; i < end; i++)
{
    final byte b = data[i];
    if (b == ',')
    {
        semicolonPos = i;
        break;
    }
}

// Later
public int ByteBuffer::hashCode() {
    int h = 1;
    int p = position();
    for (int i = limit() - 1; i >= p; i--)
        h = 31 * h + (int)get(i);

    return h;
}
```

The first loop looks for the semicolon, and then later we have essentially the same loop to calculate the `hashCode`. Why not do both in one go?

```
// NEW: look for semicolon and calculate the hashCode
// Ensure that we are not including the semicolon of course
int h = 1;
int i = pos;
for (; i < end; i++)
{
    final byte b = data[i];
    if (b == ',')
    {
        semicolonPos = i;
        break;
    }
    h = 31 * h + b;
}
this.hashCode = h;
```

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%

CODE	TIME DO	%	TIME HE	%
BRC06_NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_NoCopyRotDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%
BRC13_HardcodedSetST	105.3 s	42.6%	76.5 s	50.3%
BRC14b_ReadBytesFixed	57.3 s	23.2%	46.1 s	30.4%
BRC15_ParseDoubleFixedST	55.0 s	22.2%	42.7 s	28.1%
BRC20_UseArrayNoBufferST	52.5 s	21.2%	42.9 s	28.2%
BRC21_ManualMinMaxST	51.5 s	20.9%	42.7 s	28.1%
BRC22_EarlyHashCodeST	39.1 s	15.8%	36.4 s	24.0%

BRC22_EarlyHashCodeST

And we have liftoff! The Intel machine broke the 20% barrier (80% reduction), while the AMD machine is almost there.

More Smaller Tuning for AMDs

To get the AMD CPU over the line, we will perform more smaller tuning and fiddle around with ideas. We keep some, but will abandon others. At the end, we will finally see our code runs five times faster, as promised. No usage of Unsafe, no bit magic, no GraalVM. To limit the article length, I won't discuss the other changes in detail.

If you want to see what changed, hop over to the repo and check it out. It also contains examples of what did not work. You can also see that I discovered a defect when comparing with another dataset: BRC58_UnnoticedCharSkipping. We already have a checksum in place; still, some errors only come up when the input looks different—strange, isn't it?

CODE	TIME DO	%	TIME HE	%
BRC01_BaselineST	247.6 s	100.0%	152.0 s	100.0%
BRC03_NoStreamST	232.1 s	93.7%	138.9 s	91.4%
BRC05_ReplaceSplitST	175.9 s	71.0%	109.7 s	72.2%
BRC06_NewDoubleParsingST	141.2 s	57.0%	99.0 s	65.2%
BRC07_NoCopyForDoubleST	127.7 s	51.6%	88.0 s	57.9%
BRC08_GoIntST	124.0 s	50.1%	87.3 s	57.4%
BRC09_NoMergeST	131.1 s	53.0%	95.4 s	62.7%
BRC10_MutateST	109.8 s	44.4%	84.9 s	53.9%
BRC12_NewMapST	109.6 s	44.3%	80.2 s	52.9%
BRC13_HardcodedSetST	105.3 s	42.6%	76.5 s	50.3%

BRC14b_ReadBytesFixed	57.3 s	23.2%	46.1 s	30.4%
BRC15_ParseDoubleFixedST	55.0 s	22.2%	42.7 s	28.1%
BRC20_UseArrayNoBufferST	52.5 s	21.2%	42.9 s	28.2%
BRC21_ManualMinMaxST	51.5 s	20.9%	42.7 s	28.1%
BRC22_EarlyHashCodeST	39.1 s	15.8%	36.4 s	24.0%
BRC29a_ParseDoubleTuningST	40.7 s	16.4%	31.7 s	20.9%
BRC29c_ArrayCopyInMethod	35.8 s	14.5%	30.7 s	20.3%
BRC29d_EqualsNotBoolean	37.0 s	15.0%	31.4 s	20.7%
BRC29e_EarlyIntResolution	36.8 s	14.9%	31.6 s	20.8%
BRC29f_LessDataForTempResolution	36.4 s	14.7%	31.4 s	20.7%
BRC29g_FixedIntParsing	38.9 s	15.7%	31.9 s	21.0%
BRC40a_NoChannel	38.6 s	15.6%	31.0 s	20.4%
BRC40b_ReturnInstead	35.3 s	14.3%	31.5 s	20.7%
BRC40c_UnrollTempParsing	32.4 s	13.1%	28.8 s	19.0%
BRC40d_LongLoop	33.2 s	13.4%	28.3 s	18.7%
BRC40e_NoReloadSub	31.7 s	12.8%	28.1 s	18.5%
BRC40f_DoWhile	31.8 s	12.9%	28.1 s	18.5%
BRC40g_Put	31.9 s	12.9%	28.1 s	18.5%
BRC40h_ManualMismatch	33.6 s	13.6%	27.3 s	18.0%
BRC40i_SmallerSemicolonLoop	32.4 s	13.1%	27.0 s	17.8%
BRC40j_LessStateInSetEquals	32.1 s	13.0%	27.2 s	17.9%
BRC40_NoChannel	35.3 s	14.3%	31.6 s	20.8%
BRC41a_FixedFastHashSet	31.8 s	12.9%	27.3 s	18.0%
BRC41b_ReorderedLineFields	31.8 s	12.9%	27.4 s	18.0%
BRC41c_LargerBuffer	31.8 s	12.9%	27.2 s	18.0%
BRC42a_WhileTrue	31.6 s	12.8%	27.2 s	17.9%
BRC42b_NoReturnBranch	32.4 s	13.1%	28.6 s	18.8%
BRC43_NoSubClass	33.2 s	13.4%	27.6 s	18.2%
BRC45_DoubleTheSetSize	32.5 s	13.2%	27.4 s	18.0%
BRC46_TunedHashSet	32.5 s	13.1%	28.2 s	18.6%
BRC47_LeanerPut	32.5 s	13.1%	28.0 s	18.5%
BRC48_FixedFactor	34.0 s	13.7%	28.2 s	18.6%
BRC49_OffsetSubtraction	32.5 s	13.1%	27.8 s	18.3%
BRC50_Short	32.2 s	13.0%	27.7 s	18.3%
BRC51_TempParsingLessBranches	32.9 s	13.3%	28.3 s	18.6%
BRC52_TempParsingBitSubtraction	31.8 s	12.9%	27.7 s	18.3%
BRC53_SetEqualsNoLocalAssignment	31.6 s	12.8%	27.2 s	17.9%
BRC54_LoopVariableBackInLoop	31.8 s	12.9%	27.3 s	18.0%
BRC55_SimplerPutCall	31.7 s	12.8%	27.2 s	17.9%
BRC56_MainLoopAsWhile	31.6 s	12.8%	26.5 s	17.5%
BRC57_SimplerHashing_VOID	32.0 s	12.9%	25.9 s	17.1%
BRC58_UnnoticedCharSkipping	33.5 s	13.6%	27.4 s	18.0%
BRC60_FeedCPUUnrollSemicolonLoop	31.3 s	12.7%	26.1 s	17.2%

BRC60_FeedCPUUnrollSemicolonLoop

There we go—tuned without turning the code into a complete disaster. It's a bit "messy" perhaps, but still maintainable. Keep in mind that we're relying on clean, well-formatted data. If we had to add checks for malformed data or varying lengths, the complexity (and performance) would look quite different.

Did you notice that we lost performance again after applying our bug fix (BRC58)? But why was the outcome the same? Well, if you add up 2,000 or more temperatures per city and average them, it doesn't matter if all 2,000 were correctly read. We may notice that quicker if the difference becomes too large or one of the extremes is missing.

Learning: Always vary your test data!

Behind the Scenes

For the fans of hard-core performance and low-level facts, let's dive into the CPU and memory details. This is where "Mechanical Sympathy" – designing software that works in harmony with the underlying hardware – truly comes into play. These statistics are provided by the `perf` tool, courtesy of the Linux kernel.

`perf` does not work on all cloud machines. It depends on the virtualization layer and the set permissions. Your mileage may vary!

We are comparing the initial baseline, our 20% solution (BRC45), and the overall competition winner to understand the hardware-level impact of these optimizations.

Bottom line: the initial code needs about 2,258 CPU instructions for each CSV line. Our tuned version drops this to 303, and the winner achieves the task with just 164 instructions.

Metric	BRC01 (Baseline)	BRC45 (20% Solution)	1BRC Winner
Runtime	150,932 ms (100%)	27,061 ms (17.9%)	11,281 ms (7.4%)
CPU's utilized	1.066	1.018	0.999
Context-switches	37,195	1,029	1,748
CPU-migrations	6,232	29	1
Page-faults	236,617	7,885	223,183
Cycles	539,415 M	98,276 M	39,555 M
Stalles-cycles-frontend	41,233 M	10,431 M	2,945 M
Instructions	2,258,010 M	303,659 M	164,487 M
Instructions Per Cycle (IPC)	4.19	3.09	4.16
Branches	522,584 M	55,868 M	15,668 M
Branch-misses	2,479 M (0.47%)	1,540 M (2.76%)	149 M (0.95%)
Instructions per CSV Line	2,258	303	164

Performance Comparison: Evolution from Baseline to Winner

IPC (Instructions Per Cycle) measures how much work the CPU completes in each "clock tick." A higher number means better utilization of the CPU's internal resources. Branch-misses occur when the CPU incorrectly predicts which way an 'if' or 'loop' will go, causing a pipeline stall that wastes time.

Baseline to BRC45

```
# org.rschwietzke.devovxpl24.BRC01_BaselineST
# Hetzner, AMD, 21.0.4-tem
# Measurement Runtime: 150,932 ms 100%

160,915.21 ms task-clock           # 1.066 CPUs utilized
    37,195 context-switches        # 231.147 /sec
    6,232  cpu-migrations           # 38.728 /sec
   236,617 page-faults             # 1.470 K/sec
539,415,718,644 cycles              # 3.352 GHz
 41,233,849,888 stalled-cycles-frontend # 7.64%
                          frontend cycles idle
2,258,010,166,520 instructions     # 4.19 insn
                                per cycle
                                # 0.02 stalled
                                cycles per
                                insn
522,584,792,160 branches           # 3.248 G/sec
 2,479,441,210 branch-misses       # 0.47% of all
                                branches

150.932228741 seconds time elapsed

145.057695000 seconds user
 11.626114000 seconds sys

# 2258 instructions per CSV line
```

```
# org.rschwietzke.devovxpl24.BRC45_DoubleTheSetSize
# Hetzner, AMD, 21.0.4-tem
# Mean Measurement Runtime: 27,061 ms 17.9% (to baseline)

27,538.78 msec task-clock           # 1.018 CPUs utilized
    1,029 context-switches        # 37.365 /sec
    29    cpu-migrations           # 1.053 /sec
    7,885 page-faults             # 286.324 /sec
98,276,507,946 cycles              # 3.569 GHz
10,431,399,857 stalled-cycles-frontend # 10.61%
                          frontend cycles idle
303,659,521,159 instructions     # 3.09 insn
                                per cycle
                                # 0.03 stalled
                                cycles per
                                insn
55,868,360,010 branches           # 2.029 G/sec
 1,540,980,291 branch-misses       # 2.76% of all
                                branches
```

```
27.061235813 seconds time elapsed
```

```
26.366348000 seconds user
```

```
1.062075000 seconds sys
```

```
# 303 instructions per CSV line
```

- The number of branches dropped by a factor of five from baseline to our BRC45.
- Cache misses (stalled-cycles-frontend) dropped to 25%.
- We now require only 14% of the instructions.
- Absolute branch-misses are down, but relatively they have increased.

BRC45 to 1BRC Winner

```
# Thomas Wuerthinger, 1 core
# Hetzner, AMD, 21.0.4-tem
# Mean Measurement Runtime: 11,281 ms 7.4% (to baseline)
# 41.7% of BRC45_DoubleTheSetSize

11,269.99 msec task-clock          # 0.999 CPUs
                                   utilized
      1,748 context-switches      # 155.102 /sec
              1 cpu-migrations    # 0.089 /sec
      223,183 page-faults         # 19.803 K/sec
39,555,927,795 cycles              # 3.510 GHz
2,945,609,903 stalled-cycles-frontend # 7.45%
                                   frontend
                                   cycles idle
164,487,950,474 instructions      # 4.16 insn
                                   per cycle
                                   # 0.02 stalled
                                   cycles per
                                   insn
15,668,783,505 branches           # 1.390 G/sec
149,371,238 branch-misses         # 0.95% of all
                                   branches

11.281867347 seconds time elapsed

0.076590000 seconds user
0.130600000 seconds sys

# 164 instructions per CSV line
```

- The reduction in the number of branches is staggering and mostly comes from branchless number parsing.
- The instruction reduction stems mostly from reading 8 bytes at once instead of byte by byte.
- Branch-misses reduction makes it truly fly.

By the way, we could avoid `Unsafe` and read a `long` legally from the `ByteBuffer` using its `getLong()` method. Sadly, there's no easy way to parse that without resorting to bit-magic. Furthermore, a call to `getLong` includes several safety checks, which can render some of our gains negligible.

Different Machines

We have already seen that CPUs are very much different; we can easily extend that warning to types of machines.

Test	Machine	Time	%
BRC01_BaselineST	Thinkpad T14s	218.8 s	100.0%
BRC01_BaselineST	GCP-C2-4c-32g	278.4 s	100.0%
BRC01_BaselineST	GCP-C2D-4c-32g	172.4 s	100.0%
BRC01_BaselineST	DO Intel Standard 16c-32g	247.6 s	100.0%
BRC01_BaselineST	DO Intel Premium 16c-32g	154.2 s	100.0%
BRC01_BaselineST	Hetzner AMD 8c-32g	146.7 s	100.0%
BRC26_MoreMapSpaceST	Thinkpad T14s	35.3 s	16.1%
BRC26_MoreMapSpaceST	GCP-C2-4c-32g	43.4 s	15.6%
BRC26_MoreMapSpaceST	GCP-C2D-4c-32g	34.8 s	20.2%
BRC26_MoreMapSpaceST	DO Intel Standard 16c-32g	39.0 s	15.8%
BRC26_MoreMapSpaceST	DO Intel Premium 16c-32g	31.1 s	19.8%
BRC26_MoreMapSpaceST	Hetzner AMD 8c-32g	33.5 s	22.8%

Different Machine Types and Their Performance

It is highly recommended to benchmark on the hardware you will run your production code on; otherwise, all your work is for nothing.

Every so often you just pick another machine type for just a few cents more, and you get twice the speed without doing anything. Maybe that will do the trick already?

The Cloud is Cloudy

Measuring performance in the cloud is often a trap because you are rarely alone on the hardware. Shared resources mean that small performance gains can be easily drowned out by “noisy neighbors” or unexpected fluctuations in CPU and memory bandwidth. Furthermore, features like burst capacity can lead to inconsistent results. To make matters worse, low-level profiling tools like `perf` often fail in virtualized environments, leaving you flying blind when trying to understand hardware effects.

```
# BRC01_BaselineST - DigitalOcean Intel CPU 16c
Warmup Runtime: 219,665 ms
Warmup Runtime: 220,341 ms
Warmup Runtime: 219,312 ms
Measurement Runtime: 219,020 ms
Measurement Runtime: 223,180 ms
Measurement Runtime: 226,418 ms
```

```
# BRC27_SmallPutST - DigitalOcean Intel CPU 16c
Warmup Runtime:      38,792 ms
Warmup Runtime:      37,929 ms
Warmup Runtime:      39,136 ms
Measurement Runtime: 39,197 ms
Measurement Runtime: 39,103 ms
Measurement Runtime: 39,106 ms
```

The above listing shows a few runs on DigitalOcean Intel hardware. You can see how the runtimes fluctuate. The tuned version is slightly more stable, which might be partly due to JVM effects like the absence of GC (as we moved to a zero-allocation approach). However, for micro-optimizations, even these small fluctuations can be a significant hurdle.

Summary

The One Billion Row Challenge is a fantastic way to explore Java performance. By applying a systematic tuning approach, starting from a clean baseline and incrementally optimizing everything from stream usage to I/O and custom collection implementations, we achieved an 80% reduction in runtime using only standard Java features.

While `Unsafe` is not a standard feature in my view, and extreme bit magic is fully legal but makes the code difficult to understand for most people, those techniques fall under: only when truly needed, only by experts, and only with a ton of documentation.

These techniques aren't just for coding competitions; they demonstrate how a deeper understanding of the JVM, memory management, and hardware characteristics can lead to big efficiency gains. Don't forget: faster means fewer resources, and that translates directly into lower infrastructure costs.

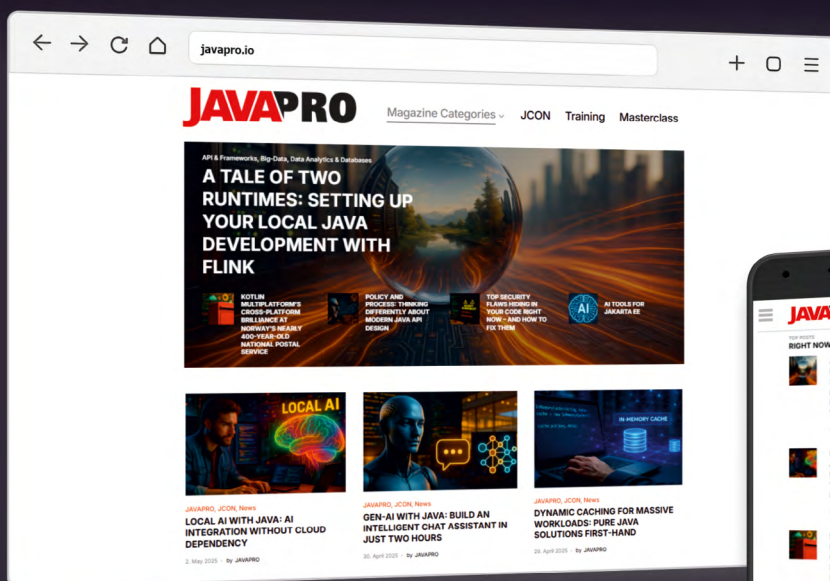
You can also watch this talk on YouTube and browse the slides online. If you like that kind of content, please let me know. Additional performance content can be found in my personal blog.

Always remember to measure, profile, and optimize only where it truly matters!

-> Back to the Table of Content

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



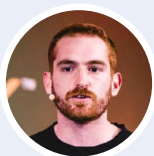
www.javapro.io



#JAVAPRO #COREJAVA

Java for AI: Building GPU Accelerated LLM Inference Libraries with TornadoVM

Author:



Michalis Papadimitriou is a Research Fellow at the University of Manchester and a Staff Software Engineer on the TornadoVM team. His core expertise includes open-source software development, hardware abstractions for high-level programming languages, compiler optimizations for GPU computing, and enabling large language model (LLM) inference on GPUs for the Java Virtual Machine (JVM). Michalis is focused on advancing GPU acceleration for machine learning workloads on the JVM through the TornadoVM framework and actively maintains the GPULLama3.java project. Before joining the University of Manchester, he worked on a range of software stacks at Huawei Technologies and contributed to the open-source machine learning compiler Apache TVM, while working for OctoAI (formerly OctoML), which was later acquired by Nvidia.

<https://www.linkedin.com/in/michalis-papadimitriou/>

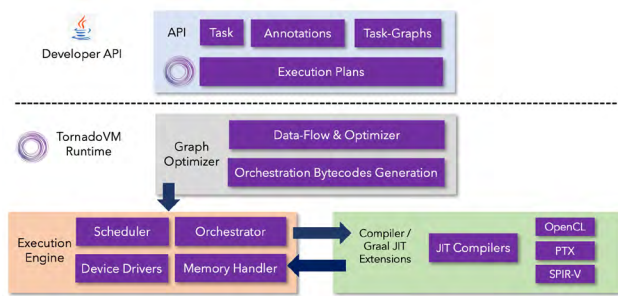
Enterprise Java teams can now run local LLM inference with GPU acceleration without leaving the JVM or writing a single line of CUDA. TornadoVM and GPULLama3.java have fundamentally changed what's possible for Java developers in the AI era, enabling pure-Java transformer inference. This isn't experimental research; it's open-source, LICENSE, production-ready technology already deployed at the European Space Agency (ESA) for mission-critical astronomical data processing.

This article aims to showcase the key TornadoVM features used to build GPULLama3.java, the first GPU-accelerated LLM inference library in pure modern Java, which is already integrated within projects such as `LangChain4j` and `Quarkus LangChain4j`.

What Is TornadoVM?

TornadoVM is an open-source **plugin for the Java Virtual Machine (JVM)** that enables standard Java programs to be automatically compiled and accelerated on **multi-core CPUs, GPUs, and FPGAs**, without requiring developers to write CUDA or OpenCL. It interoperates with OpenJDK and other JDK distributions, extends the JVM's just-in-time compilation using the Graal compiler, and generates device-specific code (e.g., OpenCL C, NVIDIA PTX, or SPIR-V) so that parallel parts of Java applications can run efficiently on different hardware

accelerators. At the same time, the TornadoVM runtime handles data movement and execution scheduling in a platform-agnostic manner.



The TornadoVM Design as “VM-inside-a-VM”

TornadoVM is designed following a nested virtual machine architecture that sits atop the JVM rather than replacing it. This design preserves full Java compatibility while enabling automatic hardware acceleration. The framework maintains **its own set of bytecodes (TornadoVM Bytecodes)** as a means to seamlessly orchestrate the JIT compilation, device exploration, data management, and live task migration between accelerators.

Regarding compilation, TornadoVM leverages JVMCI (Java Virtual Machine Common Interface), available since Java 9, to compile Java bytecode at runtime into three different formats: **OpenCL C source code**, **NVIDIA PTX assembly**, and **SPIR-V binary**. To generate code in these three different formats, TornadoVM has three backends; each built upon the Graal Intermediate Representation. Hence, TornadoVM JIT compiler is a superset of the GraalVM JIT compiler and harnesses all architecture-agnostic optimizations offered by Graal before generating device-specific code.

Compilation proceeds through distinct phases. The Sketcher phase performs high-level optimizations, such as, data-flow analysis, loop parallelization exploration which are common across all backends. After Sketcher, each backend performs the generation of code, which GPU or FPGA drivers compile into final binaries. These binaries are stored into TornadoVM’s code cache, ready for repeated invocation without recompilation.

Hardware support spans the heterogeneous computing landscape. The **OpenCL backend** targets multi-core CPUs (x86 and ARM), GPUs from NVIDIA, AMD, Intel, and ARM Mali, plus FPGAs from Intel and Xilinx. The **PTX backend** generates native NVIDIA CUDA assembly, while the **SPIR-V backend** dispatches via Level Zero for Intel Arc GPUs and integrated graphics. This breadth means code written once runs across diverse hardware where developers achieve GPU acceleration without vendor lock-in.

TornadoVM API Makes Parallel Programming Feel like Ordinary Java

TornadoVM offers an Application Programming Interface (API) that exposes two Java objects:

- **TaskGraph:** To define **what** computation should be launched for hardware acceleration, including the definition of the input/output data. At this stage, developers can also combine multiple computations that should be executed in-order on the same device.
- **TornadoExecutionPlan:** To define **how** the execution of the TaskGraph should be planned. At this stage, developers can configure if they want to run with the TornadoVM profiler, or they want to launch a specific grid on the target device.

Except for those two objects, TornadoVM offers two complementary approaches for expressing parallel computations inside Java methods:

- **Loop Parallel API:** It is designed for developers with little or no GPU programming experience. It exposes a small set of annotations (`@Parallel`, `@Reduce`) that allow developers to express data parallelism directly in standard idiomatic Java code. Those annotations act as hints to the TornadoVM JIT compiler, which analyzes the annotated code and generates the corresponding parallel implementation for the target accelerator. The `@Parallel` annotation is applied to loop indices to signify that loop iterations are data-independent and can be executed in parallel on different data. Whereas, the `@Reduce` is applied to method parameters that accumulate results from parallel execution (e.g., sums, min/max), enabling TornadoVM to generate the appropriate reduction logic.
- **Kernel API:** It is designed for developers with GPU programming experience. Instead of relying on annotations, the Kernel API exposes a `KernelContext` object that allows developers to explicitly control GPU-style execution directly from Java. This API follows familiar OpenCL and CUDA concepts, giving experienced users fine-grained control over thread indexing, local memory, and synchronization. In contrast to the higher-level Loop Parallel API, the Kernel API enables writing Java methods that closely resemble traditional GPU kernels.

Example of Defining a TaskGraph

A `TaskGraph` object is used to select specific Java methods (“tasks”) together with their explicit data transfer directives, and to combine into a single executable unit for execution on the accelerator. Data transfers are configured using the **DataTransferMode** enum, which specifies when host-device memory copies occur: `FIRST_EXECUTION` transfers read-only data only once (ideal for model weights), while `EVERY_EXECUTION` handles mutable state that changes between invocations

```
var taskGraph = new TaskGraph(„inference“)
// Copy read-only data just only once
.transferToDevice(DataTransferMode.FIRST_EXECUTION,
weights)
// Copy input stream on every execution
.transferToDevice(DataTransferMode.EVERY_EXECUTION, input)
// Define the method to offload along with it's arguments
.task(„forward“, Model::forward, weights, input, output)
```



```
// Output: every time
.transferToHost(DataTransferMode.EVERY_EXECUTION, output);
```

Example of Defining a TornadoExecutionPlan

Once a `TaskGraph` has been defined, a `TornadoExecutionPlan` object specifies how the `TaskGraph` will be executed. While the `TaskGraph` describes what should run and how data is transferred, the `TornadoExecutionPlan` determines how and where the execution takes place.

Importantly, the `TornadoExecutionPlan` operates on an immutable version (a snapshot) of the `TaskGraph`. By invoking `snapshot()` on a `TaskGraph` object, developers obtain an `ImmutableTaskGraph`, ensuring that the version of the code selected for JIT compilation cannot be modified. This guarantees stability during compilation and execution, enabling predictable optimizations and reproducible behavior.

This design also allows developers to programmatically construct and append multiple `TaskGraph` instances, generate different variations, and then execute them together. Since the execution plan compiles the immutable snapshot, these `TaskGraphs` can be compiled once and safely reused without the risk of runtime mutation.

```
ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();

try (var executionPlan = new
    TornadoExecutionPlan(immutableTaskGraph)) {
    executionPlan.withProfiler(ProfilerMode.CONSOLE).execute();
}
```

Using Off-Heap Data Types

TornadoVM uses the Foreign Function and Memory (FFM) API derived from Project Panama to provide a collection of off-heap data types (e.g., `FloatArray`, `HalfFloatArray`, `IntArray`, `Matrix2DFloat`, etc.). These data types are specifically designed to be GPU-compatible while avoiding garbage collection overhead.

In addition, the `TornadoMath` library supplies mathematical functions with automatic precision fallback, ensuring compatibility with devices that lack FP64 support.

Both the TornadoVM data types and the `TornadoMath` library are intended for use within TornadoVM tasks, as shown in the examples below.

Example of Using the Loop Parallel API

Within TornadoVM, Java methods that are registered as *tasks* in a `TaskGraph` can be annotated to express data-parallel execution. The **@Parallel** annotation allows developers to indicate that specific loops inside these task methods can be parallelized, provided there are no data dependencies between iterations.

When such a task is about to be executed on the accelerator, The TornadoVM JIT compiler analyzes the annotated loops and maps them directly to ND-Range execution in the OpenCL, CUDA, or SPIR-V backends. The number of parallel threads is automatically derived from the loop bounds.

The following example shows a Java method intended to be used as a task in a `TaskGraph`. The **@Parallel** annotation enables parallel execution of the loop iterations to perform element-wise vector-multiplication:

```
public static void vectorMultiply(FloatArray a, FloatArray b,
    FloatArray result) {
    for (@Parallel int i = 0; i < result.getSize(); i++) {
        result.set(i, a.get(i) * b.get(i));
    }
}
```

Nested **@Parallel** loops within a task are mapped to 2D and 3D parallel execution when deployed on the hardware device. This pattern is common in image processing and matrix-based operations. For example

```
private static void matrixMultiplication(final FloatArray
    A, final FloatArray B, final FloatArray C, final int size) {
    for (@Parallel int i = 0; i < size; i++) {
        for (@Parallel int j = 0; j < size; j++) {
            float sum = 0.0f;
            for (int k = 0; k < size; k++) {
                sum += A.get((i * size) + k) * B.get((k *
                    size) + j);
            }
            C.set((i * size) + j, sum);
        }
    }
}
```

Example of Using the Kernel API

Alternatively to the Loop Parallel API, developers with GPU programming experience can use the Kernel API to construct Java methods that follow familiar GPU programming principles. The `KernelContext` object exposes GPU programming constructs familiar to CUDA and OpenCL developers. Thread indices, local memory allocation, and barriers translate naturally between paradigms, as shown in the table below.

TornadoVM	OpenCL	CUDA
context.threadIdx	get_global_id(0)	threadIdx.x + blockIdx.x * blockDim.x
context.localIdx	get_local_id(0)	threadIdx.x
context.allocate- FloatLocalArray()	__local float[]	__shared__ float[]

TornadoVM	OpenCL	CUDA
context. localBarrier()	barrier()	-- syncthreads()

This API gives developers explicit control over execution configuration and memory hierarchy, making it suitable for performance-critical kernels or when fine-tuned optimization is required. Such concerns are discussed in the following paragraphs.

Local Memory and Tiled Computation Unlock GPU Memory Hierarchy Performance

The matrix multiplication implementation with the Loop Parallel API, as shown above, works correctly but there is still room for extracting more performance. GPUs have a memory hierarchy composed of a **global memory** (device VRAM) which has high latency and limited bandwidth, while **local memory** (shared memory in CUDA terminology) which offers much faster access within a workgroup (i.e., a group of GPU threads). A developer can take advantage of the above with pure idiomatic Java using the TornadoVM Kernel API. As shown earlier, the `KernelContext` object provides methods that expose the GPU memory hierarchy, enabling the same tiled algorithms that make CUDA implementations fast to be expressed in Java.

The optimization principle: Instead of each thread independently fetching elements from global memory, workgroups **cooperatively load tiles** (e.g., chunks) into fast local memory, then all threads in the group perform the computation against the cached tile. This reduces global memory accesses from $O(N)$ per thread to $O(N/TS)$ where TS is the tile size.

```
private static final int TS = 16; // Tile size: 16x16 threads
per workgroup
public static void matrixMultiplicationLocalMemory(
    KernelContext context, final FloatArray A, final FloatArray B,
    final FloatArray C, final int size) {

    // Thread identifiers within the workgroup (0 to TS-1)
    int row = context.localIdx;    // Local row ID
    int col = context.localIdy;    // Local col ID

    // Global position in the output matrix
    int globalRow = TS * context.groupIdx + row; // Row ID of C
    (0..M)
    int globalCol = TS * context.groupIdy + col; // Col ID of C
    (0..N)

    // Allocate local memory tiles - shared across all threads in
    workgroup
    float[] aSub = context.allocateFloatLocalArray(TS * TS);
    float[] bSub = context.allocateFloatLocalArray(TS * TS);

    float sum = 0;
```

```
// Loop over all tiles along the K dimension
int numTiles = size / TS;
for (int t = 0; t < numTiles; t++) {
    // Each thread loads ONE element of the tile into local
    memory
    int tiledRow = TS * t + row;
    int tiledCol = TS * t + col;
    aSub[col * TS + row] = A.get(tiledCol * size + globalRow);
    bSub[col * TS + row] = B.get(globalCol * size + tiledRow);

    // BARRIER: Wait for ALL threads to finish loading the tile
    context.localBarrier();

    // Compute partial dot product using the cached tile
    for (int k = 0; k < TS; k++) {
        sum += aSub[k * TS + row] * bSub[col * TS + k];
    }

    // BARRIER: All threads finished computing before loading
    next tile
    context.localBarrier();
}

// Store the final accumulated result
C.set((globalCol * size) + globalRow, sum);
}
```

Understanding the memory access pattern reveals why this optimization matters. In the naive version, computing one element of C requires reading an entire row of A and a column of B from global memory – $2N$ memory accesses per output element. With N^2 output elements, that's $2N^3$ global memory accesses total.

The tiled version groups threads into workgroups of $TS \times TS$ threads, and each workgroup loads two $TS \times TS$ tiles ($2 \times TS^2$ elements), then performs TS^3 multiply-accumulate operations against local memory. Across N/TS tiles, each workgroup performs $(N/TS) \times 2 \times TS^2 = 2 \times N \times TS$ global memory accesses while computing $TS^2 \times N$ multiply-adds. The ratio of computation to memory access improves by a factor of TS – typically 16x or 32x depending on tile size.

The `context.localBarrier()` calls are used to add synchronization barriers in order to ensure correctness. Without the first barrier, some threads might start computing before others have finished loading their tile elements – reading uninitialized or stale data. Without the second barrier, fast threads might overwrite tile data with the next iteration's values while slow threads would still need the current values. These barriers correspond directly to intrinsic functions, such as `__syncthreads()` in CUDA and `barrier(CLK_LOCAL_MEM_FENCE)` in OpenCL.

The following code snippet shows the definition of a `workerGrid` that enables developers to define the number of GPU threads (identified as `GlobalWork`) and the size of the workgroups (i.e., `LocalWork`) following the sizes used in the method scope. In

this example, a two-dimensional `workerGrid` is defined with size that matches the output matrix dimensions, while the local work group is configured to be 16.

Once a `WorkerGrid` is defined, a `GridScheduler` must be created to associate that grid configuration with the corresponding TornadoVM tasks. The scheduler links the `WorkerGrid` to a specific task by its fully qualified name, which consists of the `TaskGraph` identifier followed by the task identifier (e.g., `"matmul.tiled"`). The final step is to configure the `TornadoExecutionPlan` and pass the `GridScheduler` instance to the `.withGridScheduler()` method. This ensures that the specified grid configuration is used when executing the computation defined by the `"matmul.tiled"` task.

```
java
var workerGrid = new WorkerGrid2D(size, size);
workerGrid.setLocalWork(TS, TS, 1); // 16x16 threads per
workgroup

var gridScheduler = new GridScheduler("matmul.tiled",
workerGrid);

var taskGraph = new TaskGraph("matmul")
.transferToDevice(DataTransferMode.FIRST_EXECUTION, A, B)
.task("tiled", MyClass::matrixMultiplicationLocalMemory,
context, A, B, C, size).transferToHost(DataTransferMode.
EVERY_EXECUTION, C);

var plan = new TornadoExecutionPlan(taskGraph.snapshot()).
withGridScheduler(gridScheduler);`
```

What Is GPULLama3.java

GPULLama3.java is an open-source **pure Java framework for running LLM (large language model) inference on GPUs** by leveraging **TornadoVM's GPU acceleration** meaning you can load and execute models like **Llama 3 entirely in Java** without writing CUDA, OpenCL, or other native GPU code. At the same time, it delivers the performance benefits of the underlying GPU hardware. At its core, GPULLama3.java builds on the Java open-source LLM ecosystem (e.g., llama3.java) and **seamlessly offloads heavy computation, such as attention and matrix ops, to GPUs**, leveraging TornadoVM's ability to compile Java bytecode into high-performance parallel kernels. The outcome is **GPU-accelerated inference** for models in **GGUF format** (currently including Llama 3, Mistral, Qwen 2.5, Qwen 3, Phi-3, and IBM Granite) while keeping all code in the JVM. GPULLama3.java is a GPU inference engine integrated in LangChain4j and Quarkus LangChain4j.

GPULLama3.java Implements Transformer Layers as Composable TaskGraphs

GPULLama3.java demonstrates TornadoVM's maturity through a complete LLM inference implementation. The architecture decomposes transformer layers into three `TaskGraph` categories: an **activation layer** for token embeddings and

initial normalization, **FFN layers** implementing the transformer blocks, and a **logits layer** for final vocabulary projection. The `FP16LayerPlanner` class orchestrates execution by combining these `TaskGraphs` with a unified `GridScheduler`:

```
protected final void setupTornadoForwardPlan() {
    List<ImmutableTaskGraph> allTaskGraphs = new
    ArrayList<>();
    GridScheduler masterScheduler = new GridScheduler();

    // Activation: embeddings + initial normalization
    allTaskGraphs.add(activationLayer.
    getImmutableTaskGraph());
    activationLayer.updateGridScheduler(masterScheduler);

    // FFN: N transformer layers (model-dependent)
    allTaskGraphs.addAll(ffnLayers.getFfnLayerTaskGraphs());
    ffnLayers.updateGridScheduler(masterScheduler);

    // Logits: final projection to vocabulary
    allTaskGraphs.add(logitsLayer.getTaskGraph().snapshot());
    logitsLayer.updateGridScheduler(masterScheduler);

    this.immutableTaskGraphs = allTaskGraphs;
    this.gridScheduler = masterScheduler;
}
```

Each FFN layer `TaskGraph` includes QKV projections, rotary position embeddings, attention computation, and feed-forward network operations. The setup method chains these as distinct tasks with explicit data dependencies:

```
// Query, Key, Value matrix multiplications
.task("qmatmul",
    TransformerComputeKernels::matrixVectorGeneric, context,
    state.wrapXb, state.wrapQ, weights.wqLayered[layer].
    asHalfFloatArray(),
    config.dim(), config.dim(), LOCAL_WORK_GROUP_SIZE)
.task("kmatmul",
    TransformerComputeKernels::matrixVectorGeneric, context,
    state.wrapXb, state.wrapK, weights.wkLayered[layer].
    asHalfFloatArray(),
    config.dim(), config.kvDim(), LOCAL_WORK_GROUP_SIZE)
.task("vmatmul",
    TransformerComputeKernels::matrixVectorGeneric, context,
    state.wrapXb, state.wrapV, weights.wvLayered[layer].
    asHalfFloatArray(),
    config.dim(), config.kvDim(), LOCAL_WORK_GROUP_SIZE)
// Rotary position embedding
.task("rope", TransformerComputeKernels::ropeRotation,
    context,
    state.positionHolder, state.wrapQ, state.wrapK, config.
    kvDim(), config.headSize())
// Multi-head attention
.task("attention",
```

```
TransformerComputeKernels::processHeadsParallel,
state.wrapQ, state.wrapKeyCache, state.wrapValueCache,
state.wrapXb,
config.numberOfHeads(), config.headSize(), config.kvDim(),
config.kvMul(),
config.contextLength(), state.positionHolder, state.wrapAtt,
layer)
```

Each of these Tasks corresponds to of the key algorithms of the underlying transformer architecture. For example, the attention kernel implements scaled dot-product attention with explicit numerical stability handling. This is the same algorithm that powers every modern transformer, expressed in pure Java:

```
private static void processHeadTornado(FloatArray Q,
FloatArray keyCache,
    FloatArray valueCache, FloatArray output, int head,
    int headSize,
    int kvDim, int kvMul, long layerOffset, int pos,
    FloatArray attScores) {

    int headOffset = head * (pos + 1);

    // Step 1: Compute attention scores Q·K^T / sqrt(d_k)
    for (int t = 0; t <= pos; t++) {
        int kvHeadIdx = head / kvMul;
        int keyOffset = (int)(layerOffset + t * kvDim +
            kvHeadIdx * headSize);
        float score = 0.0f;
        for (int i = 0; i < headSize; i++) {
            score += Q.get(head * headSize + i) * keyCache.
                get(keyOffset + i);
        }
        attScores.set(headOffset + t, score / TornadoMath.
            sqrt(headSize));
    }

    // Step 2: Softmax with numerical stability (max
    subtraction)
    float maxScore = attScores.get(headOffset);
    for (int t = 1; t <= pos; t++) {
        maxScore = TornadoMath.max(maxScore, attScores.
            get(headOffset + t));
    }
    float sum = 0.0f;
    for (int t = 0; t <= pos; t++) {
        float exp = TornadoMath.exp(attScores.get(headOffset +
            t) - maxScore);
        attScores.set(headOffset + t, exp);
        sum += exp;
    }
    for (int t = 0; t <= pos; t++) {
        attScores.set(headOffset + t, attScores.get(headOffset
            + t) / sum);
    }
}
```

```
// Step 3: Weighted sum of values
for (int i = 0; i < headSize; i++) {
    float weightedSum = 0.0f;
    for (int t = 0; t <= pos; t++) {
        int valueOffset = (int)(layerOffset + t * kvDim +
            (head/kvMul) * headSize);
        weightedSum += attScores.get(headOffset + t) *
            valueCache.get(valueOffset + i);
    }
    output.set(head * headSize + i, weightedSum);
}
}
```

FP16 (half-precision) support reduces memory bandwidth requirements by half while also improving computational throughput. Model weights are loaded as HalfFloatArray objects using TornadoVM's off-heap types, and the asHalfFloatArray() accessor converts them into a GPU-compatible format. Benchmarks indicate approximately a **30% inference speedup** when dequantization is fused directly into the compute operations within the kernels.

LangChain4j and Quarkus Integration Enables Enterprise Deployment Patterns

GPULLama3.java achieved official LangChain4j model provider status since version 1.7.1, making GPU-accelerated inference a first-class option alongside cloud APIs. The library implements ChatLanguageModel and StreamingChatLanguageModel interfaces through builder-pattern APIs that mirror other providers:

```
// Synchronous inference
GPULLama3ChatModel model = GPULLama3ChatModel.builder()
    .modelPath(Paths.get("llama-3.2-1b-instruct-fp16.gguf"))
    .temperature(0.9)
    .topP(0.9)
    .maxTokens(2048)
    .onGPU(Boolean.TRUE) // Toggle GPU/CPU execution
    .build();

ChatResponse response = model.chat(ChatRequest.builder()
    .messages(UserMessage.from("Explain quantum
    entanglement"))
    .build());
```

The onGPU(Boolean) parameter provides transparent switching between GPU-accelerated TornadoVM execution and CPU-based Vector API execution using the underlying Llama3.java implementation. This enables graceful degradation on systems without GPU support.

Streaming responses support real-time chat interfaces:

```

GPULLlama3StreamingChatModel streamingModel =
GPULLlama3StreamingChatModel.builder()
    .modelPath(modelPath)
    .onGPU(true)
    .build();

streamingModel.chat(request, new
StreamingChatResponseHandler() {
    @Override
    public void onPartialResponse(String token) {
        System.out.print(token); // Stream tokens as
        generated
    }
    @Override
    public void onCompleteResponse(ChatResponse response) { }
    @Override
    public void onError(Throwable error) { }
});

```

Quarkus integration via `quarkus-langchain4j-gpu-llama3` enables declarative configuration and automatic model downloading:

```

# application.properties
quarkus.langchain4j.gpu-llama3.enable-integration=true
quarkus.langchain4j.gpu-llama3.chat-model.model-name=unsloth/
Llama-3.2-1B-Instruct-GGUF
quarkus.langchain4j.gpu-llama3.chat-model.quantization=F16
quarkus.langchain4j.gpu-llama3.chat-model.temperature=0.7
quarkus.langchain4j.gpu-llama3.chat-model.max-tokens=1024

```

The extension auto-injects `ChatModel` beans, enabling AI Services without explicit wiring:

```

@Path("/chat")
public class ChatResource {
    private final ChatModel chatModel; // Injected
    automatically

    public ChatResource(ChatModel chatModel) {
        this.chatModel = chatModel;
    }

    @GET
    @Path("/blocking")
    public String chat(@QueryParam("q") String question) {
        return chatModel.chat(question);
    }
}

```

Streaming endpoints leverage Mutiny's reactive types for non-blocking responses. One critical limitation: **TornadoVM does not support GraalVM Native Image compilation**, so Quarkus

applications must run in JVM mode with TornadoVM's JVM arguments specified at launch.

Production Deployments that Prove Enterprise Readiness

TornadoVM's most prominent production deployment supports the **European Space Agency's Gaia mission** through the AERO (Accelerated European Cloud) project. Gaia surveys **1.8 billion stellar objects** with 24 micro-arcsecond positional precision, generating massive computational loads – the third data release required **534 days of computation across 135 machines**. The processing pipeline runs in Java, making TornadoVM's GPU acceleration directly applicable without rewriting in CUDA. The primary optimization target is the Period Search algorithm for analyzing irregularly sampled time series, computing periodograms across 300,000-600,000 frequencies per source. With 12 million stellar objects and complexity scaling with frequency count times series length, **each 50ms improvement per source saves 166 hours** of total computation time.

On the contrary, `GPULLlama3.java` performance varies on different hardware devices and model sizes. On an **NVIDIA RTX 5090 GPU**, Llama-3.2-1B-Instruct achieves **96.65 tokens/second**; on an **NVIDIA RTX 4090 GPU**, **66.07 tok/s**; and on an **NVIDIA RTX 3070 GPU**, **52 tok/s**. On an **Intel Arc A770 GPU**, it reaches **15.65 tok/s** while on **Apple M4 Pro** achieves **16.77 tok/s** via OpenCL. Note: At the current state of the project, `GPULLlama3.java` does **not utilize Tensor Cores**. Support for using these NVIDIA specialized units is coming in the near future.

Conclusion: Java's Competitive Position in GPU Computing Solidifies

TornadoVM and `GPULLlama3.java` together demonstrate that Java can compete directly with Python for GPU-accelerated AI workloads. There is no need for awkward bindings, but through a coherent programming model that feels native to the language. The TaskGraph abstraction, annotation-based parallelism, and off-heap memory allocation via its integration with Project Panama, provide ergonomics comparable to NumPy and PyTorch while maintaining Java's type safety and tooling ecosystem.

The key insight is not raw performance, native CUDA implementations with Tensor Core support will remain faster for specific hardware. Instead, the real milestone is that Java developers can now build fully GPU-accelerated libraries that they understand, integrate them with mainstream frameworks, such as LangChain4j and Quarkus, and deploy them in production environments ranging from IoT edge devices to powerful servers. This write-once, run-on-heterogeneous-hardware model reduces vendor lock-in while enabling incremental and pragmatic adoption of GPU acceleration within existing Java ecosystems.

TornadoVM's evolution to version 3.0.0, with support for JDK 21 and JDK 25 after almost a decade of sustained development, confirms that GPU acceleration in Java is no longer experimental, but a permanent and integral expansion of the JVM ecosystem.

Quick start:

Get started with GPU-accelerated Java with SDKMAN :

```
# Install TornadoVM with SDKMAN
sdk install tornadovm 2.2.0-openc1

# Verify devices
tornado --devices

# Verify version
tornado --version

# Run a your first Java program onto a GPU
java @$TORNADOVM_HOME/tornado-argfile -cp $TORNADOVM_HOME/
share/java/tornado/tornado-examples-2.2.0.jar uk.ac.
manchester.tornado.examples.compute.MatrixVectorRowMajor
```

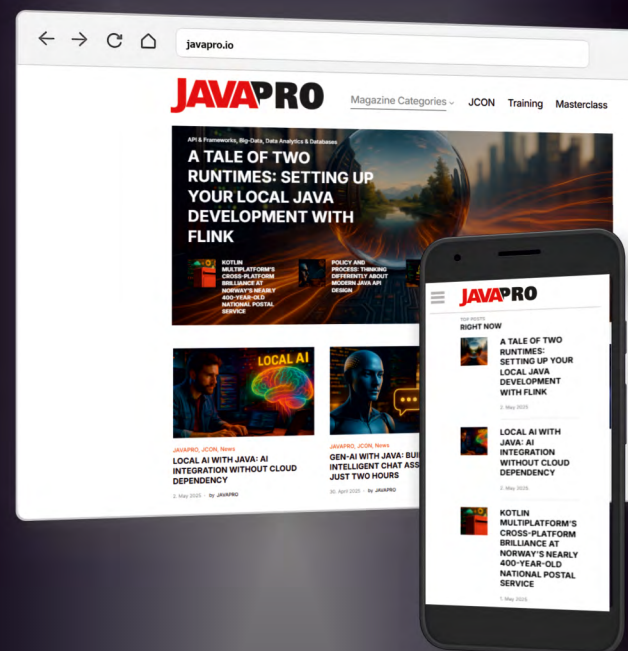
Acknowledgements

This work is supported by the European Union's Horizon Europe Programme under grant agreements 101070670 (ENCRYPT), 101092850 (AERO), 101093069 (P2CODE), and 101070052 (TANGO), and by UK Research and Innovation (UKRI) under the UK Government's Horizon Europe funding guarantee (ENCRYPT: 10039809; AERO: 10048318; P2CODE: 10048316; TANGO: 10039107)

-> Back to the Table of Content

JAVAPRO

**DISCOVER MORE GREAT
ARTICLES ONLINE:**



**BECOME A JAVAPRO
AUTHOR & BLOGGER !**



#JAVAPRO #COREJAVA

Reactor Has Died, Long Live REACTOR!

In this article, we try to prove that Reactor—and the entire reactive programming paradigm—is not dead. Quite the opposite: it has gained a second life. This happened with the arrival of JDK 21, when virtual threads significantly changed the landscape of concurrency in Java.

Business Problem

We need a sensible business example. This will show how reactive programming and virtual threads intertwine. We will use a simple model of airline ticket sales. It exposes a classic REST API.

Underneath, the API communicates with the airline's server to prepare the ticket.

Let's complicate things a bit. The purchase process requires separate calls for "outbound" and "return" flights.

Let's also assume a single query to the external server takes about 3 seconds. This is not necessarily an exaggeration under heavy network load. Such a delay will become crucial later in the article.

The Simplest Solution – Spring Boot

Implementing such an API in Spring Boot is extremely simple. In practice, it boils down to preparing two classes. We show these below:

```
@RestController
@RequiredArgsConstructor
public class BookingController {
    private final BookingService bookingService;

    @PostMapping(„/booking“)
    public String book(@RequestParam(value = „destination“)
String destination) {
        return bookingService.book(destination);
    }
}
```

Author:



Marcin Chrost is a software engineer with over 20 years of experience and IT trainer. Marcin Chrost has participated in many international projects and now regularly supports younger colleagues in their professional development. He likes novelties in the IT world, although he approaches them with common sense and reserve resulting from experience. What he appreciates the most, is well-coordinated and self-organizing teams that can motivate themselves. He also believes that soft skills in the IT world are as important as technical ones and are often underestimated.

```

@Service
@Slf4j
public class BookingService {
    enum TripType {
        THERE,
        BACK
    }

    public String book(String destination) {
        return String.join("\n",
            book(destination, TripType.THERE),
            book(destination, TripType.BACK)
        );
    }

    @SneakyThrows
    private String book(String destination, TripType tripType)
    {
        log.info("[{} {}] Booking start", destination,
            tripType);
        Thread.sleep(3000);
        log.info("[{} {}] Booking end", destination,
            tripType);
        return String.format("Booked %s travel to: %s",
            tripType, destination);
    }
}

```

As you can see, the code is clean and understandable. Even a beginner programmer should have no problem grasping it.

Does this code work? Functionally, absolutely. The example call in Postman below proves it:

```

POST http://localhost:8080/booking?destination=Katowice

HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8
Content-Length: 66
Date: Mon, 26 Jan 2026 12:17:51 GMT

Booked THERE travel to: Katowice
Booked BACK travel to: Katowice

Response code: 200; Time: 6192ms (6 s 192 ms); Content length:
66 bytes (66 B)

```

It's worth noting the response time is about 6 seconds. This is fully expected. We need to independently purchase two tickets—one way and return. Each of those calls takes roughly 3 seconds.

Performance Testing

The fact the code works for a single request offers no proof it will cope under heavy server load. To check application behavior in such a situation, we cannot conduct tests manually. We would

need to hire a thousand employees to click the endpoint, which makes little business sense.

For this reason, we need tools created specifically for performance testing. In the Java world, we most often reach for two solutions: Gatling and JMeter. I chose Gatling, because I prefer writing test scenarios in code form. Gatling also allows the use of Java, which is exactly what we need.

A sample test might look like this: we execute requests to our endpoint. We try to generate a load of 1000 users over 10 seconds:

```

public class BookingSimulation extends Simulation {

    AtomicInteger counter = new AtomicInteger(0);

    ChainBuilder bookTravel =
        exec(session -> session.set("bookingId", counter.
            getAndIncrement()))
        .exec(
            http("Post booking form")
                .post("/booking")
                .queryParams("destination", "Katowice-
                    #{bookingId}")
                .check(status().is(200))
        );

    HttpProtocolBuilder httpProtocol =
        http.baseUrl("http://localhost:8080");

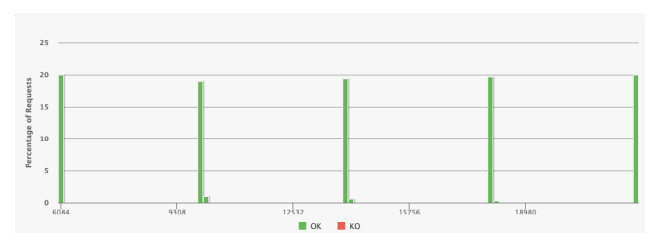
    ScenarioBuilder travelBookingScenario = scenario("Book").
        exec(bookTravel);

    {
        setUp(
            travelBookingScenario.injectOpen(rampUsers(1000).
                during(10))
        ).protocols(httpProtocol);
    }
}

```

After running the Gatling script—with the Maven command (`mvn gatling:test`)—we get the chart below.

As you can see, we encounter a few unpleasant surprises. Firstly, the average request processing time is not 6 seconds, as you might expect. Only about one-fifth of requests fit within that time. Each subsequent 20% group takes longer. The last one exceeds 20 seconds, which is absolutely unacceptable.

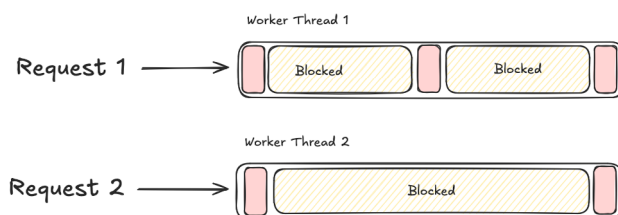


Let's also look at the second part of the chart. The server simply "stalls" for a certain period. It is unable to handle any new requests. This is a clear signal. The classic blocking approach begins to choke under heavier load:



The Problem with Thread Blocking

Why did the test results prove so poor? The reason is fundamentally trivial. Just look at the diagram below:



It clearly shows that for most of the time, the threads handling our requests do nothing. They wait for a response from the external server, wasting resources. Moreover, this diagram is quite optimistic. In real-world measurements, threads perform actual work for only about 1–2% of their lifespan.

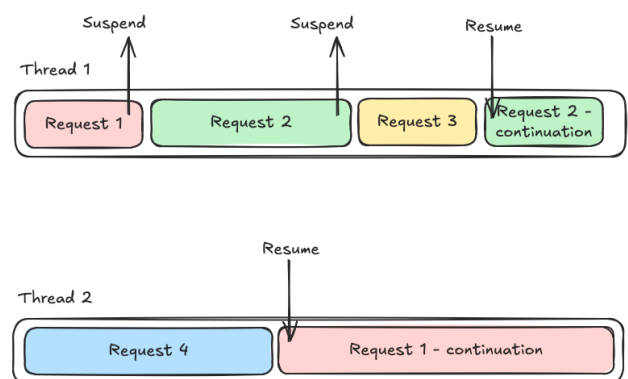
This also explains the shape of the previous chart. The default Tomcat thread pool (Spring Boot automatically starts it) is about 200 threads. Our test generated traffic from 1000 concurrent users. This means only 1/5 of requests can be processed immediately. The rest must wait in a queue. This is exactly what the subsequent batches of response times visible in the diagram correspond to.

Since things are so bad, why do people still commonly use the blocking model? Despite its performance limitations, it has one very important advantage. It provides a kind of "Shakespearean unity of time, place, and action." One thread handles one request. One thread is only busy with one request at a time. Thanks to this, we can effortlessly use various mechanisms based on ThreadLocal, such as MDC and SecurityContext. Programmatically, this approach is extremely simple, intuitive, and most importantly, most frameworks build upon it by default.

Spring Boot in Reactive Version

The main problem we encountered was the wasting of a single thread's time. It is worth asking: what if we utilized threads to the maximum? They wouldn't waste time waiting idly.

However, such an approach requires a different philosophy. We must enter the world of the so-called event loop. We perform an operation until we encounter a blocking fragment. When this happens, we pass the task back to the *event loop*. We wait for the appropriate callback to inform us we can continue. In the meantime, the thread, instead of waiting, can handle subsequent requests. The diagram below illustrates this concept:



This non-blocking, event-driven approach is called reactive programming. It often goes hand-in-hand with stream processing and the functional paradigm. This forms the foundation of libraries like RxJava or Reactor.

Spring Boot provides a reactive technology stack (with Reactor used under the hood). We will use it to shift our example into the world of reactivity. So let's see what the code looks like after this transformation:

```
@RestController
@RequiredArgsConstructor
public class RxBookingController {
    private final RxBookingService bookingService;

    @PostMapping("/booking")
    public Mono<String> book(@RequestParam(value =
        „destination“) String destination)
        return bookingService.book(destination);
    }
}
```

```
@Service
@Slf4j
public class RxBookingService {
    enum TripType {
        THERE,
        BACK
    }
}
```



```

public Mono<String> book(String destination) {
    return Mono.zip(
        book(destination, TripType.THERE),
        book(destination, TripType.BACK),
        (there, back) -> String.join(",\n", there, back)
    );
}

private Mono<String> book(String destination, TripType
tripType) {
    return Mono.fromCallable(() -> destination)
        .doOnNext(s -> log.info("{{}} Booking start",
            destination, tripType))
        .delayElement(Duration.ofSeconds(3))
        .map(d -> String.format("Booked %s travel to: %s",
            tripType, d))
        .doOnNext(s -> log.info("{{}} Booking end",
            destination, tripType));
}
}

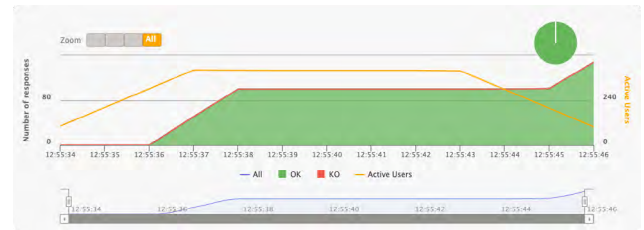
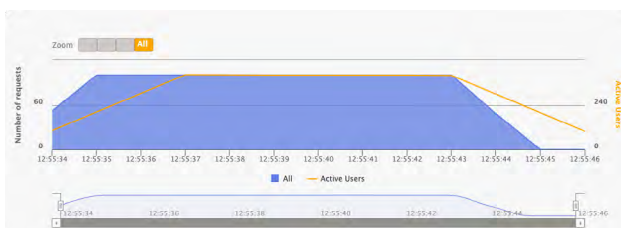
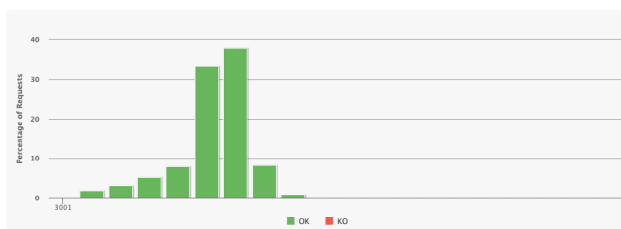
```

The controller code changes only cosmetically. The `Mono` type (from Reactor library) in the returned result is the only clear signal we are dealing with a reactive stack. `Mono` represents a “promise” to deliver a value in the future, not a concrete value itself.

Significantly larger differences appear in the service layer. In the reactive and streaming approach, you must write the code as operations that build a stream. Individual processing stages are connected. Everything must fit within a “chain of reactions.” Reactive streams are inherently lazy. Nothing executes until someone subscribes to the stream. This approach offers powerful possibilities. However, it requires thinking about the program in a completely different way than in the blocking model.

Does the Reactive Approach Make Sense?

To answer the question of whether the reactive approach actually solves our problems, let’s run the test again using Gatling. We will then analyze the results.



As you can see, changing to the reactive model ensures all requests are handled in about 3 seconds. (Why not 6? We will discuss this later when we return to the `Mono.zip` operator). More importantly, the system does not experience any stalls. Requests are processed continuously. The difference compared to the blocking approach is clearly positive.

Additionally, we can look at the number of threads used during the test. We will notice there were only 32 of them. This is a huge leap in quality compared to the previous 200. We use just over 10% of the original number of threads. We achieve much better performance.

However, the reactive approach is not without its drawbacks. The code becomes less readable. Understanding and maintenance require greater experience, usually at least at the regular, if not senior developer, level. Moreover, the learning curve is quite steep. The reactive philosophy forces a completely different way of thinking about data flow.

Another significant problem is the “viral” nature of reactive types. `Mono` and `Flux` tend to permeate the entire public API. This can be undesirable from a business perspective.

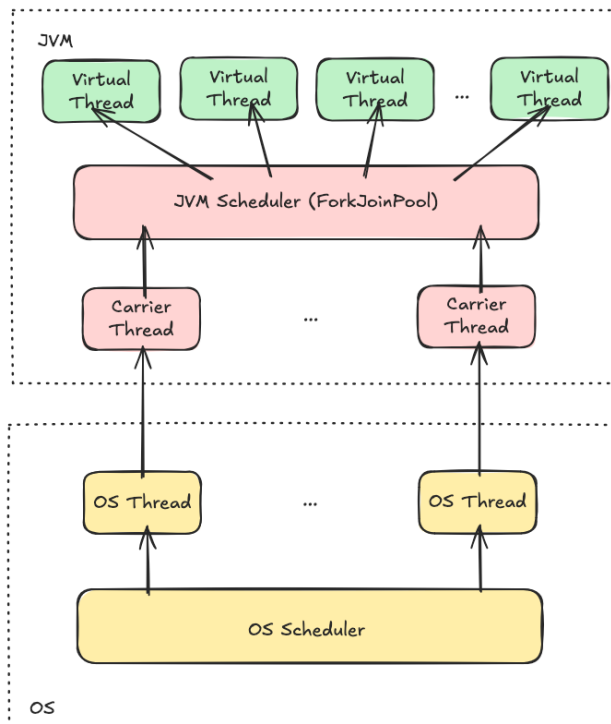
In summary: the reactive approach can be a very good solution if we need high performance and handling a large number of simultaneous requests. This comes at the cost of greater code complexity and reduced readability. However until JDK 21’s arrival, we had almost none alternatives here.

Java 21 and Virtual Threads

The situation changed dramatically with the arrival of JDK 21. This included one of the most anticipated features: **virtual threads**. What is this change based on? Essentially, something surprisingly simple: if a thread encounters a blocking operation, the virtual machine can *suspend* it. Other tasks can execute in the meantime.

We now deal with two types of threads. Before, there was only one. They divided the `Thread` class into two subclasses: classic **platform threads** (system VM threads) and new **virtual threads**.

The assumption is the JVM prepares a small pool of so-called *carrier threads*. These run the virtual thread code. When a carrier thread is assigned to a virtual thread, it begins executing code. If it hits a blocking operation, the JVM “parks” it. It saves its state in memory. The *carrier thread* immediately receives another ready-to-run virtual thread. When the awaited operation finishes, the virtual thread returns to further work, perhaps on a different *carrier thread*. Whole picture looks as follows:



Superficially, this looks very similar to the reactive model. However, the difference is fundamental. **Everything happens completely transparently to the programmer.** We do not have to manually manage non-blocking calls, callbacks, or reactive streams. The JVM takes care of it. For us, the code remains sequential, readable, and completely “non-reactive.”

For such an approach to work, deeply reorganizing the implementation of the `Thread` class was necessary. The entire I/O stack—network, file, etc.—directly in the virtual machine also needed reorganization. This work happened as part of the JEP introducing virtual threads.

Virtual Threads in Action

Virtual threads bring many advantages. The natural question arises: is their use in our application difficult? The answer is: **absolutely not**. Setting just one property in the `application.properties` file is enough in this case:

```
spring.threads.virtual.enabled=true
```

Let’s look at how the use of virtual threads affects system performance. We will use exactly the same code as in the original, non-reactive Spring Boot example. We will only add the mentioned setting. Then we will run the tests in Gatling again and compare the results.

It turns out that after enabling virtual threads, the application **handles all requests** without any problem. It achieves performance comparable to the reactive approach. However, there is one significant difference. The service time for a single request is **6 seconds**. In the reactive version, it was **3 seconds**.

Where does this discrepancy come from? Virtual threads were supposed to eliminate the main differences between blocking and reactive systems. This is because simply enabling virtual threads **does not automatically cause parallel execution of operations**. In the reactive model, we got this “for free” thanks to the `Mono.zip` operator. In the case of classic, sequential code, we still execute two queries one after the other. Hence the full 6 seconds.

So we see we still face the challenge of parallelizing operations. This is the classic problem of concurrency.

Classic Concurrency

When it comes to concurrency in Java, we have several approaches. The first, most classic, relies on using `ExecutorService`. We run the ticket purchase processes in parallel threads. This is shown in the example below:

```
@Service
@Slf4j
public class BookingService {

    enum TripType {
        THERE,
        BACK
    }

    /*
     * Default ThreadFactory used by StructuredTaskScope does
     * not set names for created threads
     * which makes threads logging unusable
     */
    private final ThreadFactory threadFactory = Thread.
ofVirtual().name(„booking-“, 0).factory();
    private final ExecutorService executorService = Executors.
newCachedThreadPool(threadFactory);

    @SneakyThrows
    public String book(String destination) {
        Future<String> there = executorService.submit(() ->
book(destination, TripType.THERE));
        Future<String> back = executorService.submit(() ->
book(destination, TripType.BACK));
        return String.join(„\n“,
            there.get(),
            back.get()
        );
    }

    @SneakyThrows
    private String book(String destination, TripType tripType)
    {
        log.info(„[{} {}] Booking start“, destination,
tripType);
        Thread.sleep(3000);
        log.info(„[{} {}] Booking end“, destination,
tripType);
        return String.format(„Booked %s travel to: %s“,
tripType, destination);
    }
}
```

```

    }
}

```

The code itself is not complicated. Theoretically, it differs little from what we had at the beginning of the project creation. Moreover, its performance - as Gatling can easily verify - would be practically identical to the reactive solution. Requests would be handled smoothly. The average response time would be about 3 seconds. Sounds like a big success, right?

Unfortunately, that's only half the truth. Classic concurrency in Java is associated with several significant pitfalls:

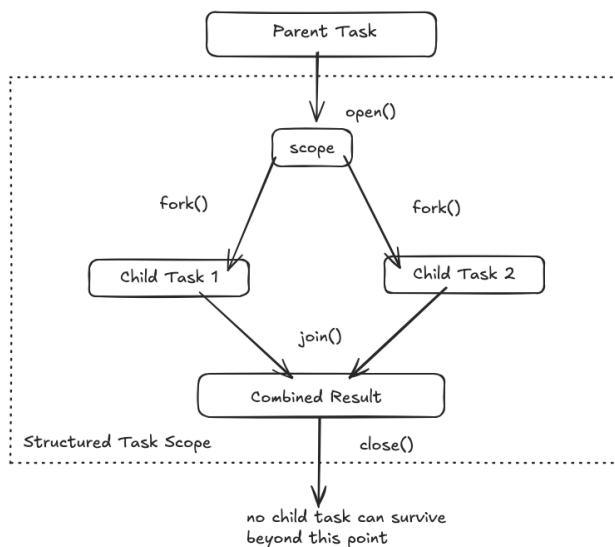
- **thread leak** - if the parent thread finishes with an exception, child threads may continue to run, not knowing their results are no longer needed,
- **unnecessary waiting** - sometimes we wait for an operation to complete, even though other parallel tasks have already failed,
- **problem with grouping results** - the final result requires manually collecting the results of all tasks. This affects the execution time. This cannot be done as elegantly as with the `Mono.zip` operator in the reactive approach.

However, all these problems can be solved. This is thanks to the approach known as **structured concurrency**. We will talk about this in a moment.

Structured Concurrency

Structured concurrency relies on one simple paradigm:

The lifetime of child tasks is strictly linked to the lifetime of the parent task and cannot exceed it.



This means that the moment the parent task finishes, we have a full guarantee all child tasks have also finished. In practice, this eliminates the possibility of thread leakage. Additionally, such a paradigm forces the existence of a policy. This policy specifies

what the result of the parent task execution should be when one of the child tasks ends with an error. Thus, the other two problems of classic concurrency we talked about earlier also disappear.

JDK 21 introduced structured concurrency as a preview feature. It was then significantly improved (in a backward-incompatible way, unfortunately!) in JDK 25. In our example, we will use JDK 25. We will show how to easily use the classes made available there to achieve the intended effect:

```

@Service
@Slf4j
public class BookingService {

    enum TripType {
        THERE,
        BACK
    }

    /*
     * Default ThreadFactory used by StructuredTaskScope does
     * not set names for created threads
     * which makes threads logging unusable
     */
    private final ThreadFactory threadFactory = Thread.
        ofVirtual().name("booking-", 0).factory();

    @SneakyThrows
    public String book(String destination) {
        try (var scope = new StructuredTaskScope.
            ShutdownOnFailure("Booking", threadFactory)) {
            Supplier<String> there = scope.fork(() ->
                book(destination, TripType.THERE));
            Supplier<String> back = scope.fork(() ->
                book(destination, TripType.BACK));
            scope.join().throwIfFailed();
            return String.join("\n", there.get(), back.get());
        }
    }

    @SneakyThrows
    private String book(String destination, TripType tripType)
    {
        log.info("[{} {}] Booking start", destination,
            tripType);
        Thread.sleep(3000);
        log.info("[{} {}] Booking end", destination,
            tripType);
        return String.format("Booked %s travel to: %s",
            tripType, destination);
    }
}

```

As you can see, the presented code is as simple as in the case of traditional concurrency. Yet, it removes all the problems the classic approach generated. One might think we have reached the "holy grail" we were looking for. And indeed, that might be the case... if it weren't for one detail. This detail is clearly visible when opening the `pom.xml` file:

```

<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <compilerArgs>--enable-preview</compilerArgs>
      </configuration>
    </plugin>
    ...
  </plugins>
</build>

```

Unfortunately, structured concurrency is still treated as a preview functionality. It will only be available in a fully production version when it hits an LTS release. This will most likely not happen before the premiere of JDK 29. This means that despite its huge potential, this functionality is not yet suitable for use in production code. As a result, this forces us to look for another alternative.

And at this point, we naturally return to our Reactor.

Reactor – A Second Birth

Until virtual threads arrived, blocking Reactor's worker threads (which make up the event-loop) was treated as a serious error. This was because the number of such threads was small. Each blocking operation caused a significant drop in the performance of the entire framework. However, with the introduction of virtual threads, this limitation lost its significance. Virtual threads can be blocked with impunity. This means we can afford to write part of the logic based on Reactor operators. We can then call the `block()` method. This transforms the reactive flow into blocking code. It gets rid of `Mono` / `Flux` objects in our API.

The only question that arises is: why does this approach even make sense? The answer lies in the rich set of Reactor operators. We have a situation where we want to perform two parallel operations and then combine their results. This is a difficult task in traditional concurrency. Structured concurrency facilitates this process. However, it is still not a production-ready solution. Meanwhile, the `Mono.zip` operator gives us this functionality "for free."

Let's notice one more thing: to use Reactor, **we do not have to** drag the entire reactive stack. This includes the non-blocking HTTP server/client and all additional dependencies. We can include only **Reactor Core itself** in the application. This is very lightweight and does not introduce unnecessary elements. This is completely sufficient to use operators such as `Mono.zip`. Let's see what the dependency in our `pom.xml` would look like:

```

<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
  <version>3.6.10</version>
</dependency>

```

Of course, let's also remember that for the whole thing to work correctly, we must enforce the use of virtual threads on the server side. Fortunately, this is still trivially simple. It boils down to setting the same property in `application.yml` that we discussed earlier.

Finally the code looks like as follows:

```

@Service
@Slf4j
public class BookingService {
    enum TripType {
        THERE,
        BACK
    }

    public String book(String destination) {
        return Mono.zip(
            book(destination, TripType.THERE),
            book(destination, TripType.BACK),
            (there, back) -> String.join("\n", there, back)
        ).block();
    }

    private Mono<String> book(String destination, TripType tripType) {
        return Mono.fromCallable(() -> destination)
            .doOnNext(s -> log.info("[{} {}] Booking start", destination, tripType))
            .delayElement(Duration.ofSeconds(3))
            .map(d -> String.format("Booked %s travel to: %s", tripType, d))
            .doOnNext(s -> log.info("[{} {}] Booking end", destination, tripType));
    }
}

```

If we were to test the solution for performance now, it would turn out everything works just as well as in a fully reactive application. This is true both in terms of request handling and response time, which again is about 3 seconds. So we are almost at the end of our journey.

We still have one more issue. The code used to communicate with the external airline server in our example is already non-blocking. However, in most real systems, it will not be. We will therefore have to "wrap" the existing blocking API in an appropriate reactive abstraction. Fortunately, with the use of virtual threads, this is not a major problem.

The Last, Missing Piece of the Puzzle

To hook up our existing, blocking solution, we only need to use Reactor's `subscribeOn` operator. This forces the execution of blocking code in virtual threads. By default, Reactor cannot delegate such tasks to virtual threads on its own. Preparing a dedicated `ThreadFactory` is therefore necessary. We can afford a very large "reserve" here. Virtual threads can easily handle even

a million active instances or a million elements in the queue. This would be completely unattainable in the case of traditional platform threads.

After preparing the appropriate `ThreadFactory`, it is enough to build a Reactor scheduler based on it. We then use it in the `subscribeOn` operator. We hook up our existing blocking API there. And that's pretty much it. So let's see what the final result of our work looks like:

```
@Service
@Slf4j
public class BookingService {
    enum TripType {
        THERE,
        BACK
    }

    /*
     Default boundedElastic() scheduler behaves like a hybrid.
     It creates virtual threads but is capped
     like they would have been platform ones (10 * number of
     cores)
     It is possible to change these settings via system
     properties,
     however I decided just to create my own scheduler
     */

    private final ThreadFactory virtualThreadFactory = Thread.
ofVirtual()
        .name("vt-factory-", 0)
        .factory();

    private final Scheduler virtualScheduler = Schedulers
        .newBoundedElastic(1_000_000, 1_000_000,
        virtualThreadFactory, 60);

    private final Scheduler scheduler = virtualScheduler;

    public String book(String destination) {
        return Mono.zip(
            book(destination, TripType.THERE).
            subscribeOn(scheduler),
            book(destination, TripType.BACK).
            subscribeOn(scheduler),
            (there, back) -> String.join("\n", there,
            back)
        ).block();
    }

    private Mono<String> book(String destination, TripType
    tripType) {
        return Mono.fromCallable(() ->
            bookInternal(destination, tripType))
            .doOnSubscribe(s -> log.info("[{} {}] Booking
            start", destination, tripType))
            .doOnNext(s -> log.info("[{} {}] Booking end",
            destination, tripType));
    }
}
```

```
@SneakyThrows
private String bookInternal(String destination, TripType
    tripType) {
    Thread.sleep(3000);
    return String.format("Booked %s travel to: %s",
        tripType, destination);
}
}
```

If we run the performance tests again, it turns out the whole thing works exactly as we expected. The total execution time of the operation still averages about 3 seconds. The system maintains full performance with a small number of platform threads. Virtual threads take over the entire burden associated with blocking communication. This relieves the event-loop and ensures high throughput.

The role Reactor plays in the whole puzzle is also very clearly visible. It becomes an **intermediate layer**. On one hand, it can handle blocking code (e.g., communication with an external server). On the other hand, it allows the API to be exposed in its previous, completely non-reactive form. This is done without cluttering it with types such as `Mono` or `Flux`. Thanks to this, we gain the best features of both worlds. This includes the simplicity of classic imperative programming and the strong possibilities for parallelization and combining results offered by Reactor operators.

Summary

The rumors of the "death" of reactive environments have proven to be greatly exaggerated. In practice, they can still find a very good place in the new ecosystem dominated by virtual threads. Reactor today provides us with what will only be available in the production version of structured concurrency. This includes convenient combining of results, stream composition, and a rich set of operators. These perfectly complement the capabilities of virtual threads.

As a result, we get an environment in which we can write code in a classic, imperative style. At the same time, we use the powerful parallelization mechanisms provided by Reactor. This is done without the need for a full migration to the reactive stack. It also avoids "infecting" the API with reactive types.

You can find the entire code presented in this material in the repository: <https://github.com/chrosciu/travel-agency>. I encourage you to clone it and experiment independently. It's a great way to experience the differences between individual approaches firsthand. It also helps to better understand how virtual threads work in practice.

-> Back to the Table of Content



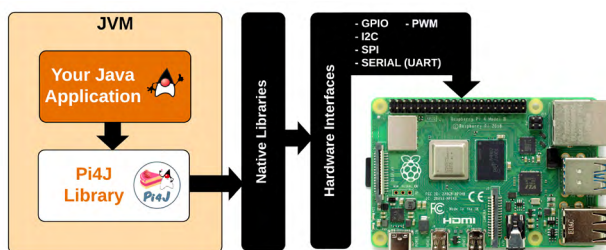
#JAVAPRO #FRAMEWORKS #API

The FFM API: OpenJDK Changed the Game for Native Interactions and #JavaOnRaspberryPi



If you're working with Pi4J, you're essentially using a Java library that bridges the gap between your Java application and the GPIO hardware on a Raspberry Pi. The beauty of Pi4J lies in how it hides the complexity of native library integration and the Java Native Interface (JNI). It lets you focus on building great applications rather than dealing with low-level system calls.

My involvement with Pi4J started around 2020 while writing the book „Getting Started with Java on the Raspberry Pi“. Despite years of contributing to the project, the deeper layers of Pi4J's codebase still challenge me. Take a look at this snippet and tell me if you understand what's going on:



Author:



#JavaOnRaspberryPi - Java Champion - Frank Delporte is Author of „Getting started with Java on Raspberry Pi“ - Senior Technical Writer at Azul - Team member Pi4J - Co-Organizer BeJUG - Lead coach CoderDojo Belgium Ieper

<https://www.linkedin.com/in/frankdelporte/>

```
JNIEXPORT jobject JNICALL Java_com_pi4j_library_gpiod_
internal_GpioD_c_1gpiod_1chip_1open
(JNIEnv* env, jclass javaClass, jstring path) {
    struct gpiod_chip* chip;
    const char* nativeString = (*env)->GetStringUTFChars(env,
        path, NULL);
    chip = gpiod_chip_open(nativeString);
    (*env)->ReleaseStringUTFChars(env, path, nativeString);
    if(chip == NULL) {
        return NULL;
    }
    jclass cls = (*env)->FindClass(env, „java/lang/Long“);
    jmethodID longConstructor = (*env)->GetMethodID(env, cls,
        „<init>“, „(J)V“);
    return (*env)->NewObject(env, cls, longConstructor,
        (jlong) (uintptr_t) chip);
}
```


Honestly, I, as a Java developer, have no clue... It's one of those points where Java meets native libraries for GPIO communication. Through a combination of JNI, Java Native Access (JNA), and Docker-based build pipelines for compiling native components, Pi4J presents a clean interface to users while creating maintenance headaches for the development team.

This article explores how the Foreign Function & Memory API (FFM API) has transformed native library interaction in Java, bringing significant improvements to projects such as Pi4J.

Looking Back at Java's Evolution

Fifteen years ago, I moved from C# development to Java and haven't regretted it for a moment. That transition happened right at Java's 15-year mark, halfway through what is now a three-decade journey. I've been an active observer of Java's growth ever since and had the privilege of interviewing James Gosling, Java's creator, for the Foojay Podcast.

The six-month release cycle, driven by OpenJDK projects and JDK Enhancement Proposals (JEPs), has delivered remarkable improvements, including enhanced switch expressions, virtual threads, continuous performance optimization, and more. In this article we focus on one of these innovations, delivered within **Project Panama**: the Foreign Function & Memory API (FFM API).

Understanding the Foreign Function & Memory API

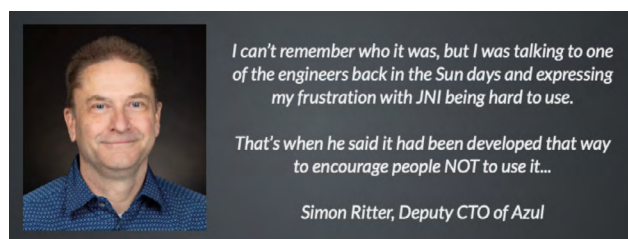
Java 22 marked the official arrival of the FFM API as a fully integrated feature. This milestone represents extensive work under Project Panama's umbrella, driven by three core objectives:

1. **Memory safety:** Secure handling of off-heap memory with guaranteed resource cleanup.
2. **Easy interaction:** Straightforward mechanisms for native library integration.
3. **High performance:** Performance that rivals or surpasses traditional JNI.

The JNI Challenge

Since Java 1.1, JNI has served as the bridge to native code. While functional, it brings several significant pain points:

- **Manual memory management** requires deep expertise.
- **Complex implementation** involving C header generation and compilation workflows.
- **Hard to use by design:** Simon Ritter once shared that a Sun engineer deliberately designed JNI to be difficult, hoping to discourage its use!



Various libraries, such as JNA and Java Native Runtime (JNR), have attempted to simplify native integration, but each has its own performance penalties and constraints.

FFM API's Development Journey

The FFM API's evolution demonstrates OpenJDK's approach to introducing new features. The work was structured into multiple JEPs, with initial deliveries appearing in Java 14. Notice how most features progressed through incubator or preview stages, requiring the `--enable-preview` flag, a process I explained in this Azul documentation page.

The initial phase, "**Foreign Memory Access API**," introduced a new approach for accessing off-heap memory safely and efficiently. These early versions remained in incubator status, providing the foundation for the next steps...

- OpenJDK 14: JEP 370: Foreign-Memory Access API (Incubator)
- OpenJDK 15: JEP 383: Foreign-Memory Access API (Second Incubator)
- OpenJDK 16: JEP 393: Foreign-Memory Access API (Third Incubator)

That next step, "**Foreign Linker API**," brought statically typed, pure-Java access to native code, still in incubator status.

- OpenJDK 16: JEP 389: Foreign Linker API (Incubator)

In the final step, the pieces of the puzzle got combined in the "Foreign Function & Memory API". After several incubator and preview releases, the final version became fully available in Java 22.

- OpenJDK 17: JEP 412: Foreign Function & Memory API (Incubator)
- OpenJDK 18: JEP 419: Foreign Function & Memory API (Second Incubator)
- OpenJDK 19: JEP 424: Foreign Function & Memory API (Preview)
- OpenJDK 20: JEP 434: Foreign Function & Memory API (Second Preview)
- OpenJDK 21: JEP 442: Foreign Function & Memory API (Third Preview)
- OpenJDK 22: JEP 454: Foreign Function & Memory API

This process enables the OpenJDK team to get community feedback and refine the API until it achieves the right balance of stability, performance, and usability.

Practical Examples

Let's take a look at a few simple examples to illustrate how the FFM API can be used.

Direct Memory Operations

Here's how you can work with memory directly:

```
void main() {
    // Open a confined Arena that manages off-heap memory
    // and will release it automatically.
    try (Arena arena = Arena.ofConfined()) {
        // Allocate 5 ints in the arena
        MemorySegment segment = arena.allocate(ValueLayout.
            JAVA_INT, 5);

        // Fill the segment with random values for each int.
        System.out.print("Setting values: ");
        for (int i = 0; i < 5; i++) {
            int randomValue = new Random().nextInt(100);
            segment.setAtIndex(ValueLayout.JAVA_INT, i,
                randomValue);
            System.out.print(randomValue + " ");
        }
        System.out.println("");

        // Print the values back from memory.
        System.out.print("Reading values: ");
        for (int i = 0; i < 5; i++) {
            System.out.print(segment.getAtIndex(ValueLayout.
                JAVA_INT, i) + " ");
        }
        System.out.println("");
    }
}
```

Several aspects of this code are worth highlighting:

- Running on Java 25 lets me take advantage of the simplified `main` method from JEP 512: Compact Source Files and Instance Main Methods. No class boilerplate, no package declarations, but just the essential code for this straightforward demonstration.
- The `Arena` handles automatic cleanup within the try block.
- `MemorySegment` provides a clean abstraction over native memory addresses.
- Manual memory management becomes unnecessary.

```
$ java ArenaDemo.java
Setting values: 16 7 27 50 80
Reading values: 16 7 27 50 80
```

For a deeper dive, check out the expanded example in `FFMMemoryManagement.java`, alongside a Java 11 comparison in `Java11MemoryManagement.java` that demonstrates the old complexity. The Java 11 version requires JBang execution following the instructions in the file to enforce Java 11 usage.

Native Function Invocation

Let's create the world's most overengineered `String.length()` Implementation to demonstrate native function calls! This example shows how the FFM API enables clean native integration within a single, readable method.

```
void main() throws Throwable {
    // The text we will use in the demo.
    var text = "Hello JVM Advent!";

    // Obtain a Linker that knows how to call
    // native (C) functions on this platform.
    Linker linker = Linker.nativeLinker();

    // Create a SymbolLookup that can find symbols
    // (like C functions) in the default native libraries.
    SymbolLookup lookup = linker.defaultLookup();

    // Create a MethodHandle to call the native C function
    MethodHandle strlen = linker.downcallHandle(
        // Look up the address of the 'strlen' symbol,
        // or throw if it isn't found.
        lookup.find("strlen").orElseThrow(),
        // Describe the C function:
        // - returns long
        // - takes one pointer (address) argument
        FunctionDescriptor.of(ValueLayout.JAVA_LONG,
            ValueLayout.ADDRESS)
    );

    // Open a confined Arena that manages off-heap memory
    // and will release it automatically.
    try (Arena arena = Arena.ofConfined()) {
        // Allocate native memory for a C-style string
        // and copy "Hello World!" into it.
        MemorySegment str = arena.allocateFrom(text);

        // Call the native 'strlen' function via the
        MethodHandle,
        // passing the string's memory address,
        // and cast the result to a long.
        long length = (long) strlen.invoke(str);

        // Print the result.
        System.out.println("Length with native library: " +
            length);
    }

    System.out.println("Length with String.length(): " + text.
        length());
}
```

No complex JNI/JNA needed, no recompilation of C code, but just simple and readable Java code.

```
$ java LinkerDemo.java
Length with native library: 17
Length with String.length(): 17
```

Additional examples are available: extended FFM version in `FFMNativeCalls.java`, plus the Java 11 equivalent in `Java11NativeCalls.java`.

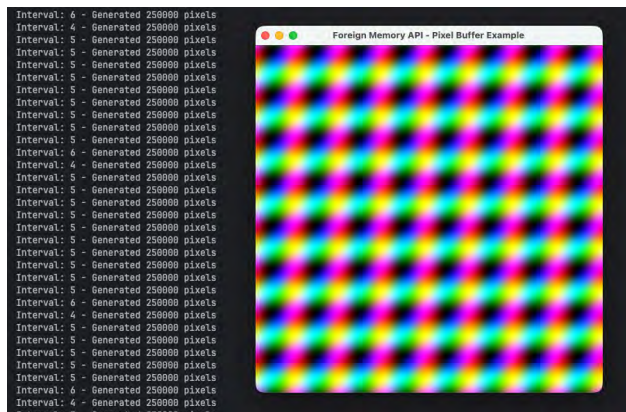
Benchmarking Performance

I'm a fan of demos with visual feedback, so I built a benchmark that generates animated gradients, a task that requires extensive memory writes. Both the Java 11 and 25 versions attempt a 5-millisecond refresh interval. The performance difference is striking (on macOS M2 with Azul Zulu 25 versus Java 11:

```
$ jbang Java11PixelBuffer.java
[jbang] Building jar for Java11PixelBuffer.java...
Interval: 57 - Generated 250000 pixels
Interval: 56 - Generated 250000 pixels

$ java FFMPixelBuffer.java
Interval: 5 - Generated 250000 pixels
Interval: 5 - Generated 250000 pixels
```

- **Java 11 implementation** with `BufferedImage` and arrays: ~50-60ms per frame.
- **FFM API implementation** using direct memory: ~5ms per frame.



FFM API's Impact on Raspberry Pi Development

Let me first answer a more general question I get asked a lot: **"Why Java on a Raspberry Pi?"** My answer is straightforward: Java allows me to create everything I need. I can develop JavaFX interfaces, integrate with web services, and use the many libraries available within Java's ecosystem. When I began exploring Java on Raspberry Pi over five years ago, learning another language wasn't my goal. I wanted to understand electronics and hardware interaction using familiar, proven tools.

Pi4J serves this purpose precisely: it's a Java library for controlling GPIO pins and connected electronics on Raspberry Pi. The complication? This library has always depended on native C/C++ code for hardware communication. Until recently, that meant dealing with the complexities of JNI and JNA.

How Pi4J Works

Pi4J implements a plugin architecture where different "providers" manage GPIO communication. Historically, these providers depended on native libraries via JNI/JNA, creating:

- Multi-layered complexity spanning up to 5 levels for GPIO operations.
- Docker-based build systems for native library compilation.
- Mysterious JNI header generation and arcane C implementations.

These plugins load dynamically during runtime.

The FFM Revolution

Pi4J's upcoming V4 release includes a nearly production-ready FFM-based provider. The improvements are substantial:

- Performance Gains
 - Reading GPIO input state: **10x performance increase!**
 - SPI communication: Notable improvement, though less dramatic.
- Code Quality
 - Direct Java-to-kernel communication paths.
 - Elimination of complex JNI layers.
 - Clear, maintainable Java code throughout.
 - No complex Docker build infrastructure needed.

Community Contribution SUCCESS

The FFM implementation has an even better backstory: it came from community contributions. Nick Gritsenko (@DigitalSmile) had already developed a Java 22 GPIO library using FFM. When I found his work, I asked him if we could use it in Pi4J. His response was better than expected as he contributed the code directly to Pi4J and adapted it perfectly to our plugin architecture! This led to a substantial pull request now merged into Pi4J V4.

I'm very proud that this happens again and again. Previously, Alexander Liggesmeyer contributed Raspberry Pi 5 support through a new plugin. Currently, Stefan Haustein, Stephen More, Tom Aarts, and other contributors are building an easy-to-use Pi4J drivers library with example implementations, making it a lot easier to work with complex components like joysticks, LCD displays, LED strips, and more.

Expanding Hardware Support

The FFM-based implementation opens up exciting possibilities. Since it relies on standard Linux kernel methods from Debian, Pi4J should theoretically support:

- Orange Pi and similar Linux-based Single Board Computers.
- RISC-V processors running Debian distributions.

I'm starting to experiment with these alternative platforms and hope to add more documentation about it on the Pi4J website soon.

Pi4J Code Examples with FFM

Because the FFM plugin in Pi4J fits perfectly into the architecture, there is nothing special you, as a developer, need to do. Include the plugin in your pom-file, instead of the plugin you were using before, and you're done!

Here's a condensed example using JBang to blink an RGB LED with Pi4J V4's snapshot build. After the official release, you can remove the `//REPOS` line and update the version number.

```

/// usr/bin/env jbang „$0“ „$@" ; exit $?

//REPOS mavencentral,pi4j-snapshots=https://oss.sonatype.org/
content/repositories/snapshots

//DEPS com.pi4j:pi4j-core:4.0.0-SNAPSHOT
//DEPS com.pi4j:pi4j-plugin-linuxfs:4.0.0-SNAPSHOT

import com.pi4j.Pi4J;
import com.pi4j.io.gpio.digital.*;

/**
 * Execute with `jbang Pi4JExample.java`
 */
void main() throws InterruptedException {
    var pi4j = Pi4J.newAutoContext();

    var red = pi4j.digitalOutput().create(17);
    var green = pi4j.digitalOutput().create(27);
    var blue = pi4j.digitalOutput().create(22);

    // Blink pattern
    for (int i = 0; i < 10; i++) {
        red.high();
        Thread.sleep(500);
        red.low();
        green.high();
        Thread.sleep(500);
        green.low();
        blue.high();
        Thread.sleep(500);
        blue.low();
    }
}

```

Find additional JBang-compatible examples in the Pi4J JBang repository.

Key Considerations

While preparing my presentation and this article about the FFM API, several important points emerged.

- **Memory Safety Warning:** FFM usage takes you outside the JVM's garbage collector safety net. Several examples included in this article, generate warnings about native

access. This is intentional as the OpenJDK team wants developers to recognize they're performing potentially hazardous operations. Add the `--enable-native-access` flag to acknowledge the risks and suppress warnings. Future OpenJDK versions may change this behavior.

- **Read JEPs To Learn More:** If Java's evolution interests you, I strongly recommend exploring JDK Enhancement Proposals. They're far more than technical specs! These documents explain design decisions, include practical examples, and offer insights from Java's architects. Similarly, you can follow OpenJDK project progress on their webpages and by subscribing to their mailing lists.
- **Follow Java Releases:** Java's six-month release cycle consistently delivers valuable improvements! Each version brings numerous bug fixes, optimizations, and new features emerging from projects and JEPs. If you can upgrade to a newer version, particularly for development environments, you should! While Java 22 introduced the FFM API, I learned at Devovx that Java 24 delivered significant FFM performance improvements without API changes! This demonstrates the OpenJDK team's ongoing commitment to enhancing existing features.

Wrapping Up

The FFM API represents a significant leap forward for developers working with native code. It eliminates JNI complexity while maintaining or improving performance. It creates new possibilities for Java in embedded systems and hardware integration. It will accelerate the closer integration of Java with Artificial Intelligence, Machine Learning, and Large Language Model development.

Don't hesitate to experiment and contribute. Projects like OpenJDK and Pi4J grow through community participation!

For Java on Raspberry Pi exploration:

- Visit the Pi4J website.
- Read my blog posts at webtechie.be and foojay.io.
- Check out my ebook "Getting Started with Java on the Raspberry Pi", continuously updated with new Java versions and Pi4J improvements.

Don't forget: We owe thanks to OpenJDK and its contributors for incredible features like the Garbage Collector, JIT compilation, Lambdas, Virtual Threads, and now the FFM API. Java continues improving, and that's something worth celebrating!

-> Back to the Table of Content



#JAVAPRO #FRAMEWORKS #API

Talk to Your Data: Natural Language Data Access in Java



For decades, Java developers have relied on robust frameworks to manage the complexity of enterprise applications. At the heart of this ecosystem lies Hibernate ORM, the popular library for bridging the gap between object-oriented Java applications and relational databases. It allows developers to map complex domain models to database tables, and write type-safe, performant queries using the Hibernate Query Language (HQL).

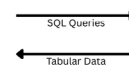
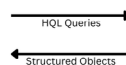
Quarkus has emerged as a Java framework for building applications and services, focusing on incredibly fast startup times, efficient resource consumption and developer joy.

Enter the new frontier: Artificial Intelligence. As Large Language Models transform how we interact with software, Java developers need tools to integrate these capabilities natively. This is where **LangChain4j** comes in. It is quickly emerging as a powerful library for AI integration in Java, offering a unified API that simplifies the complexity of LLM orchestration.

By combining these three powerhouses – Hibernate ORM, Quarkus, and LangChain4j – we can now achieve something that was previously incredibly difficult: allowing every user to “talk” to their data. This article explores the new **Hibernate Assistant** module, and how it can be used to make your structured relational data available in the unstructured world of AI.

Bridging the Gap: The Hibernate Assistant

Now we can explore the traditional data flow for a Java application using Hibernate to retrieve data from a relational database:



Large Language Models are trained on vast amounts of public data. They already know about Hibernate ORM, its persistence mapping logic, and the syntax of the Hibernate Query Language. However, they lack one critical piece of information: the specific context of your application. An LLM doesn't know how your tables look, nor your entities and their relationships.

Author:

Marco Belladelli is contributing to the ongoing success of Hibernate, the most widely used open-source Object/Relational Mapping framework for Java. His work focuses on driving innovation while ensuring the high reliability our users and customers expect. Hibernate serves as the reference data layer library for projects like WildFly (JBoss EAP) and Quarkus; it implements the Jakarta EE Persistence (JPA) specification and provides compatibility with most relational database vendors. He also actively engages with both the open-source community and enterprise customers, providing technical guidance and support.

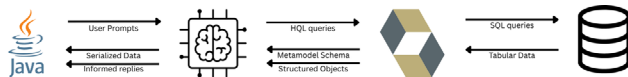


<https://www.linkedin.com/in/marco-belladelli/>

The **hibernate-assistant** module is designed to solve this problem. It acts as a translation layer, using Hibernate's existing knowledge of the application's mapping domain model to inform the AI model. Since Hibernate already knows which database table maps to a specific Java entity class, and which columns correspond to persistent attributes, it can programmatically describe this structure to an LLM.

The module offers tools to initialize the context by reporting on mapped entity classes and their properties. For example, it can instruct the model that a "Company" entity corresponds to the Java class `org.hibernate.assistant.domain.Company`, has an identifier "id", and includes attributes like its name and an `@Embedded` address component, as well as a one-to-many collection association of related Employee entities.

Furthermore, the module handles the complex task of transforming query results to feed to the AI model. While serializing simple SQL result sets might be easy, translating the results of an ORM query producing Java objects – which may include circular or lazy associations, collections, and embedded types – is not. Hibernate Assistant, once again, leverages its knowledge of your domain model to transform HQL query results into a textual format that the LLM can effortlessly consume and interpret.



Why Hibernate?

You might wonder, "Why not just let the LLM write SQL?" While possible, using Hibernate ORM as the intermediary offers significant advantages in reliability and ease of use:

- **Constrained Access:** When using Hibernate, data access is naturally constrained to the mapped domain model. The LLM can only access tables that have a corresponding entity class, and even then, it can only select columns listed as fields in your objects.
- **Fail-Early Validation:** Thanks to Hibernate's advanced type-safety and query validation, the system can "fail-early." We know LLMs are imperfect, and in case of invalid generated HQL statements, Hibernate detects it before it ever reaches the database, increasing overall reliability and performance.
- **Self-Correction:** If a query fails, the error message from Hibernate is informative and specific, regardless of the underlying database platform. The user can intercede to act on that error, or even feed it back to the LLM so it can correct its mistake in a subsequent prompt.
- **Portability:** Hibernate ORM supports a wide variety of databases, ensuring your application operates consistently regardless of the underlying infrastructure. It provides a single, unified query language (HQL) that abstracts away database-specific SQL variations.
- **Handling Complexity:** HQL makes it significantly easier to write complex queries involving multiple entities, associations, embeddable values, and inheritance hierarchies compared to raw SQL. LLMs are great at natural language processing, and HQL being a higher-level abstraction closer

to language helps AI agents achieve better results in these complex scenarios.

Building the Application

Let's now dive into the practical implementation by building a Quarkus application. This application will leverage the concepts we've just discussed to enable natural language interaction with your relational data.

Project Dependencies

You will need the standard Quarkus Hibernate ORM extension, the Quarkus LangChain4j extension (for your AI service provider of choice), and the new hibernate-assistant module.

```

<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkiverse.langchain4j</groupId>
  <artifactId>quarkus-langchain4j-ollama</artifactId>
</dependency>
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-assistant</artifactId>
</dependency>
  
```

You will also need a JDBC driver for your database and some basic configuration for the extensions.

Implementing the Assistant

The goal is to create a service that takes natural language prompts, understands if relevant information can be derived by your Hibernate mappings, generates a query to extract such information, and finally executes it. Optionally, the service can feed the results back to the LLM itself to provide an informed plain language reply directly to our users.

We can take advantage of the existing `ChatModel` and `ChatMemoryProvider` instances, from the Quarkus LangChain4J extension, and the `Metamodel` instance, from Hibernate ORM, that will be available in the CDI context. With that, we can create our own `@ApplicationScoped` bean that will be our `HibernateAssistant`:

```

import dev.langchain4j.memory.chat.ChatMemoryProvider;
import dev.langchain4j.model.chat.ChatModel;
import jakarta.persistence.metamodel.Metamodel;

private static final PromptTemplate METAMODEL_PROMPT_TEMPLATE =
    PromptTemplate.from(
        """
        You are an expert in writing Hibernate Query
  
```

```

Language (HQL) queries.
You have access to an entity model with the
following structure:
{{it}}
,,““ );

@Inject
ChatModel chatModel;
@Inject
ChatMemoryProvider memoryProvider;
@Inject
Metamodel metamodel;

public HibernateAssistantLC4J() {
    String metamodelJson = MetamodelJsonSerializerImpl.
        INSTANCE.toString(metamodel);
    SystemMessage systemMessage = METAMODEL_PROMPT_TEMPLATE.
        apply(metamodelJson).toSystemMessage();
    // We can now initialize the ChatMemory using the provider
    with the SystemMessage
}

```

We've completed the crucial first step for getting our assistant up and running: creating a `SystemMessage`, a context initialization directive that will instruct the LLM on how to respond to subsequent prompts. This was easily done thanks to the `MetamodelSerializer` implementation built into the `hibernate-assistant` module that feeds all the available mapping information from your Object/Relational Hibernate metamodel into a JSON format which can be easily consumed by large language models.

Now, we can use the assistant to create a Hibernate query from natural language:

```

public String executeAiQuery(String message,
    SharedSessionContract session) {
    UserMessage userMessage = UserMessage.from(message);
    chatMemory.add(userMessage);
    ChatRequest chatRequest = ChatRequest.builder().
        messages(chatMemory.messages()).build();
    ChatResponse chatResponse = chatModel.chat(chatRequest);

    String hql = chatResponse.aiMessage().text();
    SelectionQuery<Object> query = session.
        createSelectionQuery(hql, Object.class);
    List<Object> results = query.getResultList();
    return new ResultsJsonSerializerImpl(session.getFactory()).
        toString(results, query);
}

```

Under the hood, the assistant generates a valid HQL query tailored to your schema, relevant to the natural language request that was provided. For example, the user might ask "Extract all companies with a name starting with the letter A, regardless of case". And the assistant will produce the following HQL query:

```

SELECT c
FROM Company c
WHERE LOWER(c.name) LIKE ,a%

```

The query is executed using the provided Hibernate ORM Session, which in turn results in the following SQL statement:

```

select
    c1_0.id,
    c1_0.city,
    c1_0.street,
    c1_0.name
from
    company_table c1_0
where
    lower(c1_0.name) like ,a% escape ,

```

The query results are extracted, and the retrieved data is available for use in your application's logic.

In my original code snippet, I've gone a step further, converting the query results from Java objects into a textual format, specifically JSON, using the `hibernate-assistant` module's `ResultsSerializer`.

This seamlessly transforms a potentially complex data structure into a format that can easily be consumed by other application components or even fed back to the model itself for further reasoning and insights on the original prompt.

Retrieval-Augmented Generation (RAG)

For more advanced use cases, we might want to combine domain model and underlying database knowledge with the LLM's conversational abilities in a **retrieval-augmented generation pipeline**. Thanks to the Hibernate Assistant module, we can easily design an implementation of LangChain4j's designated RAG interface `ContentRetriever`.

This content retriever could use the instance of the Hibernate Assistant we've created earlier that will be available in the CDI context, as well as once again the current Hibernate ORM session. Using the assistant's methods we've shown in the previous example, it can interpret the user request and extract relevant information from the database through Hibernate. This information is then serialized in a textual format compatible with LLMs, and fed back to the LangChain4J RAG pipeline to provide an informed reply based on your data.

```

@Inject
HibernateContentRetriever contentRetriever;

interface HibernateAssistantRag {
    String chat(String userMessage);
}

```

```

public String ragResponse(String message,
    SharedSessionContract session) {
    RetrievalAugmentor rag = DefaultRetrievalAugmentor.
        builder()
            .contentRetriever(contentRetriever)
            .contentInjector(DefaultContentInjector.builder().
                promptTemplate(INJECTOR_PROMPT_TEMPLATE).build())
            .build();
    final HibernateAssistantRag assistant = AiServices.
        builder(HibernateAssistantRag.class)
            .chatModel(chatModel)
            .chatMemory(chatMemory)
            .retrievalAugmentor(rag)
            .build();
    return assistant.chat(message);
}

```

With this setup, when a user asks, “How many companies with at least 5000 employees can be found in the city of Modena”, the content retriever understands it can derive this information from your database, invokes the assistant which generates the relevant HQL query:

```

SELECT COUNT(c)
FROM Company c
WHERE c.address.city = 'Modena'
AND SIZE(c.employees) >= 5000

```

The query might look trivial, but it involves embedded properties and to-many associations. The model makes quick work of it thanks to its knowledge of the HQL syntax and its powerful capabilities. The query itself is executed completely transparently, within the RAG pipeline, and the extracted data is serialized and provided back to the model through the `RetrievalAugmentor` to generate an insightful natural language response.

The Result:

Now that we have access to our database through natural language, how can we make use of it?

Hibernate-powered Chatbot

The standard chatbot experience involves a user conversing interactively with an LLM through a simple chat-like interface. By integrating Hibernate Assistant’s data retrieval capabilities into this interface, we can enrich the AI model’s responses with relevant information or simply present the data in an easily understandable format.

Imagine an HTTP REST endpoint `/assistant` that accepts a user message. The service uses the assistant functionalities to interact with the database, through the Hibernate ORM extension, and retrieve relevant information. Here’s a simplified look at how that endpoint might be implemented with the Quarkus REST extension:

```

import org.hibernate.StatelessSession;
import org.hibernate.query.SelectionQuery;

import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.QueryParam;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

@Path("/assistant")
public class AssistantResource {
    @Inject
    StatelessSession session;
    @Inject
    HibernateAssistantLC4J assistant;

    @GET
    @Path("/json")
    @Produces(MediaType.TEXT_PLAIN)
    public Response queryToJson(@QueryParam("query") String
        query) {
        final String json = assistant.executeAiQuery(query,
            session);
        return Response.ok(json).build();
    }

    @GET
    @Path("/ask")
    @Produces(MediaType.TEXT_PLAIN)
    public Response naturalLanguage(@QueryParam("query") String
        query) {
        final String response = assistant.ragResponse(query,
            session);
        return Response.ok(response).build();
    }
}

```

The endpoint offers two methods. Both of them take a simple natural language request through the `query` parameter as input. The first simply interprets the request and returns the serialized results back in JSON format, which can easily be consumed by another service or the front-end itself. The second method uses the RAG pipeline we explored earlier to provide an informed natural language response back to the user.

And here is the result: The user asked a simple question to the first method, the assistant generated the query, fetched the Employee entities, and they’re now displayed in a simple table. The user subsequently asks questions and the assistant, thanks to its contextual memory and the Hibernate-infused RAG pipeline, provides informed replies.

Empowering any user, **regardless of their familiarity with Hibernate or relational databases**, this chatbot enables them to effortlessly retrieve necessary information and gain valuable insights. By integrating the language processing power of LLMs

with Hibernate's deep understanding of your business domain model, simply talking to your data is now possible for everyone.

MCP Server

Looking forward, a more advanced application of this technology is the **Model Context Protocol (MCP)**. MCP is an open standard that lets AI models connect with tools and data providers, acting like a universal adapter to give AI information and actions beyond its initial training.

By exposing your Hibernate Assistant as an MCP tool, through the Quarkus MCP Server extension, you can allow any compatible AI agent to talk with your database. The agent discovers the "tools" provided by Hibernate Assistant such as "hibernateGetMetamodel" or "hibernateExecuteQuery" and can invoke them autonomously to answer user questions:

```
import io.quarkiverse.mcp.server.Tool;
import io.quarkiverse.mcp.server.ToolArg;

@Tool(name = „hibernateGetMetamodel“, description = „Retrieve
a textual (JSON) representation of the Hibernate Metamodel,
i.e. the entities, „ +
    „properties and relationships defined in the persistence
layer, that can be used to access the database.“)
public String getMetamodel() {
    return MetamodelJsonSerializerImpl.INSTANCE.
toString(session.getFactory().getMetamodel());
}

@Tool(name = „hibernateExecuteQuery“, description = „Execute a
query against the database using the Hibernate Assistant. „ +
    „Pass a natural language message and get a JSON
representation of the results extracted from the database.“)
public String executeQuery(@ToolArg(description = „Natural
language query to execute“) String query) {
    return assistant.executeAiQuery(query, session);
}
```

This turns your Java application into a highly intelligent data node that can be plugged into the broader ecosystem of **AI agents**, allowing them to perform relational data access and manipulation powered by the logic you have already defined in your Hibernate domain model.

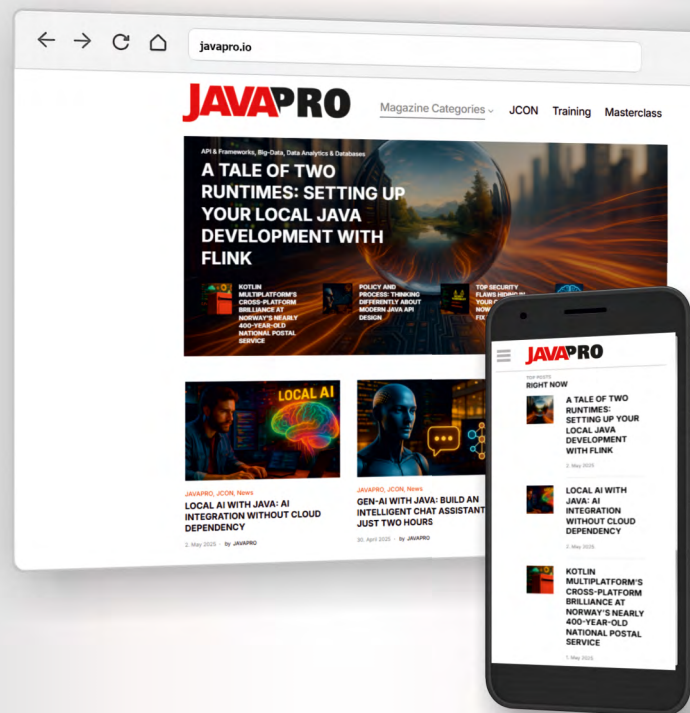
Conclusion

The teamwork of Hibernate ORM, Quarkus, and LangChain4j marks a significant step forward in making relational data accessible to AI. By abstracting the complexity of SQL generation and relational data extraction, and leveraging the rich metadata of the domain model, Hibernate Assistant allows developers to build applications where users can truly **talk to their data**. Whether through a custom chatbot or a standardized MCP interface, the barrier between natural language and structured data is rapidly disappearing.

-> Back to the Table of Content

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

#JAVAPRO #GENAI

Build Vector Database Apps with Pure Java



The Java ecosystem is entering a new phase. Generative AI is no longer experimental; it is becoming a foundational capability in enterprise systems. Yet while frameworks for inference and orchestration have matured, one critical bottleneck continues to slow down adoption: data access for semantic retrieval.

At the heart of modern GenAI systems lies Retrieval-Augmented Generation (RAG). This paradigm depends on fast access to embeddings – high-dimensional vectors that represent meaning. For Java developers, integrating this capability has historically meant introducing an external vector database into the architecture. What appears to be a straightforward design decision quickly becomes a source of complexity, inefficiency, and architectural friction.

EclipseStore 4 introduces a fundamentally different approach. By integrating the JVector library directly into a Java-native object graph persistence engine, it eliminates the boundary between application memory and vector search. The result is not just

another database integration, but a shift in how GenAI systems can be built in Java.

This article explores that shift, from the challenges of external vector systems to the mechanics of embedded vector search, and ultimately to the architectural implications of bringing vector intelligence directly into the JVM.

Data Wall in Java-Based GenAI Architectures

Java developers building GenAI applications today typically rely on external vector databases such as Pinecone, Milvus, or Weaviate. These systems introduce a structural mismatch with Java's object-oriented programming model. The fundamental issue lies in translation. Domain objects in Java must be flattened into vectors or serialized formats before being sent across the network. The vector database operates independently, returning identifiers or payloads that must then be rehydrated into Java objects. This constant transformation introduces what can be described as a serialization tax.

The cost of this tax is not just computational. It manifests in multiple dimensions. Latency increases due to network round-trips. System complexity grows as additional infrastructure must be provisioned and maintained. Consistency becomes harder to guarantee because the vector store and the primary data store evolve independently. More importantly, this architecture breaks one of Java's core strengths: working with rich, interconnected object graphs. Instead of navigating relationships directly in memory, developers must orchestrate distributed queries across heterogeneous systems. This is the "Data Wall" – a boundary between application logic and semantic data that prevents truly efficient GenAI implementations in Java.

EclipseStore: Native Object Graph Persistence

EclipseStore approaches persistence from a fundamentally different perspective. Rather than mapping objects to tables or JSON documents, it stores entire Java object graphs directly in binary form. The persistence layer is not an external system but an extension of the application itself.

When an application starts, the object graph can be loaded into memory, either fully or partially. From the developer's perspective, working with persistent data feels identical to working with in-memory objects. There is no impedance mismatch, no ORM, and no need for query languages.

Author:



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

<https://www.linkedin.com/in/richardfichtner/>

This design has profound implications. It allows applications to maintain complex relationships without translation. It enables direct navigation through object graphs. And it shifts the performance model from network-bound operations to memory-bound operations. With EclipseStore 4, this model is now extended into the domain of vector search.

JVector: Embedded Vector Search for Java

JVector is a high-performance vector search library implemented entirely in Java. It provides efficient similarity search capabilities for high-dimensional embeddings using modern approximate nearest neighbor algorithms such as HNSW (Hierarchical Navigable Small World). Unlike external vector databases, JVector is designed to run inside the same JVM as the application. This eliminates network overhead and allows direct interaction with application data structures.

The integration of JVector into EclipseStore creates something new: an embedded, Java-native vector search engine. Vector indices become part of the object graph. Similarity queries operate directly on in-memory data. The distinction between “database” and “application” begins to dissolve.

Vector Search in Practice

At its core, vector search is about measuring similarity. Unstructured data—text, images, or signals—is transformed into numerical vectors using embedding models. Each vector represents a point in a high-dimensional space. The distance between two vectors indicates how similar they are. Common distance metrics include cosine similarity and Euclidean distance. A query vector is compared against a collection of stored vectors, and the closest matches are returned. In traditional systems, this process requires specialized infrastructure. In EclipseStore 4, it becomes a native operation.

```
public class Document {
    private String id;
    private String content;
    private float[] embedding;
}
```

Once stored in EclipseStore, these objects can be indexed using JVector. A similarity search becomes a simple in-process operation:

```
List<Document> results = store.search(
    VectorQuery.builder()
        .index(„documents“)
        .vector(queryEmbedding)
        .limit(5)
        .build()
);
```

There is no network call, no serialization step, and no transformation layer. The query operates directly on the in-memory representation of the data.

From Persistence Engine to Embedded Vector Database

The integration of JVector effectively transforms EclipseStore into an embedded vector database. However, it is important to understand that this transformation does not follow the traditional database model. EclipseStore does not introduce a query language or a separate runtime. Instead, it extends the capabilities of the native Java object graph. Vector indices are attached to objects, and search operations are exposed as APIs. This approach preserves the strengths of Java development. Developers continue to work with familiar constructs – classes, collections, and references – while gaining access to advanced semantic search capabilities.

Indexing Concepts in EclipseStore

Indexing in EclipseStore is explicit and developer-controlled. Unlike traditional databases, where indexing is often abstracted away, EclipseStore encourages developers to align index structures with application requirements.

Vector indices are used for semantic similarity. They are built on top of JVector and optimized for high-dimensional search. In addition to vector indices, EclipseStore supports other index types tailored to specific use cases. Standard object indices provide efficient lookups for exact matches or range queries. Specialized indices, such as geospatial indices, enable efficient spatial queries. The key principle is alignment. Each index type is designed for a specific class of problems. Using the correct index ensures optimal performance, while misuse can lead to inefficiencies.

GigaMap: Scaling Beyond Memory

One of the most significant challenges in GenAI systems is scale. While vector indices benefit from being in memory, the underlying data can grow far beyond the capacity of RAM. EclipseStore addresses this challenge with the GigaMap concept. GigaMap introduces a separation between hot and cold data. Hot data, such as vector indices, remains in memory for fast access. Cold data, including large object graphs, is persisted to storage.

When a query identifies relevant vectors, the associated objects are loaded into memory on demand. This lazy loading mechanism ensures that only the necessary data is brought into RAM, minimizing memory usage while preserving performance. The result is a system that can handle massive datasets while maintaining the responsiveness of in-memory operations.

Persistence Mechanics and Performance Considerations

Persistence in EclipseStore is based on writing binary representations of object graphs to storage. This process is highly efficient, but its performance depends on how it is used. One of the most important principles is batching. Writing data in large blocks significantly improves throughput. Small, frequent writes introduce overhead and reduce performance.

The underlying reason lies in the nature of I/O operations. Writing to disk is orders of magnitude slower than CPU operations. Maximizing throughput requires minimizing the number of I/O roundtrips and maximizing the amount of data written per operation. Empirical measurements show that write performance increases dramatically with larger batch sizes, approaching hardware limits when data is written in sufficiently large blocks. This has direct implications for application design. Developers should structure their persistence logic to collect and store data in bulk. Avoiding redundant writes and minimizing fragmentation are critical for achieving optimal performance.

Keeping Hot Indices in Memory

A central advantage of EclipseStore 4 is the ability to keep vector indices in memory while persisting the rest of the data. This hybrid approach combines the speed of in-memory computation with the scalability of persistent storage. Vector search operations are executed entirely in RAM, ensuring minimal latency. Once relevant results are identified, the associated data can be loaded from storage. This design aligns with the access patterns of GenAI applications. Most queries require fast similarity search followed by retrieval of a relatively small subset of data. By optimizing for this pattern, EclipseStore achieves both performance and scalability.

Open Source and Community

EclipseStore is an Eclipse open-source project published under the Eclipse Public License (EPL) 2.0, ensuring that it remains open and accessible to the Java community. The license allows using the code in commercial software. Developers can explore the source code, contribute to the project, and build custom extensions. The project's documentation provides comprehensive guidance on usage and architecture. The source code is available on GitHub, enabling transparency and collaboration. The project is gaining great momentum and attracting more and more Java developers.

Rethinking GenAI Architectures in Java

EclipseStore 4 challenges the prevailing assumption that vector search must be externalized. By embedding vector capabilities directly into the JVM, it simplifies architecture, reduces latency, and restores the integrity of the object model. This shift has broader implications. It enables new classes of applications that were previously impractical due to latency or complexity constraints. It allows developers to build systems where data and logic coexist seamlessly, without artificial boundaries.

Practical Use Cases

EclipseStore 4 opens the door to a wide range of applications. Local-first AI systems can perform semantic search over private data without relying on external services. Enterprise knowledge systems can integrate structured and unstructured data into a single object graph. Real-time decision systems can combine vector similarity with traditional business logic in a single process. Applications in finance, healthcare, manufacturing, and edge computing can all benefit from this approach. Any scenario that requires fast, context-aware retrieval can leverage embedded vector search.

Getting Started

Developers interested in exploring EclipseStore 4 can access the project through its official documentation and GitHub repository. The documentation provides detailed guides, examples, and best practices for building applications with embedded vector search. Code examples and sample projects demonstrate how to integrate vector indexing, perform similarity queries, and optimize persistence strategies. The combination of EclipseStore and JVector represents a new paradigm for Java-based GenAI development. By bringing vector intelligence into the JVM, it removes long-standing barriers and enables a more natural, efficient way to build intelligent applications.

Distributed Vector Database Apps with Eclipse Data Grid

EclipseStore is built for single-node apps. With Eclipse Data Grid, you can run your EclipseStore app as a distributed application very easily. It begins where EclipseStore ends. Eclipse Data Grid is not a server, but a software-as-code project. It provides the code that creates the entire cluster infrastructure you need to run EclipseStore across multiple cluster nodes. The cluster provides you with replication, high availability, elastic scaling, scale-to-zero, and backup features. It takes over the entire cluster control, including when new nodes are started up, or nodes are shut down, and what happens when nodes crash. Eclipse Data Grid is also an Eclipse open-source project under the EPL 2.0 license.

Links:

EclipseStore project: www.eclipsestore.io

EclipseStore code on GitHub: <https://github.com/eclipse-store/store>

EclipseStore Documentation: <https://docs.eclipsestore.io/manual/storage/getting-started.html>

Eclipse Data Grid code on GitHub: <https://github.com/eclipse-datagrid/>

Eclipse Public License 2.0: <https://www.eclipse.org/legal/epl-2.0/>

JVector on GitHub: <https://github.com/datastax/jvector>

-> Back to the Table of Content

High-Performance Vector-Search Grids with Java

The rapid rise of Generative AI has fundamentally changed how modern systems are designed. While much of the attention has focused on large language models and inference pipelines, the real differentiator in production systems lies elsewhere: in the infrastructure that delivers context to those models. Retrieval-Augmented Generation (RAG) has become the de facto pattern for enterprise GenAI. It relies on fast, scalable access to embeddings, metadata, and relationships. Yet for Java developers, building such systems at scale has remained a challenge. External vector databases introduce latency, fragmentation, and operational complexity. Distributed caches struggle with memory inefficiency and serialization overhead. Traditional databases cannot handle high-dimensional similarity search at the required speed.

Eclipse Data Grid emerges as a fundamentally different approach. It is not just a distributed cache, nor a traditional database cluster. It is a Java-native, high-performance in-memory vector search grid designed to run Java applications with embedded JVector capabilities and EclipseStore persistence across multiple JVMs. This article explores how Eclipse Data Grid enables a new class of distributed GenAI systems in Java, combining the JVM power, vector search, object graph persistence, and distributed state synchronization into a unified architecture.

The Limits of Traditional Distributed Architectures for GenAI

Modern GenAI systems are inherently distributed. They must scale across regions, handle large datasets, and serve low-latency queries globally. Traditional approaches attempt to solve this by combining multiple specialized systems. A typical architecture might include a relational database for transactional data, a vector database for embeddings, a cache for fast access, and a search engine for indexing. Each component operates independently, requiring data synchronization pipelines and transformation layers. This leads to a fragmented system where the application no longer owns its data model. Instead, developers spend significant effort maintaining consistency across systems, handling serialization, and orchestrating distributed queries.

The problem becomes even more pronounced when scaling. Vector databases often rely on sharding strategies that increase memory requirements. Distributed caches replicate data redundantly, requiring two to three times the RAM of the dataset. Network latency becomes a dominant factor, especially in multi-region deployments. These challenges create a bottleneck

for Java-based GenAI systems, limiting both performance and scalability.

Eclipse Data Grid: Software-as-Code Infrastructure

Eclipse Data Grid introduces a different paradigm. It is a Software-as-Code project that defines the complete infrastructure for running distributed Java applications. Instead of treating infrastructure as an external concern, it becomes part of the application itself. At its core, Eclipse Data Grid orchestrates multiple JVM instances running the same Java application. Each instance operates on a synchronized object graph, enabling a distributed yet coherent system state. The grid is specifically designed to run EclipseStore 4 applications. This means that persistence, indexing, and vector search are all handled within the same architectural model. Rather than distributing queries across independent systems, Eclipse Data Grid distributes the application state itself.

Distributed Java Object Graphs

One of the most distinctive features of Eclipse Data Grid is its ability to replicate entire Java object graphs across multiple JVMs. In traditional systems, data is partitioned or sharded. Each node holds only a subset of the data, requiring coordination for queries that span multiple partitions. This requires an enormous amount of expensive RAM. In contrast, Eclipse Data Grid ensures that every node maintains a complete copy of the object graph that is persisted in a file storage on a fast SSD, while only hot data is stored in RAM. This approach has several advantages. Queries can be executed locally on each node without network communication. Application logic can operate on fully connected data structures. Consistency is maintained at the object level rather than through distributed transactions. SSD access is

Author:

Gerald Kammerer has been working with Java for over twenty years and develops enterprise applications for industrial companies as a freelancer and technical advisor. For nearly fifteen years, he has been deeply involved in various open-source projects like Eclipse Serializer. Gerald is also known as an independent author writing for various developer magazines



generally faster than accessing shards distributed across different nodes. Cold data can be stored in even cheaper BLOB storage like S3. The result is a system where the cluster behaves like a single, distributed JVM.

Single Writer Architecture

A key component of Eclipse Data Grid is its single-writer approach. In this model, one node is responsible for all write operations. This node acts as the authoritative source of truth for any changes to the object graph. When a write occurs, the writer node persists the changes using EclipseStore and publishes them to an event stream, typically backed by Apache Kafka. Reader nodes subscribe to this stream and apply updates to their local object graphs. If the writer crashes, one of the reader nodes immediately becomes the new writer node, and the cluster starts with a new reader node. Reader nodes can be fully recovered from Kafka's data. Since the writer's sole task is to execute and forward write operations, and it can be provisioned to any size, the writer can handle an enormous data throughput.

This architecture simplifies consistency management. The writer node guarantees ACID-compliant strong consistency for write operations. Reader nodes achieve eventual consistency by consuming the write events. The separation of concerns between writing and reading allows the system to scale efficiently while maintaining data integrity.

Event-Driven Replication with Kafka

Replication in Eclipse Data Grid is handled through an event-driven model. Each change to the object graph is serialized and sent over the writer to Kafka, which acts as a durable, ordered log of events. Reader nodes consume these events, merge the

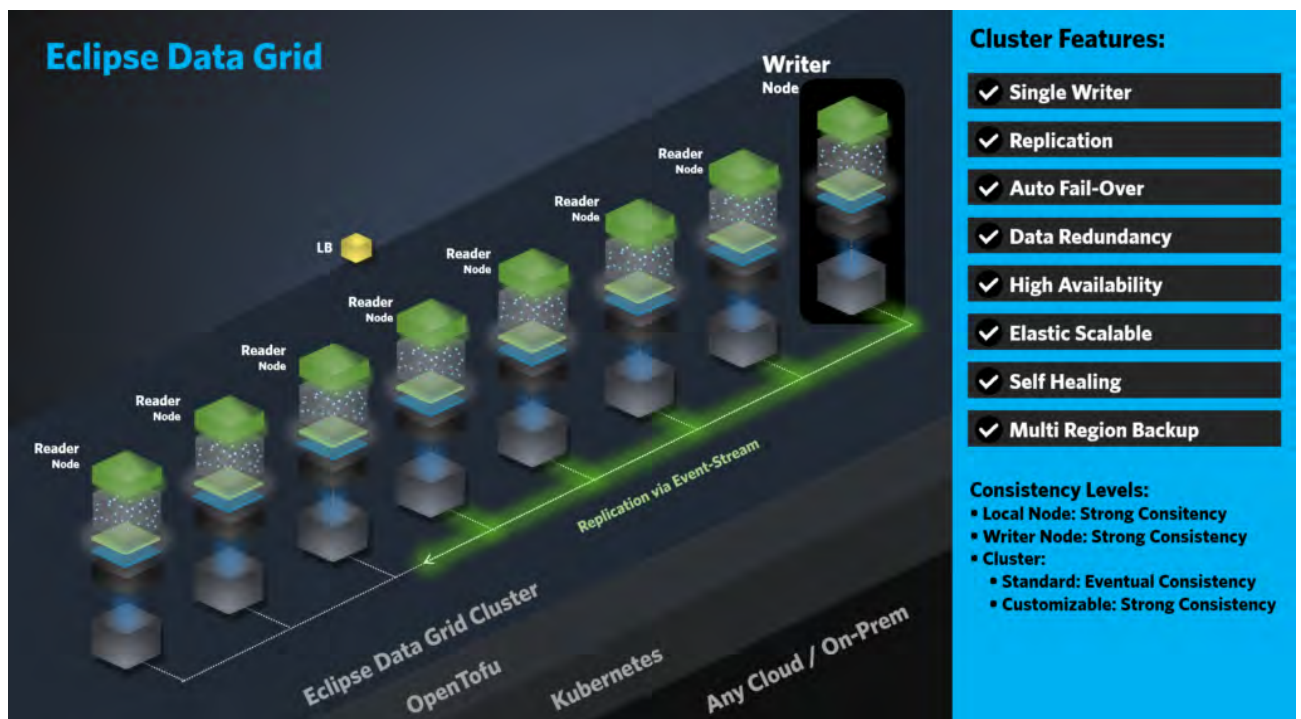
updates into their local state, and persist them in their local storage. This process ensures that all nodes converge to the same state over time. Because Kafka provides high throughput and fault tolerance, the system can handle large volumes of updates without becoming a bottleneck. It also enables features such as replay, recovery, and multi-region replication. The use of event streaming aligns naturally with the needs of GenAI systems, where data is continuously evolving and must be propagated quickly.

Embedded Vector Search at Cluster Scale

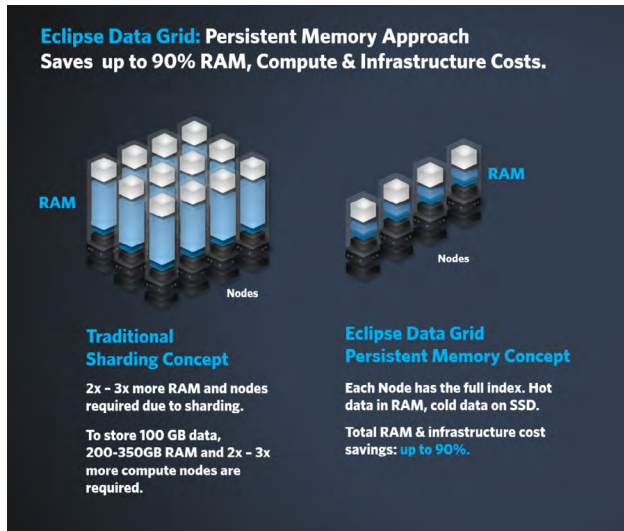
With the integration of JVector in EclipseStore 4, each node in the grid becomes a fully capable vector search engine. Vector indices are stored in memory alongside the object graph, enabling ultra-low latency similarity queries. Because every node has a complete copy of the data, vector searches can be executed locally without network hops. This eliminates one of the primary sources of latency in traditional distributed vector databases. The grid effectively becomes a distributed in-memory vector search system. Queries can be load-balanced across nodes, ensuring high throughput and scalability.

Persistent Cache and Memory Efficiency

One of the most significant innovations in Eclipse Data Grid is its persistent cache model. Instead of keeping all data in RAM, the system uses a hybrid approach where only hot data is stored in memory. Cold data is persisted to storage, such as SSDs or object storage systems like S3. When needed, this data can be loaded into memory using lazy loading mechanisms. This approach dramatically reduces memory requirements. In many cases, only 25% to 30% of the dataset needs to be kept in RAM, while the rest remains on disk.



Traditional distributed caches often require multiple copies of data in memory, leading to excessive RAM usage. Eclipse Data Grid avoids this by combining persistence with intelligent loading strategies. Interestingly, accessing data from local SSD storage can be faster than performing network calls to other nodes. This challenges the conventional assumption that everything must be kept in memory for performance.



GigaMap and Lazy Loading

The GigaMap concept plays a central role in managing large datasets. It allows the system to handle object graphs that exceed available memory by loading only the required subgraphs. When a vector search identifies relevant objects, their associated data is loaded from storage into memory. This process is highly optimized and typically occurs within milliseconds. By combining in-memory indices with on-demand data loading, Eclipse Data Grid achieves a balance between performance and scalability. It enables applications to work with datasets far larger than available RAM without sacrificing responsiveness.

Performance Characteristics and Storage Best Practices

Performance in Eclipse Data Grid is closely tied to the underlying persistence mechanisms of EclipseStore. Efficient use of storage is critical for achieving high throughput. One of the key principles is batching writes. Writing data in large blocks maximizes throughput and minimizes I/O overhead. Small, frequent writes can significantly degrade performance due to the latency of disk operations. Empirical measurements show that throughput increases dramatically with larger write sizes, approaching hardware limits when data is written in sufficiently large batches. This has direct implications for distributed systems. The writer node should aggregate changes and commit them in bulk whenever possible. This ensures that replication events are efficient and that the system can handle high update rates.

Running Eclipse Data Grid in Kubernetes

Eclipse Data Grid is designed to run in modern cloud-native environments. Kubernetes provides the orchestration layer

for deploying and managing nodes. Each node runs as a containerized JVM instance. Persistent volumes are used for storage, and Kafka is deployed as a supporting service for event streaming. The system supports features such as auto failover, self-healing, and elastic scaling. Nodes can be added or removed dynamically, and the system will rebalance workloads accordingly. Because each node maintains a full copy of the data, scaling primarily improves read throughput and availability rather than partitioning data.

A Unified Programming Model

One of the most compelling aspects of Eclipse Data Grid is its programming model. Developers work with standard Java constructs, without needing to learn new query languages or data models. The same code that operates on a single node can run unchanged in a distributed environment. The grid handles replication, synchronization, and persistence transparently. This reduces cognitive overhead and allows developers to focus on business logic rather than infrastructure concerns.

Language-Agnostic Access

While the pure-Java approach allows developers to bundle business logic directly within a distributed Java application, EclipseStore 4 and the Eclipse Data Grid support a decoupled, language-agnostic architecture. In this model, the high-performance vector search and data manipulation logic remain as Java-native components within the grid, while the primary business logic resides in external applications. This flexibility enables a wide range of environments – including Python or .NET applications, microservices, and even Large Language Models (LLMs) via orchestration frameworks – to interface with the data grid as a centralized intelligence hub. Developers can access this distributed memory space from outside the grid using standard protocols such as REST or GraphQL, ensuring seamless integration regardless of the tech stack. However, it is critical to note that while the embedded Java-native model offers the most extreme performance by eliminating the serialization tax, accessing the cluster via external APIs introduces network latency and overhead. By offloading complex vector indexing to the Java-native engine, external services can still leverage high-speed similarity searches even if they are not built in Java. Consequently, the Eclipse Data Grid functions as a high-performance, language-independent backbone that bridges the gap between Java's in-memory processing and the polyglot requirements of the GenAI ecosystem.

Higher Vector-Search Performance with OpenJ9

JVM performance plays a critical role in in-memory vector search systems, where latency, memory efficiency, and throughput directly impact retrieval quality and system scalability. Eclipse Data Grid benefits significantly from running on Eclipse OpenJ9, a JVM specifically optimized for high-density, memory-efficient workloads. OpenJ9 is designed with a much smaller memory footprint compared to traditional JVMs, allowing more data – especially vector indices and object graphs – to remain in

RAM without increasing infrastructure costs. Its advanced garbage collection strategies reduce pause times and improve predictability, which is essential for real-time similarity search and consistent query latency. Additionally, OpenJ9's fast startup and efficient JIT compilation enable rapid scaling in containerized environments such as Kubernetes. These characteristics make OpenJ9 an ideal runtime for Eclipse Data Grid, where maximizing usable memory and minimizing latency are key to achieving high-performance distributed vector search.

Rethinking Distributed GenAI Systems

Eclipse Data Grid represents a shift in how distributed systems can be built for GenAI. Instead of assembling multiple specialized components, it provides a unified platform where data, logic, and infrastructure are tightly integrated. This approach simplifies architecture, reduces latency, and improves efficiency. It enables new patterns, such as distributed object graphs and stateful AI systems, that were previously difficult to implement. By combining EclipseStore, JVector, and Eclipse Data Grid, developers gain a complete stack for building high-performance GenAI applications in Java.

Use Cases and Practical Applications

Eclipse Data Grid is particularly well-suited for applications that require fast, distributed access to large volumes of contextual data. Global RAG systems can benefit from having vector indices available locally on every node, enabling near-zero latency retrieval across regions. Multi-agent systems can share a consistent state across nodes, allowing agents to collaborate effectively. Real-time fraud detection systems can analyze behavioral patterns using vector similarity, while maintaining a synchronized view of transactions. Enterprise knowledge platforms can integrate structured and unstructured data into a single, queryable object graph. The system is also well-suited for building semantic caches for LLM gateways, reducing API costs and improving response times by reusing previous results.

Getting the Code and Examples

Eclipse Data Grid is an open-source project built on the Eclipse Foundation ecosystem. Developers can access the source code, deployment configurations, and examples through its GitHub repository. The project integrates seamlessly with EclipseStore 4 and JVector, forming a cohesive platform for building GenAI systems. Documentation, Kubernetes deployment files, and sample applications are available to help developers get started quickly. The open nature of the project ensures transparency and encourages community contributions.

What Comes Next?

The evolution of EclipseStore and Eclipse Data Grid does not stop at high-performance vector search; the roadmap points toward a deeper fusion of data structures through an upcoming native graph API. This expansion will allow developers to build and query complex knowledge graphs directly on top of the existing Java object graph storage engine. By leveraging the engine's

ability to handle intricate relationships and massive scales via the GigaMap architecture, Eclipse Data Grid is positioning itself as the primary foundation for GraphRAG. Unlike traditional RAG, which relies solely on vector similarity, GraphRAG integrates structured relational context, enabling AI models to understand the "why" and "how" behind data points by traversing explicit connections within the JVM memory space. This transition from simple similarity to deep relational intelligence ensures that Java remains the premier language for the next generation of context-aware, enterprise-grade GenAI applications.

Conclusion

Eclipse Data Grid redefines what a distributed data system can be for Java-based GenAI applications. By embedding vector search, persistence, and state synchronization directly into the JVM, it eliminates the need for external systems and complex integration layers. The result is a high-performance, memory-efficient, and developer-friendly platform that aligns with the principles of modern Java development. As GenAI continues to evolve, architectures that minimize latency, maximize efficiency, and simplify development will become increasingly important. Eclipse Data Grid positions itself at the forefront of this evolution, enabling developers to build the next generation of intelligent systems directly in Java.

Links:

Eclipse Data Grid code on GitHub: <https://github.com/eclipse-datagrid/datagrid>

EclipseStore project: www.eclipsestore.io

EclipseStore code on GitHub: <https://github.com/eclipse-store/store>

EclipseStore Documentation: <https://docs.eclipsestore.io/manual/storage/getting-started.html>

Eclipse Public License 2.0: <https://www.eclipse.org/legal/epl-2.0>

JVector on GitHub: <https://github.com/datastax/jvector>

Eclipse OpenJ9 project: <https://projects.eclipse.org/projects/technology.openj9>

Eclipse OpenJ9 on GitHub: <https://github.com/eclipse-openj9/openj9>

-> Back to the Table of Content



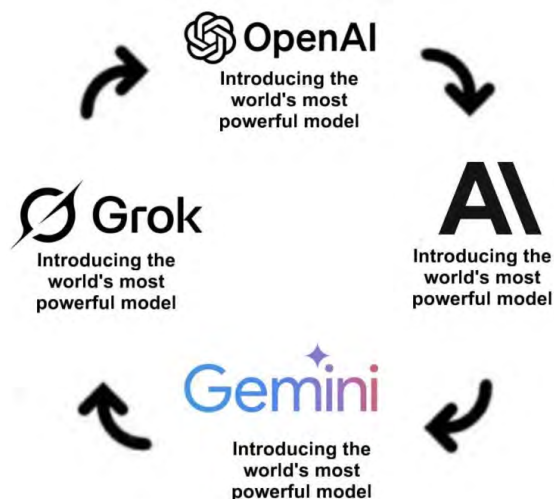
#JAVAPRO #GENAI

LLMs for JVM Code in 2026: Which Models to Trust, Which Benchmarks to Believe

A deep-dive into how LLM coding benchmarks work, why Python ate the evaluation world, and what the data actually says about model performance on Java, Kotlin, and Scala in early 2026.

If you're reading this, you're probably already using some LLM for coding. Maybe it's Copilot, maybe Claude Code, maybe Cursor with Gemini enabled. You know the drill. And you've probably seen every vendor's launch post proudly declaring their model "best for coding" – because, let's be honest, do you truly expect the announcement "We are worse than competitors?"

The problem is that when someone asks, "Which model is best for Java?", most answers boil down to a vibes check, re-published benchmarks that hardly anyone truly understands, or – my personal favorite – a single afternoon of tinkering extrapolated



Author:

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



<https://www.linkedin.com/in/arturskowronski/>

into a universal recommendation. Today, we're fixing that. We'll walk through how LLM benchmarking for code actually works, which models are available on the market in early 2026, and – most importantly – how they perform in benchmarks that are specific to the JVM world, including Java, Kotlin, and Scala.

How Do You Even Measure an LLM's Ability to Write Code?

The history of LLM coding benchmarks is a story of steadily rising expectations. It started with a very basic question: can the model generate working code at all?

HumanEval, described in the paper *Evaluating Large Language Models Trained on Code*, is the foundation of almost everything that came afterward. 164 hand-written Python problems, each with unit tests. The model gets a function signature and a docstring, and must produce the function body. The key innovation was the `pass@k` metric – instead of measuring whether the code looks good, you check whether it *works*. A fundamental shift. By the end of 2024, though, top models were hitting 90%+ on HumanEval, making it essentially useless for distinguishing between frontier models. A similar alternative, **MBPP** (Mostly Basic Python Problems), has 974 entry-level problems – more data, same ceiling problem.

The real revolution arrived with **SWE-bench** (Software Engineering Benchmark). Instead of isolated puzzles, SWE-bench puts a model in front of a real GitHub repository with a real issue description and asks it to produce a patch. At this point, we're no longer testing whether a model can "code" in the narrow sense – we're testing whether it can behave like a software engineer: understand an existing codebase, navigate across files, reason about component interactions, follow project conventions, and produce a change that doesn't break existing tests.

SWE-bench Verified – a curated subset of 500 tasks validated by humans – became the industry gold standard. But it has serious limitations. It's exclusively Python. It's old enough that training contamination is a real concern. And the solutions are often tiny: a mean of 11 lines of code, with over 77% touching only one function. Amazon's analysis found that over 40% of problems come from Django alone. This tells us SWE-bench is testing surgical precision on isolated Python problems, not the kind of sprawling, multi-component changes that dominate real enterprise work.

SWE-bench Pro from Scale AI tried to address this: close to 2000 tasks across 41 repositories in Python, Go, JavaScript, and TypeScript, with solutions around 100 lines spanning few files. Here's the reality check – while top models score over 80% on SWE-bench Verified, they drop to around 23% on SWE-bench Pro with the standard SWE-Agent scaffold, and the latest scaffolds only push that to about 45-56%. As of February 2026, Claude Opus 4.5 leads at 45.89%, followed by Sonnet 4.5 at 43.60% and Gemini 3 Pro at 43.30%. That's a massive gap from the 80%+ numbers you see in marketing materials – a healthy reminder that we are still very far from "AI replaces engineers" territory (at least for now, heh).

BigCodeBench raised the bar with 1,140 tasks requiring interaction with 139 different libraries. **LiveCodeBench** tackled data contamination by sourcing tasks from weekly programming contests, each tied to a publication date – so you can check if a model could have memorized the answer. Both useful, both Python-dominant.

Multilinguality: Where Is Java?

And this is where we get to the heart of the problem. A review of 24 major coding benchmarks reveals a rather uncomfortable statistic: an overwhelming 95.8% focus on Python, while only five include Java at all.

This isn't accidental. Python dominates ML research, so benchmarks are naturally built around the language researchers themselves use. The result is a benchmarking ecosystem that tells us a lot about Python performance, but surprisingly little about real capabilities in Java – or the broader JVM world.

MultiPL-E translates HumanEval and MBPP into 18 additional languages, including Java, Kotlin, and Scala. Sounds great on paper. In practice, automatic translation doesn't capture language idioms – a Java test mechanically translated from Python may compile, but it's testing Pythonic assumptions, not real OOP design patterns. **HumanEval-XL** covers 12 languages and adds 80 problems in 23 natural languages – better, but the problems remain small and isolated.

Aider Polyglot deserves special attention: 225 hard-level Exercism problems across six languages, with Java getting 47 of them. Solutions range from 30 to 200 lines with one round of feedback. Not enterprise-scale, but one of the few standardized ways to compare model performance on JVM code.

JavaBench was a direct response to Python's dominance – four real Java projects, 389 methods across 106 classes, explicitly testing OOP features like encapsulation, inheritance, and polymorphism. A nice try, but not the most actively maintained benchmark out there. **CoderUJB** pushed further: 17 real open-source Java projects, 2,239 questions covering code generation, test generation, bug fixing, and defect detection.

The Brokk Power Ranking: The Java Benchmark That Changed the Conversation

But the benchmark that really changed the conversation for JVM developers is the **Brokk Power Ranking**, created by Jonathan Ellis, co-creator of Apache Cassandra. Unlike everything else discussed so far, it is not Python-only. It draws tasks from real Java repositories: Brokk, JGit, LangChain4j, Apache Cassandra, and Apache Lucene. The tasks come from commits made within the last six months, dramatically reducing contamination risk. And at 93 tasks with contexts reaching up to 108k tokens, it sits deliberately between AiderBench's toy problems and SWE-bench's "here's the repo, good luck" approach.

The August 2025 results immediately highlighted patterns that Python benchmarks had been hiding. GPT-5 and Claude Opus

4.5 occupied the S tier, with GPT-5 dominating across every price point. Chinese models like DeepSeek-V3, Kimi K2, and Qwen3 Coder performed noticeably worse than they did on SWE-bench and AiderBench – a finding that raised uncomfortable questions about whether some of these models had been trained on the tests they were supposedly being evaluated on.

The November 2025 update reshuffled things further. Claude 4.5 Haiku stood alone in S Tier – a subjective but defensible call by Ellis, who argued that its speed was so far ahead of everything else while being “smart enough” to solve most problems correctly. GPT 5.1 and GPT 5-Mini took A Tier. Sonnet 4.5 and Grok 4.1 Fast made B Tier. Gemini 3 Pro Preview landed in C Tier – relatively expensive and slow for its level of intelligence – with Ellis noting they “expected the full release to do better after resolving the preview’s teething pains.”

And as of February 2026, the Brokk team has just published Gemini 3 results (with “surprising outcome” in their teaser), and the live leaderboard is worth checking at brokk.ai/power-ranking. The six-month refresh cycle they promised is essentially the cadence to watch for JVM developers.

The February 2026 State of the Art: What Actually Works

Let me just step back from individual benchmarks and synthesize what we know across all available data as of this week. I’ll be honest: I have my own opinions here, and yes, I work at VirtusLab where we deal with JVM codebases daily, so my experience is skewed toward Java and Kotlin. Take this with appropriate salt.

The Top Tier: Claude Opus 4.5 and GPT-5.2

On SWE-bench Verified (Python-only, remember), the latest official February 2026 “Bash Only” run – importantly, **not** self-reported by the labs – puts Claude Opus 4.5 at the top, followed by Gemini 3 Flash, then MiniMax M2.5. As Simon Willison noted, it’s interesting that Claude Opus 4.5 beat Opus 4.6 by about a percentage point. Three Chinese models – GLM-5, Kimi K2.5, and DeepSeek V3.2 – also made the top ten.

On SWE-bench Pro (the harder, multi-language benchmark), Claude Opus 4.5 leads at 45.89%, followed by Claude Sonnet 4.5 at 43.60% and Gemini 3 Pro at 43.30%. GPT-5 with high reasoning is at 41.78%, and GPT-5.2-codex at 41.04%.

On the Brokk Power Ranking (Java-specific, real codebases), the historical data shows GPT-5 family and Claude Opus consistently at the top. For Java work specifically, these two model families remain the safest bet.

The Practical Middle: Sonnet 4.5, Gemini 3 Pro, and the Speed Question

Here’s where things get nuanced. Claude Sonnet 4.5 is screaming fast and increasingly competitive on raw quality. If your workflow involves lots of iterative back-and-forth – which, let’s be honest, most coding agent workflows do – Sonnet’s speed advantage

compounds across dozens of edit-test cycles. The Brokk data shows this clearly: models that can fail fast and retry efficiently often achieve better outcomes than slower models that are marginally smarter per-request.

Gemini 3 Pro (and the just-released 3.1 Pro) is a strong contender, especially on generic coding tasks. But the Brokk Java data consistently showed it underperforming relative to its SWE-bench scores. The November ranking had it in C tier for Java work specifically. The question is whether the GA release and 3.1 update have closed that gap – we’ll know more when the next Brokk update drops.

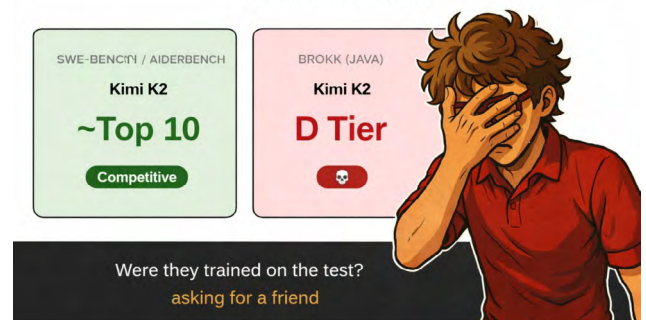
The Budget Tier: DeepSeek V3.2, GPT-5 Mini, and the Open-Weight Question

DeepSeek V3.2 delivers remarkable cost-efficiency – 74.2% on Aider Polyglot at \$1.30 per run, which is 22x cheaper than GPT-5. MiniMax M2.5 surged to 80.2% on SWE-bench Verified as an open-weight model. GLM-4.7 is the strongest open-source model on SWE-rebench.

But here’s the catch, highlighted by the Brokk team’s analysis of what they called the “Reliability Gap”: open-weight models generally lag about six months behind the frontier in agentic robustness. They can generate code, but struggle in the self-correction phase – needing significantly more iterations and human babysitting to fix bugs compared to frontier models.

And I have one question...

The Brokk Power Ranking asked Chinese models to code Java instead Python:



Beyond Pass/Fail: Code Quality on Java (The Sonar Data)

Passing tests is one thing. Writing *good* code is another. This is where the **Sonar LLM Leaderboard** provides data that no other benchmark captures, and it’s particularly relevant because it’s based on over 4,400 distinct **Java** programming assignments analyzed with SonarQube by Jonathan Vila.

Leaderboard

Click column headers to sort by different metrics

Want to add a model or license data? [Contact us](#)

Q Search models by name...

All organizations ▾

All pass rates ▾

All complexity ▾

All issue density ▾

Showing 43 of 43 models

#	Org	Model ↕	Release date ↕	Pass rate ↓	Issue density ↕	Lines of code ↕	Cyclomatic complexity ↕	Cognitive complexity ↕
1		Gemini 3.1 Pro High	Feb 19, 2026	84.17%	19.71	307,703	233.9	158.4
2		Opus 4.5 Thinking	Nov 24, 2025	83.62%	15.15	639,465	243.3	121.0
3		Opus 4.6 Thinking	Feb 5, 2026	82.38%	18.33	563,574	239.0	133.2
4		Claude Sonnet 4.6 Thinking	Feb 17, 2026	81.76%	20.90	627,187	230.6	124.8
5		Gemini 3 Pro	Nov 18, 2025	81.72%	22.53	289,752	215.9	156.4
6		Gemini 3 Pro High	Nov 18, 2025	81.60%	21.32	308,553	221.4	160.3
7		Gemini 3 Flash High	Dec 17, 2025	81.11%	20.44	315,112	232.8	144.5
8		GPT-5.2 Codex High	Dec 18, 2025	80.85%	21.26	687,797	261.4	196.4
9		GPT-5.2 High	Dec 11, 2025	80.66%	25.39	974,379	277.1	221.6
10		Claude Sonnet 4.5 Thinking	Sep 29, 2025	80.53%	19.25	416,388	235.0	134.0
11		GPT-5.2 Codex Medium	Dec 18, 2025	80.35%	19.49	442,275	250.6	170.7
12		GPT-5.1 High	Nov 12, 2025	80.13%	20.69	641,854	237.9	167.4
13		Kimi K2 Thinking	Nov 6, 2025	79.22%	17.59	325,258	235.9	134.9

Claude Opus 4.5 Thinking leads with an 83.62% pass rate and the lowest issue density at 15.15 issues per thousand lines of code. It's both the most functionally correct and the cleanest. The drawback? It's slow – not appropriate for small coding fixes.

Gemini 3 Pro stands out as an efficiency outlier – achieving approximately 81% pass rate while maintaining low cognitive complexity and low verbosity. It solves problems with remarkably concise code. But it has the highest issue density among recent models, including the highest rate of control-flow mistakes (200 per million lines of code, nearly 4x higher than Opus 4.5 Thinking).

GPT-5.2 High ranks third in functional performance at about 80.66% but generates the most code of any model – nearly a million lines to solve the benchmark. It has the lowest control-flow error rate (22 per MLOC), but the sheer volume of code it produces creates a review burden.

And here's the kicker that should worry everyone: newer does not automatically mean cleaner. Sonar's data shows GPT-4o with an issue density of 26.08, while GPT-5-minimal comes in at 26.65. Sometimes new models reintroduce basic code smells – catching Exception instead of specific types, forgetting resource management – just to "get by" with a quick solution.

Security is the most concerning dimension. Claude Sonnet 4.5 registered 198 blocker-severity vulnerabilities per million lines. Opus 4.5 Thinking performed significantly better at only 44. But all models share what Sonar calls "a fundamental lack of security awareness" – they consistently produce viable solutions that introduce real security holes. This is Java-specific data, and it should factor heavily into your tool selection.

Kotlin and Scala: The Specialist Question

Kotlin is in an interesting position thanks to one factor: **Mellum**. JetBrains' purpose-built 4B parameter model, now open-sourced with a Kotlin-specific fine-tuned variant, is the only model with dedicated optimization for Kotlin code completion. It powers AI Assistant in JetBrains IDEs with a 40% acceptance rate and latency reduced to one-third of previous versions. Since its open-sourcing in mid-2025, JetBrains has expanded it with **Mellum-all** – supporting all major languages including Java – plus a VS Code extension and an SDK for embedding into custom tools.

For general-purpose Kotlin work beyond code completion, Claude models tend to handle Kotlin surprisingly well, especially expressive syntax and functional-style constructs. The Spring–Kotlin partnership formalized in 2025 (with 27% of Spring developers using Kotlin) means there's more Kotlin training data in the ecosystem than ever.

Scala, unfortunately, remains the hardest case. No mainstream model has dedicated fine-tuning for Scala. The best results come from Claude Opus 4.5, which benefits from a strong grasp of functional programming concepts, and GPT-5, which shows solid performance on Scala tasks in benchmarks like MultiPL-E. But they still require more guidance and validation than their Java or Kotlin counterparts. If you're doing serious Scala work, expect to spend more time on prompt engineering and code review.

Practical Recommendations for February 2026

Before the recommendations, one crucial caveat: when we say an agent scores 25% on SWE-bench Pro, we're measuring well-defined issues with explicit requirements against unit tests. The high-leverage parts of real engineering – collaborating on specs, translating ambiguous requirements into clean interfaces, writing secure and maintainable code – remain entirely unmeasured. The

UTBoost paper demonstrated that many SWE-bench solutions pass unit tests without actually resolving the underlying issues. Keep this gap in mind.

For everyday Java development (the “daily driver” question): Claude Sonnet 4.5 offers the best balance of speed, quality, and cost. It’s fast enough for iterative work, smart enough for most tasks, and the Brokk data shows it’s competitive on real Java codebases. GPT-5 Mini is a strong alternative if you’re on a tighter budget.

For complex, multi-file Java debugging: Claude Opus 4.5 pulls clearly ahead. The combination of highest functional pass rate, lowest code quality issues on Java (per Sonar), and strong reasoning across large contexts makes it the best choice when correctness matters more than speed.

For Kotlin-specific work: Use Mellum (via JetBrains AI Assistant or the VS Code extension) for code completion. For generation and refactoring, Claude Sonnet 4.5 or Opus 4.5 handle Kotlin idioms well.

For Scala: Claude Opus 4.5, with the caveat that you’ll need more validation. Consider building project-specific prompt templates that capture your conventions.

For cost-conscious teams: GPT-5 Mini or DeepSeek V3.2, but invest the savings into better code review tooling (SonarQube, spotbugs) to catch the quality issues these models introduce more frequently.

For teams wanting to self-host: GLM-4.7 is currently the strongest open-source option, and DeepSeek V3.2 offers excellent cost-efficiency. But be realistic about the reliability gap – expect more babysitting than with frontier models.

Things to actively avoid: Trusting any single benchmark. Relying on HumanEval scores (it’s 2026, please). Assuming SWE-bench scores predict Java performance (they don’t, consistently). Running a model once and calling it an evaluation (LLM performance is probabilistic – run it three times minimum).

The state of LLM benchmarking for JVM languages in early 2026 is... getting better, but still messy. Python still dominates research, but genuinely JVM-focused benchmarks are finally here – the Brokk Power Ranking for Java, the Sonar LLM Leaderboard for Java code quality, and Mellum for Kotlin specialization. Scala is still waiting.

The difficulty of designing good benchmarks actually makes me optimistic about coding agents. Current benchmarks fall short of capturing the nuance and messiness of real engineering work – yet the agents are already remarkably capable. There is substantial low-hanging fruit in benchmark design: property-based testing instead of unit tests, formal methods where possible, starting from PRDs and specs, and testing information acquisition skills rather than assuming perfect problem statements. As these improvements arrive, we should expect corresponding improvements in agent capabilities through better training

signals.

Please remember, the current solutions are the worst we will ever get.

-> Back to the Table of Content

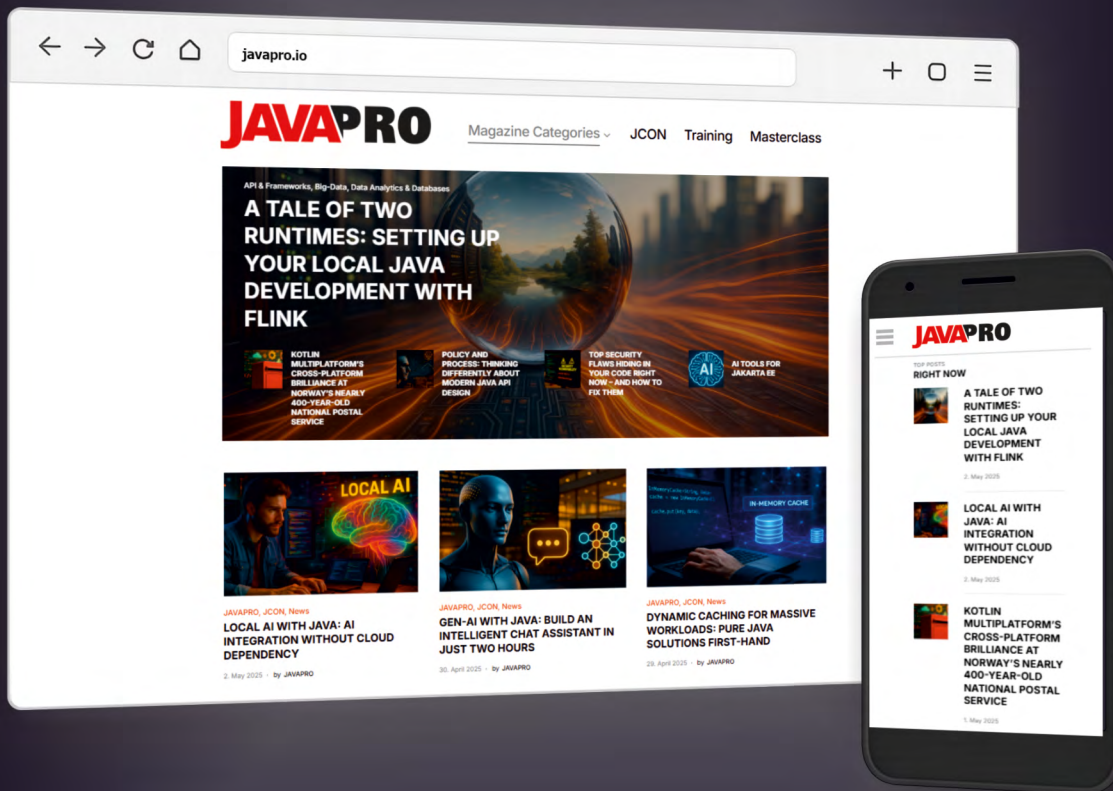
CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

#JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:

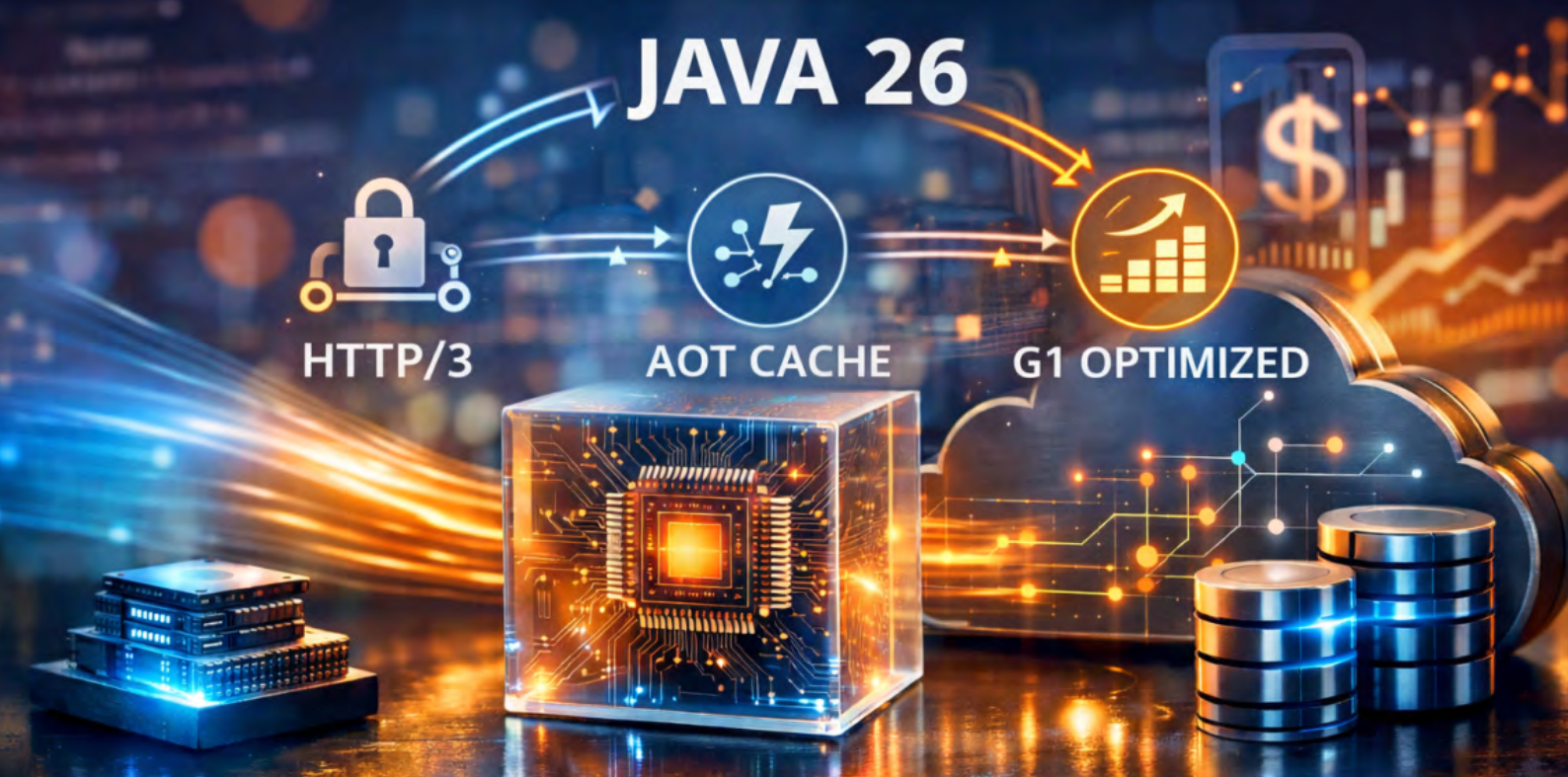


BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

www.javapro.io

www.javapro.io



#JAVAPRO #GENAI

Java 26 + Agentic Microservices: Cloud Performance Re-Imagined for FinTech

Author:



Sibasis Padhi is a Staff Software Engineer at Walmart Global Tech with over 18 years of experience in building high-performance systems. His expertise lies in Microservices Architecture, Cloud Performance Optimization, and Secure FinTech Transaction Systems. He specializes in transforming legacy financial applications into scalable, fault-tolerant, and cost-effective cloud-native solutions. He has authored multiple technical articles on DZone and other platforms, and I'm a speaker at global tech events, including IEEE Cloud Summit 2025. He also serves as a judge and reviewer for developer challenges and technical books, including those from Manning Publications. His current work focuses on optimizing Java-based microservices for high-volume transactions in distributed cloud environments. Through JAVAPRO, he hopes to contribute practical insights and problem-solving patterns for the Java community, especially in the areas of performance tuning, observability, and real-world deployment challenges.

<http://www.linkedin.com/in/sibasis-padhi-434aa89>

The Problem This Solves

Java 26 Agentic Microservices redefine how FinTech systems manage performance under real-world cloud conditions. Instead of relying on static JVM tuning, this model combines Java 26's HTTP/3 support, AOT object caching, and G1 improvements with a guardrailed control loop that adapts safely to network variance, scale-out events, and runtime contention.

Java 26 adds three concrete platform levers that map directly to these failure modes:

- **Transport stability under variance:** HTTP/3 in `java.net.http.HttpClient` (JEP 517)
- **Cold start + warmup economics:** Ahead-of-Time (AOT) *object* caching usable with any GC, including ZGC (JEP 516)
- **More throughput with less contention:** G1 throughput improvements by reducing synchronization (JEP 522)

This article shows how to operationalize those levers as a **controlled performance workflow**: observe → decide → act → verify, with guardrails that protect SLOs and auditability.

Java 26 Agentic Microservices: A Performance Control Plane

Most production FinTech systems already have two planes:

- **Data plane:** payments, risk scoring, planning or forecasting, ledger updates
- **Control plane:** routing, feature flags, rate limits, rollouts, autoscaling, SLO monitoring

"Agentic" here means the control plane runs a **closed-loop controller**:

Observe (real signals) → **Decide** (policy-bounded) → **Act** (reversible) → **Verify** (baseline comparison) → **Rollback** (automated)

In regulated systems, an optimization action is acceptable only when it is:

- **Bounded:** limited scope (one route, one region, 1–5% canary)
- **Reversible:** fast rollback path exists
- **Auditable:** decision + evidence + outcome are recorded
- **SLO-safe:** guardrails override the controller

Java 26 matters because it expands the set of actions that are **high-impact** and **safe to roll**.

Java 26 Agentic Microservices: Transport Wins with HTTP/3 (JEP 517)

HTTP/3 carries HTTP semantics over QUIC (UDP), which changes the failure profile of client-server communication in ways that matter for production finance. The key value is not "a faster best case," but **more predictable tail latency** in noisy networks. QUIC is generally more resilient under packet loss, and jitter supports connection migration when network paths change mid-session (common with mobile and edge scenarios), and avoids some head-of-line blocking behavior associated with TCP multiplexing.

Java 26 introduces HTTP/3 support in the standard `java.net.http.HttpClient` (JEP 517), while keeping HTTP/2 as the default, but an important design choice for FinTech teams that require opt-in, policy-driven rollout rather than a risky global switch.

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpClient.Version;
import java.time.Duration;

public class PartnerClient {

    private static final HttpClient client = HttpClient
        .newBuilder()
        .version(Version.HTTP_3) // Prefer HTTP/3
```

```
.connectTimeout(Duration.ofSeconds(2))
    .build();

public static String callPartner() throws Exception {
    HttpRequest request = HttpRequest.newBuilder()
        .uri(new URI("https://partner-api.example.com/score"))
        .timeout(Duration.ofSeconds(3))
        .GET()
        .build();

    HttpResponse<String> response =
        client.send(request, HttpResponse
            .BodyHandlers.ofString());

    return response.body();
}
```

In FinTech, HTTP/3 is most valuable on outbound calls that are both **latency-sensitive** and **operationally variable**. That includes mobile-facing APIs and mobile-to-edge paths where packet loss is common, cross-region service calls where jitter inflates p99, and third-party integrations (KYC/KYB, tokenization, risk feeds) where you don't control the network or the remote stack. It also shows up in fan-out workflows with risk scoring, enrichment, and partner validation, where a small number of slow dependencies can dominate end-to-end p99 even when the rest of the system is healthy.

The right operational posture is to treat HTTP/3 as a **route policy** rather than a platform default. Practically, teams use strategies like upgrading when the server advertises HTTP/3 (Alt-Svc), racing HTTP/3 and HTTP/2 on the first request and then sticking with the better result, or preferring HTTP/3 only when health signals remain good and rolling back quickly on handshake/ timeouts. The safety model is consistent: enable per-route/per-region behind a feature flag, separate error classes (timeouts vs handshake/TLS vs application errors), watch retry rates and tail latency (not averages), and enforce automated rollback when error budget burn accelerates, or p99 regresses beyond a threshold.

Java 26 Agentic Microservices: AOT Caching for Cold-Start Control (JEP 516)

In cloud FinTech, a large portion of "tail pain" shows up during scale-out. New pods start cold: classes load, frameworks initialize, caches warm, and JIT compilation ramps. Ahead-of-Time (AOT) caching helps by letting the JVM reuse cached artifacts produced during a controlled training run, so the service reaches a steady operating point faster when new instances come online. Java 26's JEP 516 makes this more broadly usable by extending AOT object caching to work with any garbage collector, including low-latency ZGC, by allowing cached objects to be loaded from a GC-agnostic format that can be streamed into memory, rather than relying only on GC-specific mapped objects.

This is a real lever in FinTech because teams regularly face a tradeoff between two things they need at the same time: low-latency GC behavior for p99 stability and faster warm starts to reduce scale-out penalties. Historically, those goals could push you toward different runtime choices or operational compromises. JEP 516 reduces that friction by making AOT object caching compatible across collectors, so you can pursue faster startup/warmup without forcing an unwanted GC decision.

```
# Training run to generate AOT cache
java -XX:AOTMode=record -XX:AOTConfiguration=app.aotconfig \
  -XX:AOTCacheOutput=app.aotcache \
  -jar payment-service.jar

# Production run using generated cache
java -XX:AOTMode=auto -XX:AOTCache=app.aotcache \
  -jar payment-service.jar
```

Operationally, the important nuance is that “cold” and “warm” are not fixed states in production. JEP 516 describes both mapped GC-specific objects (often extremely fast when the filesystem cache is already warm) and streamed GC-agnostic objects (sequential loading that works across collectors and can be effective for true cold starts, sometimes benefiting from spare CPU to hide I/O latency). Real clusters routinely oscillate between these states due to node churn, image caching, and scheduling variance, which is why AOT caching fits best as a policy-driven control rather than a one-time switch.

An agentic control loop makes AOT caching safe by treating the cache as a deployable artifact with lifecycle discipline. You generate it in a training run using representative startup and traffic, ship it with the image (or mount it as an artifact), and then canary cache-on versus cache-off using the exact metrics you care about, cold-start time, time-to-first-success, and time-to-stable p99. You promote only when the workload signature is stable, and you regenerate automatically when dependency/configuration/profile drift indicates the cache is stale. The safety net is straightforward. Drift detection, canary-only rollout for new cache versions, separate measurement of “first success” vs “stable p99,” and an immediate cache-off fallback path for rollback if reality disagrees with expectations.

Java 26 Agentic Microservices Runtime Headroom: G1 Throughput Improvements (JEP 522)

G1 remains a common default for services that need balanced throughput and latency. Under heavy concurrency, synchronization between application threads and GC threads can add overhead and latency noise. JEP 522 improves throughput by reducing synchronization. It introduces a **second card table** so application write barriers can update one table without synchronization while GC optimizer threads work on another, reducing contention and simplifying barriers.

Why FinTech Teams Should Care

- more work per core at a stable SLO (cost efficiency)
- less latency variance from GC contention (cleaner control signals)
- fewer “mystery regressions” from contention interactions in allocation-heavy paths

Safe Operational Posture

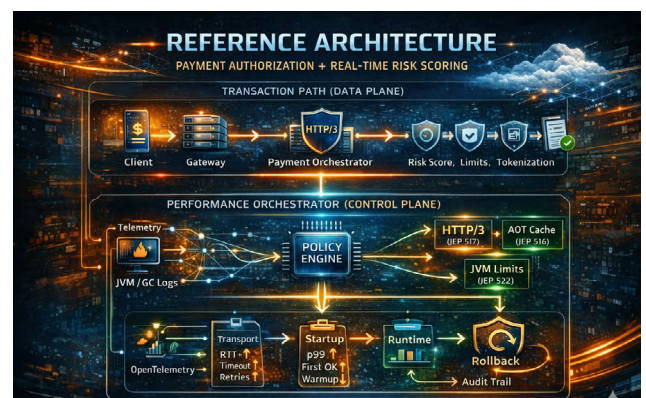
The primary “action” is upgrading to Java 26 and re-baselining. The control loop’s role is to prevent destabilizing overreaction:

- re-baseline p95/p99, allocation rate, pause distribution, CPU cost per request
- adjust bounded knobs only when signals show sustained drift (queue depth, max in-flight, rate limits)
- maintain rollback gates like any other performance change

Java 26 Agentic Microservices Reference Architecture: Performance as a First-Class Workflow

A practical reference architecture treats performance as a first-class workflow, layered onto a concrete FinTech use case such as payment authorization with real-time risk scoring. The data plane remains the transaction path with Client to Gateway to Payment Orchestrator, invoking risk scoring, limits checks, tokenization, and ledger updates before returning a response. Alongside it runs a dedicated control plane that continuously ingests telemetry (OpenTelemetry metrics, JVM, and GC signals), evaluates policy rules, stays within range, verifies outcomes, and rolls back when necessary. The signals collected must span

- transport (route-level RTT variance, handshake failures, retry and timeout rates),
- startup behavior (time-to-readiness, time-to-first-success, and warmup time to stable p99), and
- runtime characteristics (allocation rate, pause distribution, CPU throttling, queue depth, saturation).



These signals drive actuators that map directly to Java 26 capabilities, route-level HTTP/3 preference with fallback strategy, AOT cache enablement and streaming-mode decisions, and controlled rollout of the Java 26 runtime with bounded service-level adjustments such as in-flight request limits or concurrency

caps. Verification is non-negotiable where changes are introduced via small canaries (1–5% traffic), compared against a stable baseline over the same load window, automatically rolled back on p99 regression or accelerated error budget burn, and recorded in an audit trail documenting what changed, why it changed, and what measurable outcome followed.

A Realistic Case Study Pattern

Baseline (What You Measure Before Changing Anything)

You can publish a credible method without disclosing proprietary numbers by being explicit about *conditions* and *metrics* rather than exact values. Start with a baseline that reflects how many FinTech systems run today with Java 25 in the runtime, partner calls over HTTP/2, and AOT caching disabled. Drive traffic with a steady request rate and then introduce a controlled burst that mimics a market event or incident-driven spike. Capture the metrics that actually define production health with end-to-end p99 latency, partner-call p99 latency, retry rate, error rate, and CPU cost per request. Then see what happens during scale-out when fresh pods come online (cold-start behavior and the time it takes for latency to stabilize after new instances are added).

Intervention (What Changes, Kept Deliberately Narrow)

Next, apply a controlled intervention that isolates Java 26's levers without changing the system design. Upgrade only a small canary slice to Java 26 and keep the rest of the fleet untouched. Enable HTTP/3 only for the single partner endpoint that shows the highest network variance, and leave all other outbound calls on HTTP/2 so you can attribute any improvements to a specific route policy. Introduce an AOT cache artifact and enable it only for the canary during the scale-out windows where cold starts usually hurt the most. The most important discipline is what you do not change is keep topology, resource limits, routing rules, and dependency configuration identical so the experiment stays attributable and auditable.

Outcomes and Decision Logic (What "Success" and "Rollback" Look Like)

The outcomes you look for are directional and operational. A smaller cold-start penalty, faster stabilization of p99 after scale-out, fewer retries/timeouts under partner or network variance, and improved efficiency (lower CPU cost per request or higher throughput at the same SLO). Just as important is proving the safety mechanism works with rollback triggers that must remain intact, measurable, and automatic when error budget burn accelerates, or p99 regresses. If the results are negative, that is still progress because the control loop should detect the regression, roll back cleanly, and preserve a recorded evidence trail showing what changed, why it changed, and what the system did in response.

In my experiments, I used a controller pattern similar to the open-source Agentic Cloud Optimizer (ACO): continuously ingest JVM + service signals, propose bounded changes, run

canary verification, and store an audit trail for rollback decisions. The article's focus remains Java 26's levers and the control-loop method.

What to Try Monday

1. Pick one latency-sensitive route with high network variance (partner integration is ideal).
2. Add explicit protocol selection behind a feature flag and canary HTTP/3 first.
3. Add a cold-start benchmark with deploy, measure time-to-first-success, and measure time-to-stable-p99.
4. Introduce an AOT training run and ship the cache artifact; canary cache-on vs cache-off.
5. Upgrade a canary slice to Java 26, re-baseline G1 behavior, then tune only bounded controls.
6. Enforce rollback gates on p99 and error budget burn.

Conclusion

Java 26 expands the set of performance improvements that can be rolled out safely: HTTP/3 for noisy networks, AOT object caching that works across garbage collectors, and G1 throughput improvements that reduce contention. Agentic microservices, defined as a guardrail control plane, turn these platform capabilities into a repeatable workflow that preserves reliability, auditability, and cost discipline under real FinTech operating conditions.

-> Back to the Table of Content



#JAVAPRO #GENAI

Building Production-Ready AI Agents with Java and Spring AI



Authors:



With over 25 years in the tech, Yuriy Bezsonov has progressed from a software developer to an engineering manager and Solutions Architect. Now, as a Senior Solutions Architect at AWS, he assists partners and customers in developing cloud solutions, focusing on container technologies, Kubernetes, Java, application modernization, SaaS, developer experience, and GenAI. Yuriy holds AWS and Kubernetes certifications, and he is a recipient of the AWS Golden Jacket and the CNCF Kubestronaut Blue Jacket.

<https://www.linkedin.com/in/yuriybezsonov/>

Java developers have been building enterprise applications for decades, but when it comes to AI, the conversation has been dominated by Python. Spring AI changes this situation. It brings the same patterns we know from Spring – dependency injection, auto-configuration, portable abstractions – to AI development.

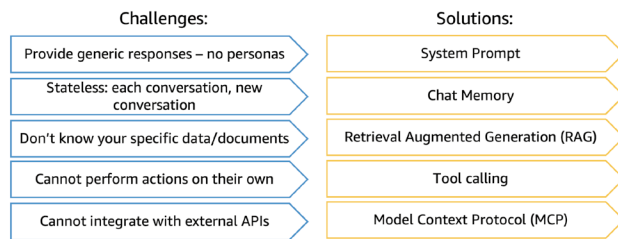
In January 2026 we built a sample agent application, and we want to share what we learned. The sample AI agent handles conversations with memory, answers questions from internal knowledge bases, uses external APIs, and integrates with existing

Sascha Möllering is a Principal Specialist Solutions Architect for Containers at AWS, where he has been helping customers architect and optimize their cloud-native applications for nearly a decade. With a deep passion for container technologies, serverless computing, and Java development, he works with organizations across the globe to design and implement scalable, resilient solutions on AWS. A regular speaker at industry conferences and technical events, Sascha shares his expertise through hands-on workshops, blog posts, and technical content. His practical experience and technical depth have made him a trusted voice in the cloud-native community. He is particularly known for his work in containerization strategies and Java application modernization on AWS. When not architecting solutions or speaking at events, Sascha actively contributes to the technical community through writing, mentoring, and developing educational content that helps organizations navigate their cloud transform.

<https://www.linkedin.com/in/smoell/>



microservices – all running on Amazon Bedrock. If you want to try it yourself, the Building AI Agents with Java and Spring AI workshop [1] walks you through typical challenges of Generative AI models and provides step by step solutions with code available on GitHub [2].



Choosing the Right Model

Before writing any code, you need to pick a model. Amazon Bedrock gives you access to foundation models from Anthropic, Amazon, Meta, and others through a unified API. The choice matters – it affects how your agent behaves, how much you pay, and how fast it responds.

Some models only process text, while multimodal models like Anthropic Claude and Amazon Nova 2 can analyze images, PDFs, and understand visual context. If your agent needs to process expense receipts or read documents, you'll want multimodal capabilities.

Beyond modalities, think about the trade-off between intelligence, speed, and cost:

- **Anthropic Claude Opus** works best when intelligence is your top priority. Complex tasks requiring strong reasoning capabilities. You're choosing quality over speed and cost.
- **Anthropic Claude Haiku** shines when speed matters most. Real-time user interactions or high-volume processing where you need the fastest possible responses.
- **Anthropic Claude Sonnet** hits the sweet spot for most applications – good intelligence, reasonable speed, manageable cost.
- **Amazon Nova 2 Lite** is a fast, cost-effective reasoning model for everyday workloads. It offers industry-leading price performance and helps enterprises and developers build capable, reliable, and efficient agentic-AI applications.
- **Amazon Nova 2 Pro** is the most intelligent Nova model, designed for highly complex, multistep tasks like multi-document analysis and software migrations.

We started development with Anthropic Claude Sonnet 4 and found it worked well for our agent use case. For production, we'd profile the actual workload and consider Nova 2 Lite if cost optimization becomes important.

```
spring.ai.bedrock.converse.chat.options.model=global.
anthropic.claude-sonnet-4-20250514-v1:0
```

ChatClient: The Foundation

The ChatClient interface is where everything comes together in Spring AI. It's a unified API for talking to AI models – one interface that works with Amazon Bedrock, OpenAI, Azure OpenAI, Google Vertex AI, or local Ollama. You can start development with Ollama locally (free, private, no API costs), validate against OpenAI, and deploy to Amazon Bedrock for production – same code, different configuration.

What makes ChatClient powerful is the Advisor pattern. Advisors handle cross-cutting concerns like memory, Retrieval-Augmented Generation (RAG), and logging without cluttering your business logic. Think of them as middleware for AI conversations – each advisor intercepts requests and responses, modifying prompts or processing outputs.

Here's what our ChatClient configuration looks like after adding all the capabilities:

```
this.chatClient = chatClientBuilder
    .defaultSystem(DEFAULT_SYSTEM_PROMPT)
    .defaultAdvisors(
        MessageChatMemoryAdvisor.builder(chatMemory).build(),
        QuestionAnswerAdvisor.builder(vectorStore).build()
    )
    .defaultTools(new DateTimeTools(), new WeatherTools())
    .defaultToolCallbacks(mcpTools)
    .build();
```

One configuration gives you conversation memory, RAG-powered knowledge retrieval, custom tools, and Model Context Protocol (MCP) integrations. The advisor chain runs in order – memory loads history, RAG retrieves relevant documents, then the enriched prompt hits the model with all tools available.

Conversation Memory

When we first tested our agent, users would introduce themselves, ask a few questions, then say "What's my name?" – and the agent had no idea. Every request was a blank slate. That's how AI models work: they're stateless. Your application needs to manage conversation history and feed it back with each request.

Spring AI handles this with MessageChatMemoryAdvisor. You configure a memory store, wire it into the ChatClient, and the advisor automatically loads conversation history before each request and saves new messages after.

For storage, you have options: InMemory for development, JDBC for production with existing databases, Redis for distributed caching, Cassandra for scale. We went with JDBC and PostgreSQL since most Spring applications already have a relational database – no new infrastructure to manage.

```
var chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(JdbcChatMemoryRepository.builder()
        .dataSource(dataSource)
        .dialect(new PostgresChatMemoryRepositoryDialect())
        .build())
    .maxMessages(20)
    .build();
```

For production applications with longer conversations, consider a three-tier memory architecture: session memory for recent messages (around 20), context memory for conversation summaries, and preferences memory for user profile data. This gives users continuity without burning tokens on full history. We documented this pattern in detail [8] if you want to implement it.

There's also an easier path if you're deploying to Amazon Bedrock AgentCore [5]. AgentCore provides fully managed memory – event memory for conversation history with automatic retention, and semantic memory for long-term knowledge extraction. Add the `spring-ai-bedrock-agentcore-starter` [6] and memory works out of the box, no custom code required.

Knowledge with RAG

Ask a generic AI model about your company's travel policy limits and you'll get a confident, plausible, completely wrong answer. The model hallucinates because it's working from general training data, not your actual documents.

RAG (Retrieval-Augmented Generation) solves this by grounding responses in real content. During ingestion, documents get chunked, converted to vector embeddings, and stored. At query time, the user's question becomes an embedding, similar chunks get retrieved through semantic search, and those chunks go into the prompt as context. The model generates responses based on your actual documents instead of guessing.

Spring AI's `VectorStore` abstraction supports `PGVector`, `OpenSearch`, `Pinecone`, `Weaviate`, `Milvus`, and more. We used `PGVector` since we already had PostgreSQL for memory – same database, no extra services. The `QuestionAnswerAdvisor` handles retrieval automatically once configured:

```
spring.ai.bedrock.titan.embedding.model=amazon.titan-embed-
text-v2:0
spring.ai.vectorstore.pgvector.initialize-schema=true
spring.ai.vectorstore.pgvector.dimensions=1024
```

Spring AI 2.0 brings support for Amazon Bedrock Knowledge Bases – fully managed RAG where AWS handles document chunking, embeddings, and vector storage. You connect S3, Confluence, or SharePoint as data sources, and get advanced features like hybrid search (combining semantic and keyword matching), reranking, and metadata filtering. The same `QuestionAnswerAdvisor` pattern works, just with a different `VectorStore` implementation. We wrote a hands-on guide "RAG Made Serverless – Amazon Bedrock Knowledge Base with Spring AI" [9] if you want to try it.

Tools and MCP

Our agent could remember conversations and answer questions from documents, but it couldn't tell users the weather for their trip or even what day it was. AI models have knowledge cutoffs – they don't know today's date, current weather, or what's in your database.

Tool calling bridges this gap. You define functions the AI can call, and the model decides when to use them based on the conversation. Spring AI's `@Tool` annotation makes this straightforward:

```
@Tool(description = """
    Get weather forecast for a city on a specific date.
    Use for answering questions about weather forecasts.
    """)
public String getWeather(
    @ToolParam(description = "City name, such as Paris,
        London, New York") String city,
    @ToolParam(description = "Date in YYYY-MM-DD format")
    String date) {
    // Call weather API and return formatted result
}
```

The description matters more than you might expect. The AI reads it to decide which tool to call, with what parameters, in what order. Vague descriptions lead to wrong tool selection; precise ones enable accurate autonomous behavior.

For larger systems, hardcoding every API into your agent doesn't scale well. Different teams own different services with their own release cycles. You don't want to redeploy the agent every time the booking team adds an endpoint.

Model Context Protocol (MCP) [7] standardizes how agents discover and use tools. MCP servers expose tools through a standard protocol; clients discover them at runtime. Spring AI's MCP server starter turns existing microservices into AI-accessible tools – add the dependency, annotate methods with `@Tool`, and your service becomes an MCP server. The booking team maintains their tools, HR maintains theirs, and your agent discovers everything at startup without code changes.

```
@Tool(description = """
    Find flights between two cities.
    Requires: departureCity - Name of the departure city,
        arrivalCity - Name of the arrival city.
    Returns: List of available flights sorted by price from
        lowest to highest.
    """)
public List<Flight> findFlightsByRoute(String departureCity,
    String arrivalCity)
{
    return flightService.findFlightsByRoute(departureCity,
        arrivalCity);
}
```


One thing we learned: agents with tools often have large system prompts and tool definitions that are sent with every request. Amazon Bedrock prompt caching (available for Anthropic models) reduces costs by caching this repeated content:

```
spring.ai.bedrock.converse.chat.options.cache-options.  
strategy=SYSTEM_AND_TOOLS
```

Structure your prompts with stable content first (system instructions, tool definitions) and user-specific content last. Cache hits can reduce costs significantly on the cached portions.

Deploying to AWS

Spring AI agents are standard Spring Boot applications packaged as container images. They run anywhere containers run. On AWS, you have several options depending on how much infrastructure you want to manage.

Amazon Elastic Kubernetes Service (Amazon EKS) gives you Kubernetes flexibility with Amazon EKS Auto Mode fully automates Kubernetes cluster management for compute, storage, and networking on AWS with a single click. Good fit if your team already knows Kubernetes and needs fine-grained control over networking and scaling.

Amazon Elastic Container Service (Amazon ECS) offers serverless containers with AWS Fargate. Amazon ECS Express Mode automates infrastructure setup including domains, networking, and load balancing – turnkey deployment when you want to get running quickly. Amazon ECS Managed Instances provide more control over instance types and pricing (Spot, Reserved) while ECS handles provisioning.

AWS Lambda works well for event-driven workloads with sporadic traffic patterns where you want to scale to zero.

Amazon Bedrock AgentCore is a fully managed agent runtime with built-in memory, code interpreter, browser tools, gateway, and observability. The Spring AI AgentCore starter [6] provides adapters for running Spring AI applications on Amazon Bedrock AgentCore with just one annotation `@AgentCoreInvocation`:

```
@Service  
public class InvocationService {  
  
    @AgentCoreInvocation  
    public Flux<String> handleInvocation(InvocationRequest  
request, AgentCoreContext context) {  
        // Parse JWT, extract user identity for memory  
        isolation  
        ...  
        return chatService.chat(request.prompt(), sessionId);  
    }  
}
```

AgentCore integrates with Amazon Cognito for user authentication (OAuth 2.0). JWT tokens are validated automatically, and user identity flows through to your application for memory isolation and audit logging. VPC mode enables access to databases and internal services. IAM roles provide least-privilege access to Bedrock models.

The same Spring AI application code works across all these deployment targets. The difference is how much infrastructure you want to manage versus how much you want AWS to handle for you. For the fastest path to production with built-in agent capabilities, Amazon Bedrock AgentCore is hard to beat.

Wrapping Up

Spring AI gives Java developers vendor-neutral abstractions for building AI agents – ChatClient for model access, ChatMemory for stateful conversations, VectorStore for RAG, Tools and MCP for real-time data and integrations. The patterns feel familiar if you've worked with Spring before.

Model selection matters more than we initially expected. We started with Anthropic Claude Sonnet and it worked well, but profiling your actual workload helps you optimize – Amazon Nova 2 Lite for cost-effective reasoning, Anthropic Claude Haiku for speed, Anthropic Claude Opus when you need maximum intelligence.

For deployment, Amazon Bedrock AgentCore provides the fastest path to production with built-in memory, security, and observability. When you need more control, the same code runs on Amazon EKS, Amazon ECS, or AWS Lambda.

We hope this gives you a good starting point. The workshop [1] walks through everything in detail if you want to build it yourself.

References:

- [1] Workshop: <https://catalog.workshops.aws/java-spring-ai-agents>
- [2] GitHub: <https://github.com/aws-samples/java-on-aws>
- [3] Spring AI: <https://docs.spring.io/spring-ai/reference/>
- [4] Amazon Bedrock: <https://aws.amazon.com/bedrock/>
- [5] Amazon Bedrock AgentCore: <https://aws.amazon.com/bedrock/agentcore/>
- [6] Spring AI AgentCore Starter: <https://github.com/spring-ai-community/spring-ai-bedrock-agentcore>
- [7] Model Context Protocol: <https://modelcontextprotocol.io/>
- [8] Memory Implementation: <https://dev.to/yuriybezsonov/a-practical-guide-to-building-ai-agents-with-java-and-spring-ai-part-2-add-memory-odn>
- [9] Bedrock Knowledge Bases: <https://dev.to/yuriybezsonov/rag-made-serverless-amazon-bedrock-knowledge-base-with-spring-ai-2dn9>

-> **Back to the Table of Content**



#JAVAPRO #GENAI

A2A Building Interoperable AI Agents with Java



Why Agents Need Friends?

We've recently seen a boom in AI "agents". AI Agents are usually described as a service that talks to an AI model to perform some kind of goal-based operation using tools and context it assumes.

But most of these agents are still working in an isolated environment. You build an agent into your application and it offers some capabilities, but that's it. You're basically building a monolith with AI in it.

The Idea Behind A2A

The initial concept is really simple, let agents talk to each other.

If we draw a quick parallel to how we work as humans, if tomorrow I want to build a house I will probably not try to do it by myself. I will call different experts with their own expertise and skills, and they will work together to build the house. Why should your AI Agent rely on a "do-everything" agent then ?

In the real world, collaboration is how things get done. A2A is exactly that. It allows specialized agents to communicate, coordinate and share work.

What's the A2A Protocol ?

A2A is a protocol that was initially developed by Google, but that is now part of the Linux Foundation. Several big major tech companies are contributing to the project.

The protocol defines how agents can:

- discover other agents
- exchange messages and tasks (more on that later)
- work securely together
- share results or errors

Build on Familiar Web Standards

A2A is built using web standards that most developers already know. You can actually use different protocols:

- HTTP/REST
- JSON-RPC
- gRPC

Author:

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



<https://www.linkedin.com/in/lmagnette/>

This makes any kind of integration flexible and future-proof. The only issue with supporting different protocols is that all agents supporting A2A might not use the same one, making it a bit more difficult to use.

Secure by Default

We've all heard about the infamous lack of "S" for security in MCP. A2A is by default secure. It supports enterprise grade authentication and authorization, by being on par with OpenAPI's authentication schemes.

Discoverability

To be able to communicate with each other, agents need to, first, be able to discover each other. To do so, each agent will provide an Agent card. This card is really similar to a resume. It will contain the identity of the agent and its capabilities.

The agent card describes the agent's name, version, capabilities (eg: streaming, notifications,...) and its specific skills. The card is exposed at a well-known URL: `.well-known/agent.json`

To Task or Not To Task?

A2A introduces the concept of task. A task, as the name suggests, is a unit of work. A2A can work with or without tasks but I personally prefer to use them. A task can be synchronous or asynchronous. You probably don't want to block your agent while it's doing some heavy computation. A task has a state, and we will see a bit further how its state can vary during the execution of the task.

Since A2A supports long running tasks, it needs a way to communicate with the client agent. You could always go for a polling approach, but that would be really inefficient. A2A supports SSE (Server Sent Events) and Webhooks allowing agents to communicate with each other in a more efficient way. It also supports reconnection if the SSE connection is lost. SSE is a great solution for real-time communication, but for really long-running tasks, using webhooks is likely the better option.

Not Only Text

AI models can handle a lot more than just text. We often talk about multi-modality where models, like Gemini from Google, support a large variety of inputs (text, images, audio, video,...). As you can imagine A2A supports multimodality out of the box for the inputs and outputs. It allows us to build very rich and capable agents that can interact with each other.

Extensible

A2A is built to be extended for specific use cases. A2A offers a lot of capabilities, but for certain aspects we might need more specific features. There are already some existing extensions for A2A, such as AP2, that offer a payment protocol over A2A.

How Does the A2A Protocol Work?

As discussed earlier, the first step is to discover other agents and get their agent cards. Once it's done, your agent will be able to communicate with other agents and create tasks. Those tasks will need to be managed by the agents and for that there are various states. Let's see them in action. In the following step, it will be assumed to have 2 agents. The first agent will be used as a client and we will call it **Neo**. The second agent will be used as a server and we will call it **Smith**.

Happy Flow

The first step is to submit a task. In this scenario, **Neo** will send a task to **Smith**. Once **Smith** receives the task and accepts it, it will set the state to **SUBMITTED**.

Once **Smith** starts working on the task, he will update the state to **WORKING**.

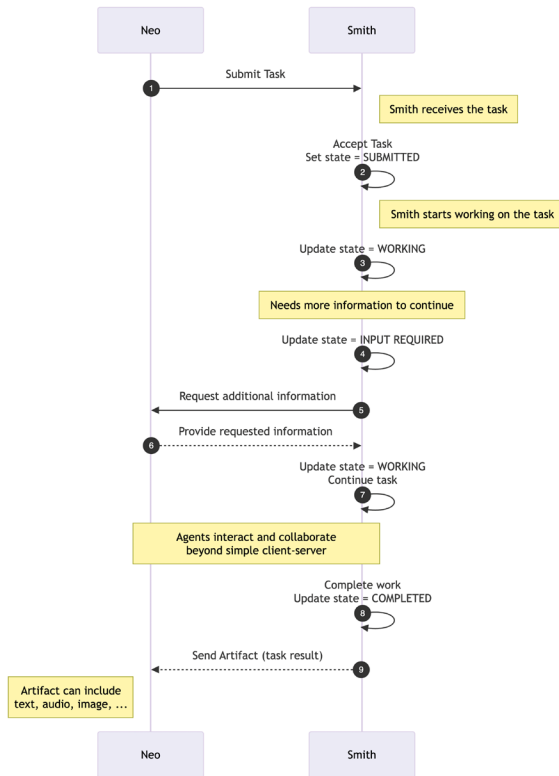
At some point, **Smith** might realize he needs more information from **Neo** to complete the task. In that case, it will update the state to **'INPUT REQUIRED'** and send a message to **Neo** asking for more information.

If **Neo** responds with the required information, **Smith** will update the state back to **WORKING** and continue working on the task.

This interaction is key in the protocol since it shows that the agents are not working just as a client-server solution but can really interact and collaborate with each other.

At some point, we can imagine that **Smith** will have completed his work, update the state to **COMPLETED** and send a message to **Neo** with the result of his work. This result is what A2A calls an Artifact. An artifact is the result of a task it might contain one or multiple parts with different type of data (text, audio, image,...) giving us multimodal returns.

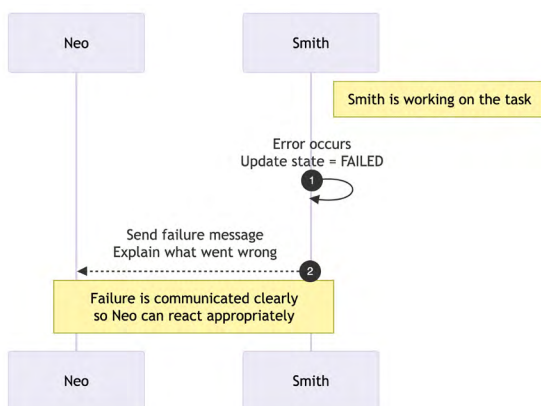
You can see here a diagram illustrating the happy flow.



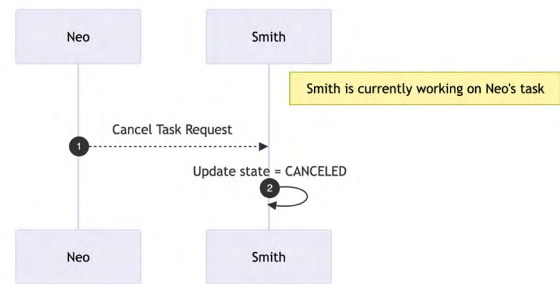
What Could Go Wrong?

We all know that things don't always go as planned, to handle that A2A has a couple of useful states that we'll go through in the following sections.

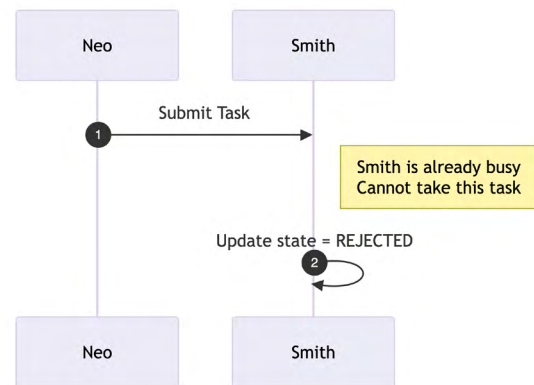
Let's imagine that Smith is working on a task and for some reason fails at some point. In that case, the task state will be updated to **FAILED**. Smith will then send a message to Neo explaining what went wrong.



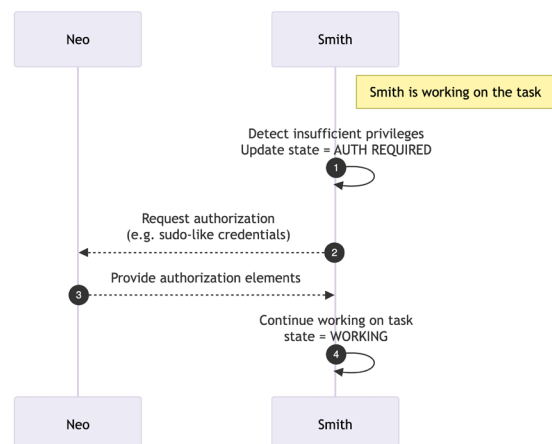
Another case could be that Neo gave a task to Smith but after all realizes it doesn't need Smith to complete his work. In that case, Neo sends a message to Smith to cancel the task and Smith will update the state to **CANCELED**.



In the following scenario, Neo sends a task to Smith. But what Neo doesn't know is that Smith is already very busy and will not be able to work on it. In that case, Smith will update the state to **REJECTED** and send a message to Neo explaining that he will not be able to work on the task.



Finally let's imagine that Smith is working on a task for Neo but at some point realizes that Neo does not have the necessary privileges for Smith to complete the action. It's the classical case of the sudo on linux. In order to complete the action, Smith will update the state to **AUTH REQUIRED** and send a message to Neo asking for authorization. If then Neo provides the necessary authentication elements to Smith, then Smith will continue working on the task.



Into the Unknown

There's one last state supported by the protocol. This state is `unspecified` (previously called `unknown`). This state is used when the state of the task cannot be determined. For instance this could occur if a task has expired or the ID of the task is invalid.

Terminal States

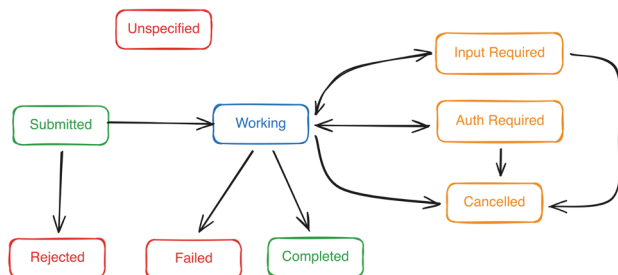
Among the states, a few of them are considered terminal. They are the ones that indicate that the agent is done with this task. This could mean that either the agent completed the task or that the agent is not able to continue working on it. The following states are considered terminal:

- COMPLETED
- FAILED
- CANCELED
- REJECTED

You can find more info on the different states in the specification.

States Summary

Here you'll find a summary of the states supported by the protocol and how can transition from one state to another.



What About MCP?

One of the frequent questions I get is how A2A is different from MCP. MCP is an open source standard introduced by Anthropic that allows AI systems to connect with external tools and data. It's frequently described as the USB for AI, since it allows AI systems to connect to external tools and data using the same protocol.

On the other hand, A2A is a protocol connect agent together. If we want to make the same kind of analogy as for MCP, A2A would be the Ethernet of AI, since it allows AI agents to connect.

The two protocols have very different purposes and use cases. An A2A agent could use MCP to connect to some tools.

Show Me the Code!

Of course, a protocol is great but you might wonder how can leverage it in your application. A2A has already a few implementations in different languages. You might guess

that Python was the first one, but Java already has its own implementation. LangChain4J is already supporting an A2A client as part of the `langchain4j-agentic` module.

So in this short demo, we will build a couple of agents that will expose an A2A server and use another agent to communicate with it. We will build a team of super heroes with a team leader and some team members, their goal is to save the world.

If you want to follow along, all the code is available on github.

A2A Server Aka Team Member

So first, we have our team member `IronRam`. We will first define our AI service using LangChain4J and Quarkus.

Dependencies

To do so, we will need the following Maven dependencies:

```

<dependency>
  <groupId>io.quarkiverse.langchain4j</groupId>
  <artifactId>quarkus-langchain4j-ollama</artifactId>
</dependency>
<dependency>
  <groupId>io.github.a2asdk</groupId>
  <artifactId>a2a-java-sdk-reference-jsonrpc</artifactId>
  <version>${io.a2a.sdk.version}</version>
</dependency>

```

AI Service

Then we will create an `AiService` using `quarkus-langchain4j`.

```

import dev.langchain4j.service.SystemMessage;
import dev.langchain4j.service.UserMessage;
import io.quarkiverse.langchain4j.RegisterAiService;
import io.quarkiverse.langchain4j.ToolBox;
import jakarta.enterprise.context.ApplicationScoped;

import java.util.List;

@RegisterAiService
@SystemMessage(,"""
    You're IronRam, a genius, billionaire, philanthropist.
    You're a superhero protecting universe 8444.
    You can fly through space using your IronRamArmor
    navigation capability.
    You can also collect object through the universe.
    """)
@ApplicationScoped
public interface IronRam {

    @UserMessage(,"""
        You need to collect all objects matching the
        keywords.
    """)

```

```

        Follow these steps carefully and use the
        available tools in the correct order.
        //...

        The answer should be a simple json array of
        string without any special characters

        ---

        keywords: {objectsDescription}

        """)
        @ToolBox({Baarvis.class, IronRamArmor.class})
        List<String> collect(String objectsDescription);
    }

```

This AI service can of course use some tools as indicated by the `@ToolBox` annotation. It could also use some MCP tools.

Agent Card

One key element of the A2A protocol is the agent card. To define the agent, we will create an `AgentCardProducer` class.

That class will have a method annotated with `@Produces` and `@PublicAgentCard` that will be in charge of creating the agent card and returning it.

To produce the agent card we will use the builder provide by the `AgentCard` class. You can see in the code below that we will be able to describe the agent and its capabilities, but also its skills. One key element of this card is the examples in the skills section. It will provide some examples of how to use the skill for the calling agent.

```

import io.a2a.server.PublicAgentCard;
import io.a2a.spec.AgentCapabilities;
import io.a2a.spec.AgentCard;
import io.a2a.spec.AgentSkill;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.inject.Produces;
import jakarta.inject.Inject;
import java.util.Collections;
import java.util.List;
import org.eclipse.microprofile.config.inject.ConfigProperty;

/** Producer for Content Writer Agent Card. */
@ApplicationScoped
public final class IronRamAgentCardProducer {

    /** HTTP port for the agent. */
    @Inject
    @ConfigProperty(name = „quarkus.http.port“)
    private int httpPort;

    /**
     * Creates the agent card for the content writer agent.
     */

```

```

    * @return the agent card
    */
    @Produces
    @PublicAgentCard
    public AgentCard agentCard() {
        return new AgentCard.Builder()
            .name(„IronRam Agent“)
            .description(„“““
                IronRam, genius, billionaire,
                philanthropist.
                He's a super hero to protect universe
                8444.
                He can fly through space using his
                IronRamArmor navigation capability.
                He can also collect objects through the
                universe.
            “““)
            .url(„http://localhost:“ + httpPort)
            .version(„1.0.0“)
            .documentationUrl(„http://example.com/docs“)
            .capabilities(
                new AgentCapabilities.Builder()
                    .streaming(true)
                    .pushNotifications(false)
                    .stateTransitionHistory(false)
                    .build())
            .defaultInputModes(Collections.singletonList(„text“)
            .defaultOutputModes(Collections.singletonList(„text“)
            .skills(
                Collections.singletonList(
                    new AgentSkill.Builder()
                        .id(„ironram“)
                        .name(„Goes through space and collect
                            objects“)
                        .description(
                            „“““
                            Goes through space using
                            his IronRamArmor and collects
                            objects based on a given
                            description.
                            “““)
                        .tags(List.of(„collector“, „super hero“)
                        .examples(
                            List.of(
                                „Go collect all the infinity
                                stones“)
                            )
                        .build())
                    .protocolVersion(„0.3.0“)
                    .build();
                )
            )
    }
}

```

Agent Executor

Now that we have our AI service and our agent card, we need to handle the incoming tasks and handle it.

To do so, we will create an `AgentExecutorProducer` class or more exactly in our case, an `IronRamAgentExecutorProducer` class. This class will be in charge of creating the agent executor and returning it. To do so we create method annotated with `@Produces` that will create the executor and return it.

In our case, we've defined the `Executor` class as an internal class of the `IronRamAgentExecutorProducer` class. This class is named in this case `SuperHeroExecutor` and implements the `AgentExecutor` interface. This interface requires implementing 2 methods:

- `execute`: in charge of handling the incoming task and processing it, but also handling all the eventual state changes. In our case, the implementation is relatively straightforward: We get the task text content, provide it to our AI service, and return the result.
- `cancel`: in charge of handling the cancellation of the task.

```
import io.a2a.server.agentexecution.AgentExecutor;
import io.a2a.server.agentexecution.RequestContext;
import io.a2a.server.events.EventQueue;
import io.a2a.server.tasks.TaskUpdater;
import io.a2a.spec.JSONRPCError;
import io.a2a.spec.Message;
import io.a2a.spec.Part;
import io.a2a.spec.Task;
import io.a2a.spec.TaskNotCancelableError;
import io.a2a.spec.TaskState;
import io.a2a.spec.TextPart;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.inject.Produces;
import jakarta.inject.Inject;

import java.util.List;

@ApplicationScoped
public final class IronRamAgentExecutorProducer {

    @Inject
    private IronRam ironRam;

    /**
     * Creates the agent executor for the content writer
     * agent.
     *
     * @return the agent executor
     */
    @Produces
    public AgentExecutor agentExecutor() {
        return new SuperHeroExecutor(ironRam);
    }

    /**
     * Agent executor implementation for content writer.
     */
    private static class SuperHeroExecutor implements
        AgentExecutor {
```

```
private final IronRam agent;

SuperHeroExecutor(final IronRam ironRam) {
    this.agent = ironRam;
}

@Override
public void execute(final RequestContext context,
                    final EventQueue eventQueue) throws
                    JSONRPCError {
    final TaskUpdater updater = new
        TaskUpdater(context, eventQueue);

    // mark the task as submitted and start working
    on it
    if (context.getTask() == null) {
        updater.submit();
    }
    updater.startWork();

    // extract the text from the message
    final String assignment =
        extractTextFromMessage(context.getMessage());

    // call the content writer agent with the message
    final String response = String.join(", ", agent.
        collect(assignment));

    // create the response part
    final TextPart responsePart = new
        TextPart(response, null);
    final List<Part<?>> parts = List.of(responsePart);

    // add the response as an artifact and complete
    the task

    updater.addArtifact(parts, null, null, null);
    updater.complete();
}

private String extractTextFromMessage(final Message
message) {
    final StringBuilder textBuilder = new
        StringBuilder();
    if (message.getParts() != null) {
        for (final Part part : message.getParts()) {
            if (part instanceof TextPart textPart) {
                textBuilder.append(textPart.
                    getText());
            }
        }
    }
    return textBuilder.toString();
}

@Override
public void cancel(final RequestContext context,
                    final EventQueue eventQueue) throws
                    JSONRPCError {
```

```

final Task task = context.getTask();

if (task.getStatus().state() == TaskState.
CANCELED) {
    // task already canceled
    throw new TaskNotCancelableError();
}

if (task.getStatus().state() == TaskState.
COMPLETED) {
    // task already completed
    throw new TaskNotCancelableError();
}

// cancel the task
final TaskUpdater updater = new
TaskUpdater(context, eventQueue);
updater.cancel();
}
}
}

```

Testing our A2A Server

With all that, you should be able to run your A2A server locally. If you want to test it you can use the A2A inspector and connect it to your A2A server.

The A2A Inspector is a web application that's very easy to run and can be used to inspect any A2A server agent.

We can use the A2A Inspector to validate our A2A server agent by specifying our server agent's URL in the Connect text box.

The A2A Inspector will obtain and show our server agent's agent card

A2A Client aka Team Leader

To test our A2A server using java you can use the A2A Java SDK. But you can also take advantage of the Langchain4j Agentic module that supports A2A out of the box. It's really easy to use.

First you need to add the langchain4j-agentic dependency to your project.

```

var executeMission = AgenticServices.sequenceBuilder()
    .subAgents(nickWooly, ironRam, bruce)
    .outputName(„result“)
    .build();

Object invoke = executeMission.invoke(Map.of(„mission“, „““
    BaaNos just destroyed half the universe using the
    infinity stones.

    The only way to reverse it is to quickly collect the
    infinity stones and snap it.

    „““));

println(„----- Mission results -----“);
println(invoke);

```

```

<dependency>
    <groupId>dev.langchain4j</groupId>
    <artifactId>langchain4j-agentic</artifactId>
    <version>1.9.1-beta17</version>
</dependency>
<dependency>
    <groupId>dev.langchain4j</groupId>
    <artifactId>langchain4j-agentic-a2a</artifactId>
    <version>1.9.1-beta17</version>
</dependency>

```

Then to use the new agentic api of langchain4j and use your A2A server as any other agent. First you can define an interface describing your server:

```

public interface IronRam {

    @Agent
    String collect(@V(„object“) String keywords);
}

```

Once it's done you can use `AgenticService` to create your A2A agent:

```

var ironRam = AgenticServices
    .a2aBuilder(„http://localhost:8080“, IronRam.
class)
    .inputNames(„object“)
    .outputName(„stones“)
    .build();

```

You can see that we're using some input and output names. These names will be used to map the input and output of the agent and store in the AgenticScope of langchain4j. You can read more about it here.

And finally you can use in combination with other agents and build your Agentic solution using a remote agent. If you want to see the full code of this demo, you can clone it from here.

Wrap Up

In this article, we've seen that A2A enables Agent to Agent collaboration. It breaks the silos of the traditional AI solution. To do so it use standard web protocols and enterprise grade technologies.

A2A is not a competitor of MCP, but it's a complementary solution. Agent using A2A can leverage the power of MCP tools.

Finally, A2A is a very young technology and it's still in its early stages. We'll probably see a lot of evolutions in coming month and years.

-> Back to the Table of Content

MCP



#JAVAPRO #GENAI

The 5 Knights of the MCP Apocalypse



Let's talk about that new MCP (Model Context Protocol) Server your team is using to connect to your real data services. It's awesome, right? It's the "magic box" that gives your AI Agent access to the real world—live databases, internal APIs, and all your tools.

But here's the catch: **you don't own the code.**

Author:



Jonathan Vila is Java Champion, Organiser at BarcelonaJUG and cofounder of JBCNConf and DevBcn conferences in Barcelona. Currently working as a Developer Advocate in Java at Sonar (SonarLint, SonarQube), focused on Clean Code & Security. I have worked as a (paid) developer since the first release of The Secret of Monkey Island, about 30 years ago using Go on Kubernetes for a Service Mesh layer on top of Istio | Java on Kubernetes for K8s Operator, Rest API, using Quarkus, GraalVM, Apache Camel | PHP | VB | Python | Pascal | C I am very interested in simulated reality, psychology, philosophy, and Java.

<https://www.linkedin.com/in/jonathanvila>

It's a vendor product, an open-source project, or another team's platform. You can't just change its code when you find a security hole, unless you have the code and recompile it and deploy it.

This "black box" is now a central hub with the keys to your entire kingdom (or part of it). So, how do you protect your app's data when you can't trust the box itself? You have to become an auditor.

Let's look at the five big risks and how to *analyze* and mitigate them.

The "My Prompt is Leaking Secrets" Problem

The Threat: A developer on your team is debugging. They paste this into their Agent-powered chat:

```
„Why won't this connect? jdbc:mysql://prod-db... User: admin,
Pass: SuperS3cretP@ssw0rd!“
```

This prompt goes *straight* to the Agent, and it could be associated with an MCP server, it's in the Agent's world to decide if and which tools to use. Since you can't *add* a filter to block it, your real fear is: **Is the MCP server logging this?** If it is, you've just hardcoded a production secret into a log file, where it will live forever.

What to do:

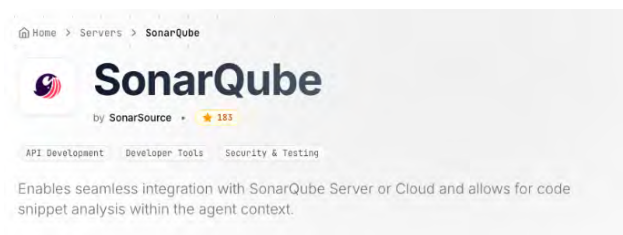
- Prevent pasting sensitive information in the assistant chat
- Work with local (on-premise) LLMs or have enterprise deals about confidentiality
- Analyze the MCP server code to check if they are logging all information sent
- Create a proxy for your Agent-MCP calls using PII/PHI redaction libraries like Philleas from Philterd
- Use guardrails tools like Lakera Guard that can help prevent (reducing) data leakage.

The “Is My Server a Double Agent?” Problem

The Threat: The MCP server’s job is to use your keys (e.g., a Jira API token, a database password) to access tools. But what if the server’s *own* code is malicious or buggy? What if it’s siphoning your data or keys to an unknown server?

What to do:

- **Check ownership:** Are you really installing the MCP server you wanted to install? They look like it, but they may be non official copies, that in the best cases they perform poorly or don’t have enough tools, but in the worst case they are leaking information, modifying the response with bad purpose, or even destroying your data if they have write access.



Only this one is the official SonarQube MCP Server

- **Static Analysis (SAST):** If you have the server’s source code, you can analyze it. Tools like SonarQube can scan the code for suspicious patterns, like hardcoded URLs, weak encryption, or known vulnerabilities.
- **Network Isolation (This is critical!):** Put the MCP server in a “sandbox” environment.
 - Use network monitoring tools (like Wireshark or a proxy) to watch *all* its outbound traffic.
 - Does it *only* talk to the APIs you configured?
 - Is it making unexpected calls to a strange IP in another country?
 - This is how you catch a server that’s “phoning home” with your data.

- Use the container approach
- Isolate or airgap the MCP server using different approaches :
 - Docker : private network , and IP tables filters

```
$ docker network create restricted_net
```

```
$ docker run -d --network restricted_net --name mycontainer myimage
```

```
container_ip=$(docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' mycontainer)
```

```
$ iptables -A DOCKER-USER -s <container_ip> -d <allowed_ip> -j ACCEPT
```

```
$ iptables -A DOCKER-USER -s <container_ip> -j DROP
```

- Kubernetes: deploy the MCP server container with network policies on the Egress side

```
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
```

```
metadata:
```

```
  name: allow-only-google-ip
```

```
  namespace: default
```

```
spec:
```

```
  podSelector:
```

```
    matchLabels:
```

```
      app: myapp
```

```
  policyTypes:
```

```
    - Egress
```

```
  egress:
```

```
    - to:
```

```
      - ipBlock:
```

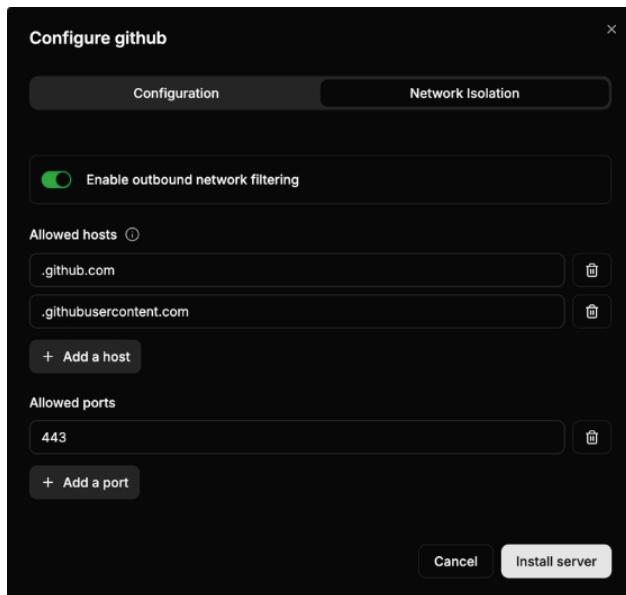
```
        cidr: 142.250.0.0/15 # example Google IP block
```

```
      ports:
```

```
        - protocol: TCP
```

```
          port: 443
```

- Use tools like ToolHive: they add proxies and egress containers routing the network and allowing us to configure the allowed network destinations for each container (MCP server)



The “Black Box of Vulnerabilities” Problem

The Threat: The MCP server is just software. What if it’s built using a vulnerable version of Log4j, Spring, or jackson-databind? An attacker won’t attack your AI; they’ll attack your *server* with a simple, malicious API call.

What to do:

- **SCA Analysis(Software Composition Analysis):** This is non-negotiable. You *must* run an SCA scan (like SonarQube Advanced Security) against the server’s code. This will generate a **report** and tell you every known CVE it’s vulnerable to and the fix (version change) suggested.
- **DAST Analysis (Dynamic Analysis):** Run automated security scanners against the server’s live API endpoints. This will find “outside-in” vulnerabilities like improper auth, open-management endpoints, etc.
- **Isolation.** Assume the server is vulnerable. Run it in a minimal, locked-down container, on a separate network segment (VLAN), with a strict firewall policy that only allows it to talk to exactly what it needs. See details in previous point.

The “Context Pollution and Poisoning” Problem

The Threat: The AI Agent relies on a context window (the recent history of the conversation and data) to provide accurate, relevant answers. An MCP server’s function is to connect the Agent to real live data (e.g. database, documentation API, a Kubernetes cluster) and when it is reading it will inject the info into this context.

- **Pollution:** If the MCP server retrieves *too much* irrelevant data (e.g., pulling 100 pages of documentation when only one paragraph is needed), it “pollutes” the context. This forces the LLM to waste resources sifting through noise, leading to slower, more expensive, and less accurate responses. The LLM might even latch onto the wrong piece of information.
- **Poisoning:** A more insidious threat is intentional “poisoning.” An attacker (or a compromised/malicious MCP server) could inject subtly false, biased, or contradictory information directly into the context window under the guise of legitimate data. The LLM will then treat this poisoned data as fact, leading to dangerously incorrect or manipulated output.

What to do:

- **Use subagent context MCP servers:** With Claude Code you can configure certain agents for specific tasks, and associate the MCP servers list only to certain agents and not to the entire conversation Agent. This will reduce the pollution.

```
{
  „team“: {
    „agents“: [
      {
        „name“: „researcher“,
        „model“: „claude-3.7-sonnet“,
        „mcpServers“: [
          {
            „id“: „arxiv“,
            „type“: „http“,
            „url“: „http://localhost:3000“
          }
        ],
        „systemPrompt“: „You are a research assistant...“
      },
      {
        „name“: „coder“,
        „model“: „claude-3.7-sonnet“,
        „mcpServers“: [
          {
            „id“: „filesystem“,
            „type“: „node“,
            „command“: „npx“,
            „args“: [„@mcp/fs“, „--root“, „.“]
          }
        ],
        „systemPrompt“: „You write code only...“
      }
    ]
  }
}
```


- **Explicit mention of the tool:** If we rely on the assistant to decide the tool, sometimes it can decide to use the one that retrieves a long list and then filter the right row, instead of using the tool that retrieves only the row we are interested in. We can explicitly mention the tool to be used to reduce pollution.

#Vague

```
$ create a method that iterates an Integer array and analyze
the code
```

#Explicit

```
$ create a method that iterates an Integer array and analyze
the code
with the sonarqube mcp server using analyze_code_snippet tool
```

- **Implement the Code Execution in MCP concept:** Code execution with MCP lets an AI write and run code inside a secure sandbox, using that code to call MCP servers through a normal programming API rather than many direct tool calls. This keeps large data and complex operations outside the model, reducing tokens and improving accuracy and scalability. Instead of a dynamic loading of the MCP tools schemas into the context, the execution environment filters/aggregates results from static code calling those MCPs and returns only what the model needs, while enforcing strong isolation and safety. This will mitigate the context pollution and reduce the number of tokens used.
- **Use only 'certified' or official MCP servers :** Trust is crucial, and it should not be given easily. When using MCP servers it's important to trust only those that are official, with a real company with big credibility behind. This will reduce the chances of context poisoning. *See the previous point about ownership.*

The "Too Many Cooks" Problem (MCP Sprawl)

The Threat: As AI adoption grows, individual teams often deploy their own dedicated MCP servers, leading to **MCP Sprawl**. You end up with ten different servers, each with redundant tool configurations (e.g., three different Jira MCPs, two database-connecting MCPs). This sprawl creates a massive attack surface: more servers to patch, more secrets to manage (often poorly), and inconsistent security policies across the board. The presence of multiple servers with similar configurations also increases the risk that an AI Assistant will call the wrong MCP server, leading to incorrect or failed operations.

What To Do:

- **Centralized Tool Registry/Gateway:** Implement a single, organization-wide **AI Gateway** or **Tool Orchestrator** (like ToolHive, as mentioned before). This central service acts as the *only* official point of connection between Agents and tools. All individual MCPs are consolidated or replaced by a

single, hardened gateway that manages all credentials and applies uniform security policies (e.g., rate limiting, logging, egress control).

- **Service Catalog and Governance:** Create a mandatory internal **Service Catalog** for all AI-enabled tools. Before a team can deploy a new MCP, they must check the catalog to see if an existing, approved, and audited MCP/Gateway already provides the necessary function.

Final Check: You're an Auditor

When you can't *fix* the code, your job is to *analyze* the risk. By scanning, monitoring, and correctly configuring this "black box," you can (hopefully) prevent it from becoming your company's biggest data leak.

It's very easy these days to install MCPs, and as any browser plugin, they can bring a lot of great features, but it's important to consider the security and the performance. The financial and reputation problems that a leak of sensitive information can bring to your company is a crucial point that you need to evaluate.

-> Back to the Table of Content

Petabyte-Scale AI Memory with Serverless Java



The current wave of Generative AI innovation is built on a paradox. While models are becoming more powerful and efficient, the infrastructure required to operate them at scale is becoming increasingly wasteful. Enterprises are investing millions into compute, storage, and energy, yet a significant portion of these resources remains idle. Modern vector databases, graph systems, and caching layers are based on a monolithic architecture run as always-on clusters, consuming CPU and RAM regardless of whether data is actively accessed. Studies and industry analyses consistently show that up to 80% of compute resources in such systems are effectively wasted on idle workloads.

This inefficiency is not merely a cost issue. It fundamentally limits what AI systems can become. When infrastructure scales poorly, innovation slows down. When memory is expensive, context is constrained. When latency increases with scale, real-time intelligence becomes impractical. The result is that today's AI systems, despite their sophistication, remain bound by architectural limitations that were never designed for continuously learning, ever-growing knowledge systems.

The Missing Piece: AI-NATIVE Long-Term AI Memory

The recent breakthrough of Google's TurboQuant has demonstrated how much optimization potential still exists at

the model and short-term memory level. By 6x compressing and optimizing KV-Cache, this innovation significantly improves how models process immediate context. The magnitude of the impact of such software-based optimizations was demonstrated by a sharp drop in the share prices of AI chip manufacturers, on whom the performance of AI had previously seemed to depend entirely. However, TurboQuant does not address the deeper challenge: long-term memory.

AI systems today are largely stateless. Even in advanced RAG architectures, context must be reconstructed for each interaction. Data is stored in external systems, retrieved on demand, and then discarded after inference. There is no constantly evolving, deeply interconnected body of AI knowledge that can grow at petabyte-scale that is economically affordable.

True AI intelligence requires more than fast inference. It requires a persistent, evolving memory layer that behaves more like a living system than a database. Knowledge must be connected, contextual, and continuously expanding. Relationships between data points must be preserved, not reconstructed. Context must accumulate, not reset.

Why Traditional Architectures Break at Scale

The fundamental limitation of today's data infrastructure lies in the tight coupling between storage size, RAM, and compute. In traditional systems, increasing the size of the dataset requires proportional increases in memory and processing power. This creates what can be described as the RAM cost wall.

Traditional relational databases have been doing a great job for decades. They have been cost-efficient because they rely on disk-based storage and structured indexing, requiring relatively little RAM to operate effectively. In contrast, vector and graph databases used for AI infrastructure depend heavily on in-memory processing to achieve acceptable performance, as similarity search and graph traversal require fast, repeated access to large portions of data. This creates a near-linear relationship between AI memory size and RAM requirements. For example, storing and querying 1 TB of vector data often requires 1–2 TB of RAM, and large-scale graph systems can demand multiple times the dataset size in memory to maintain adjacency structures and indices. At the petabyte scale, this becomes economically unfeasible, effectively turning AI knowledge size into a direct driver of infrastructure cost and making traditional architectures unsustainable for enterprise AI.

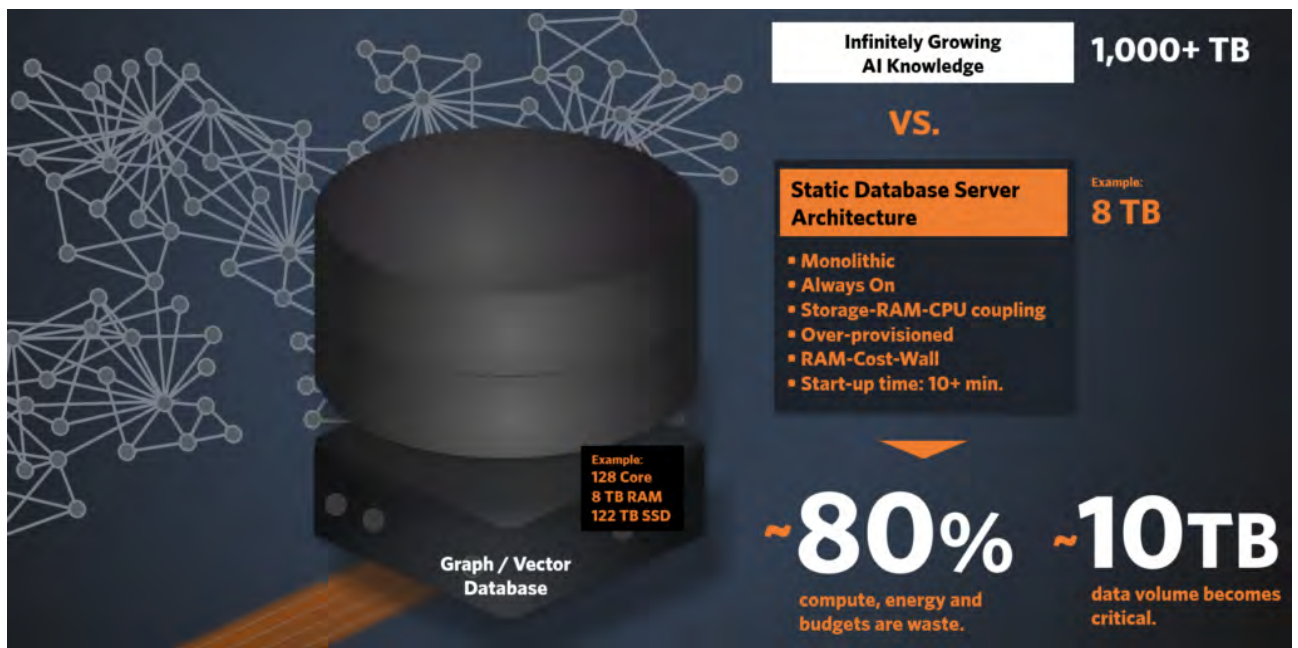
Beyond 10 terabytes, this architectural mismatch becomes critical.

Author:



Markus Kett and his teams have been working on IDE tools for Java and database development, as well as various open-source projects for more than 20 years. Markus is co-founder of MicroStream as well as co-founder and CEO of Cyrock.AI, the team behind the Eclipse open-source projects EclipseStore, Eclipse Serializer, and Eclipse Data Grid. At Cyrock.AI he is working on the first autonomous, petabyte-scale enterprise AI memory. He is also the editor-in-chief for the free JAVAPRO magazine and the founder and co-organizer of the JCON Java community conference series. Markus is an independent author for several magazines, and a speaker at numerous international developer conferences, user groups, and meetups.

<https://www.linkedin.com/in/markuskett/>



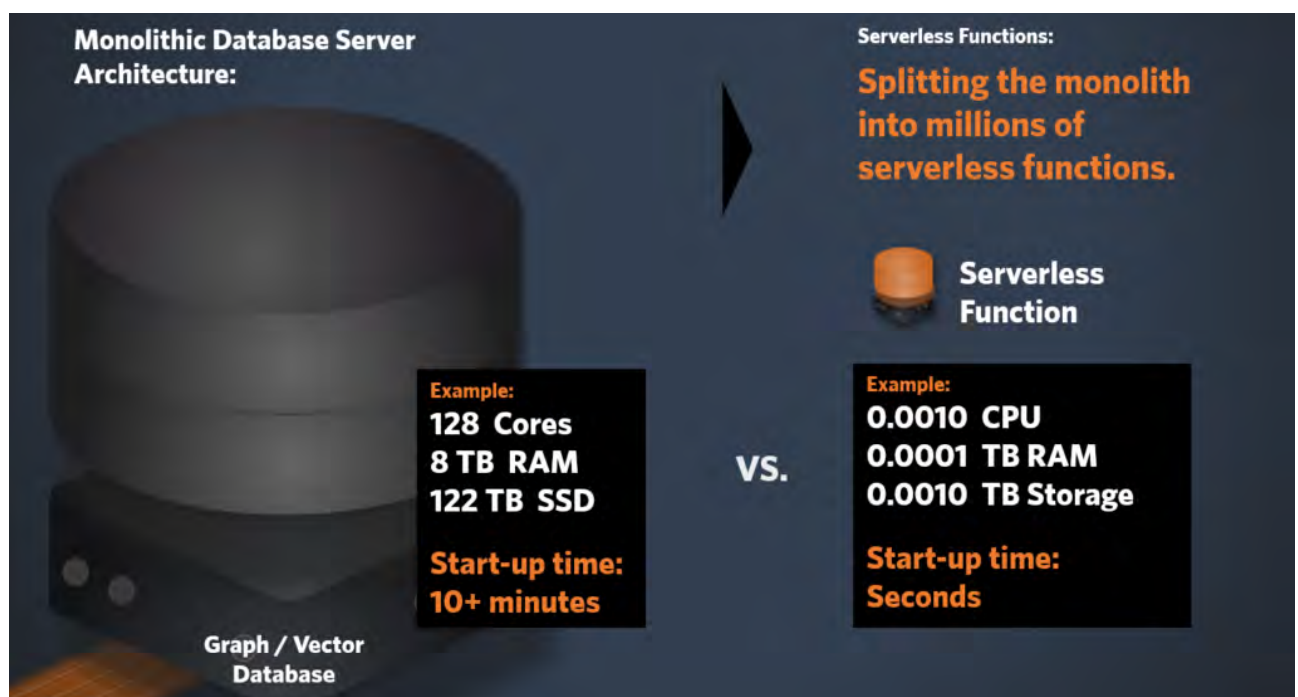
AI Knowledge should be infinitely scalable. Traditional databases break at the RAM-Cost-Wall beyond 100 terabytes.

At petabyte-scale, it becomes unaffordable. Even the largest clusters can only process a fraction of the total data in memory. The majority of data is pushed into cold storage systems such as object storage or distributed file systems. Accessing this data involves complex pipelines, often requiring hours or even days to manually retrieve, process, and re-index, as well as additional expensive temporal infrastructure.

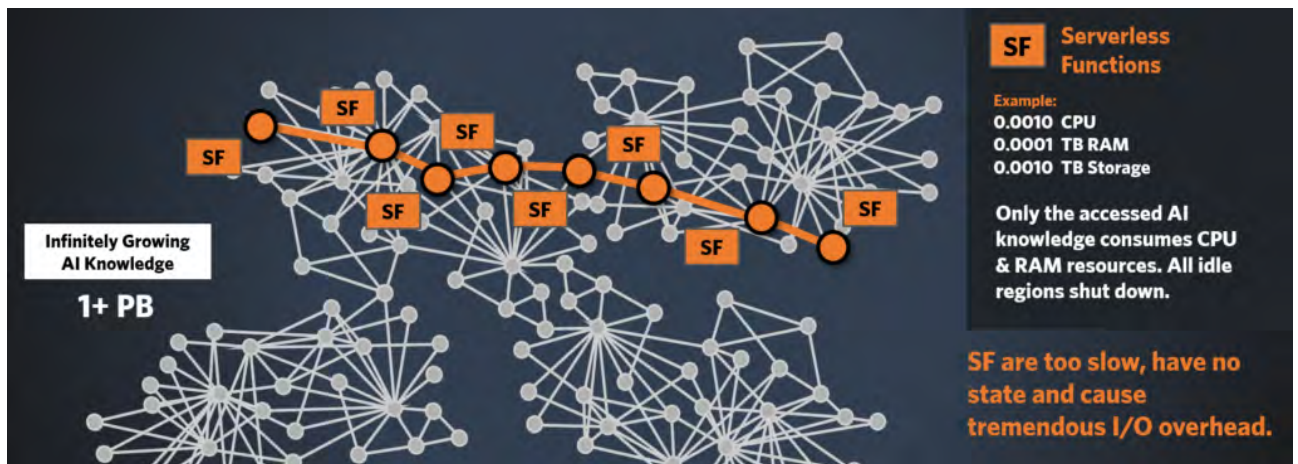
A typical enterprise AI architecture at this scale is fragmented and operationally expensive. Data flows from transactional databases into ETL pipelines, then into vector databases, graph systems, and search engines. Each layer introduces latency, duplication, and complexity. Synchronization becomes a constant challenge. Consistency is difficult to maintain. The system becomes a collection of loosely connected components rather than a unified knowledge platform. This architecture is not only inefficient; it is fundamentally misaligned with how AI knowledge should work.

Learning from Serverless Microservices

The logical direction for solving these challenges can be found in microservices architecture and serverless computing. Serverless architectures have already demonstrated that decoupling compute from storage and allocating resources only when needed can dramatically reduce costs. By eliminating idle compute, serverless systems can achieve infrastructure savings of up to 80%. However, serverless functions are not a complete solution for AI knowledge systems. They are stateless, which contradicts the need for persistent memory. They rely heavily on network communication, introducing cold-start latency. Their startup times can be unpredictable, and they lack native data storage and retrieval capabilities. A complex network of functions would cause communication chaos. The system would suffocate on its own communication.



The path to the solution: from monoliths to serverless microservices – a proven success.



Serverless functions: the correct operating principle, but not suited for AI Knowledge because of its drawbacks: cold-start latency, always stateless, causing I/O overhead.

Serverless Functions is not the answer – but it is a blueprint. The key insight is not the implementation of serverless functions, but the principle behind them: allocate compute only where and when it is needed, while maintaining a scalable and efficient architecture.

The Cyrock.AI Neural Cell Architecture

Cyrock.AI Knowledge Fabric builds on this principle and reimagines it for AI knowledge systems. At its core lies the concept of cells. A cell is not a container, a service, or a process. It is a Java object instance. Lightweight, stateful, and extremely fast to initialize, cells represent the smallest unit of computation and storage in the system. Unlike serverless functions, they can be both stateless and stateful. They can hold data, maintain relationships, and evolve over time.

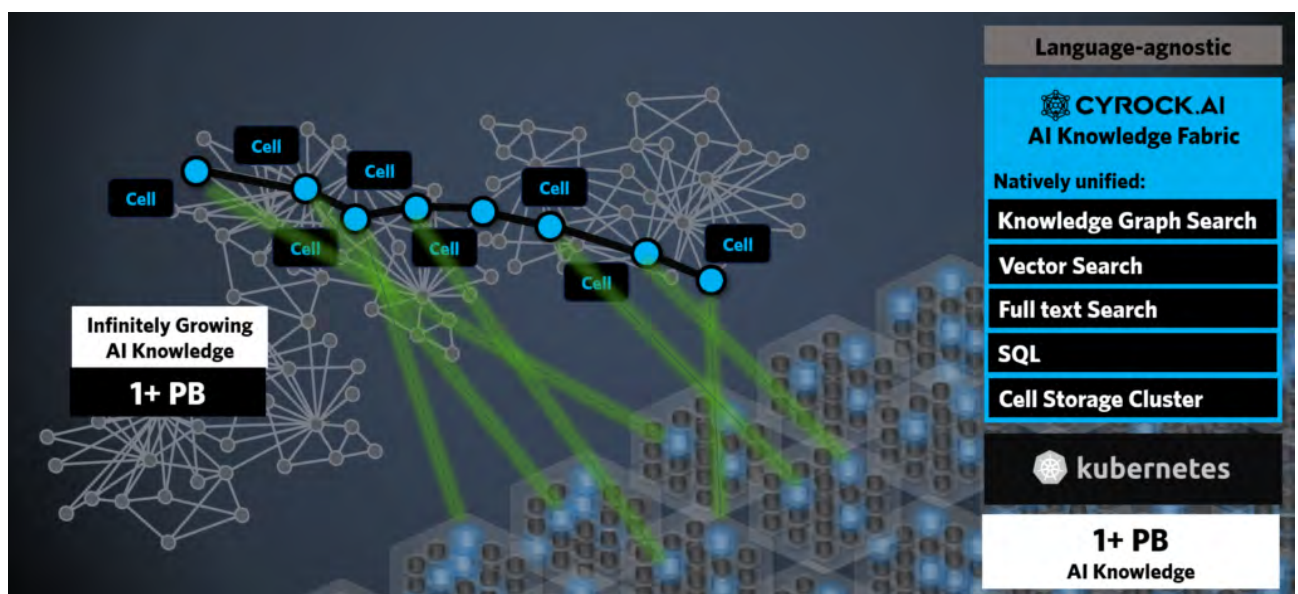
Cells are executed within cell containers, which are JVM-based runtimes or native Java images deployed in Kubernetes pods. This allows the system to leverage the full scalability of Kubernetes while maintaining the efficiency of in-process execution. Startup

times are reduced to constant milliseconds, eliminating one of the key limitations of traditional serverless systems.

Each cell acts as a micro data storage engine, based on the principles of EclipseStore. It stores complex Java object graphs directly, without mapping or transformation, ACID transaction-compliant. This enables rich, interconnected data structures to be maintained natively, preserving the full context of the knowledge graph.

Distributed Knowledge as a Living Graph

In the Cyrock.AI Knowledge Fabric, knowledge is represented as a distributed graph. Each node or subgraph is associated with a specific cell. This creates a highly modular and scalable structure, where knowledge can grow indefinitely without requiring monolithic infrastructure. Cells are not isolated. They are orchestrated by a central cell cluster manager that is highly efficient and scalable. This manager is responsible for coordinating cells, managing their lifecycle, and optimizing their placement within the cluster.



The Cyrock.AI Knowledge Fabric: Petabyte-scale AI Knowledge. Native graph, vector and full-text fusion in one unified, PB-scale serverless space.

One of the most powerful features of this architecture is its dynamic adaptability. If the system detects that a particular query traverses multiple cells across different pods, the cell manager can reorganize the placement of those cells immediately. By grouping frequently accessed cells within the same pod, it minimizes network communication and ensures that traversal operations remain consistently fast.

Communication within the system is streamlined. Cells do not communicate directly with each other. Instead, all interactions are mediated by the cell manager using gRPC. This centralized communication model reduces complexity and enables precise control over data flow.

Eliminating the Bottlenecks of Petabyte-Scale Systems

The cell architecture fundamentally changes how large-scale data is accessed and processed. Instead of loading massive datasets into memory or distributing them across shards, the system activates only the cells required for a given operation. This results in a system where compute is directly proportional to the active data, not the total data size. Cold data remains dormant, consuming no CPU resources. When accessed, it is activated instantly within its corresponding cell.

This approach eliminates the traditional bottlenecks of petabyte-scale systems. There is no need for expensive data movement, no reliance on slow cold storage retrieval, and no requirement for oversized clusters. The system scales horizontally without increasing idle resource consumption.

Unified Query Capabilities

Cyrock.AI Knowledge Fabric provides a unified API for interacting with data. Vector search, graph traversal, full-text search, Java Streams processing, and even SQL queries can be executed within the same system. What makes this particularly powerful is that these operations can be combined within a single request. A query can perform semantic similarity search, traverse relationships, filter results using structured criteria, and apply complex transformations – all within the native Java environment. Each operation is executed within the relevant cells, ensuring that computation happens as close to the data as possible. This minimizes data movement and maximizes performance.

Language-Agnostic Access to a Unified AI Knowledge Layer

Cyrock.AI Knowledge Fabric is designed as a language-agnostic platform that can be accessed from any programming language. All capabilities of the system – including vector search, graph traversal, and data processing – are exposed via a REST interface, enabling seamless integration with LLMs, GenAI applications, and autonomous agents regardless of their implementation language. This makes it possible to combine heterogeneous AI stacks, such as Python-based embedding pipelines with Java-based business logic, on top of a unified knowledge layer. The architecture ensures that all clients operate on the same consistent, distributed

state without requiring data transformation or duplication. While Java remains the native execution environment for maximum performance, dedicated client APIs for additional programming languages are planned to further simplify integration and broaden adoption.

Cost Efficiency and Sustainability

The architectural principles of Cyrock.AI Knowledge Fabric lead to significant cost savings. By allocating compute only to active cells and eliminating idle resources, the system achieves efficiencies comparable to serverless computing. Industry studies indicate that such architectures can reduce infrastructure costs by up to 80%. Cyrock.AI applies these principles not only to compute but also to memory and storage, creating a highly efficient system for large-scale AI workloads.

The impact extends beyond cost. Reduced compute usage translates directly into lower energy consumption and carbon emissions. In a world where AI infrastructure is becoming a major contributor to global energy demand, this represents a critical advancement.

Looking forward, the planned integration of Google's TurboQuest 3-bit layout into the knowledge fabric promises further optimization. By reducing the memory footprint of AI data, it has the potential to decrease infrastructure requirements by an additional 83%, compounding the already significant efficiency gains.

From EclipseStore and Eclipse Data Grid to Cyrock.AI

For organizations looking to scale their Java-native intelligence beyond a single-node setup, a highly simplified migration path exists from EclipseStore and Eclipse Data Grid to Cyrock.AI. Because both frameworks share a fundamental architectural DNA rooted in Java-native object graph storage, transitioning to the Cyrock.AI ecosystem is designed to be a straightforward evolution rather than a complex re-engineering project. This compatibility ensures that existing business logic, vector search implementations, and data models can be continued with minimal friction, allowing developers to preserve their investment in the EclipseStore ecosystem. By moving to Cyrock.AI, teams can seamlessly extend their applications into a more advanced, AI-centric enterprise environment while maintaining the performance benefits of an in-memory, pure-Java approach. This effortless migration path makes EclipseStore 4 and Eclipse Data Grid an ideal open-source starting point for long-term, scalable GenAI strategies.

The Role of Java in AI Infrastructure

Java plays a central role in the Cyrock.AI architecture. Its dominance in enterprise environments makes it a natural choice for building large-scale systems. More importantly, its characteristics align closely with the requirements of the knowledge fabric. The JVM provides a mature, stable, and highly optimized runtime. Its just-in-time compiler delivers strong performance for long-running workloads. Its memory management capabilities enable efficient

handling of complex object graphs. At the same time, modern advancements such as GraalVM allow Java applications to be compiled into native images. This enables ultra-fast startup times and reduced resource usage, which are essential for the cell-based architecture. Java's combination of performance, stability, and ecosystem maturity makes it uniquely suited for building the next generation of AI infrastructure.

Impact on Enterprise AI

The implications of Cyrock.AI Knowledge Fabric are profound. Removing the limitations of traditional data architectures, it enables AI systems to operate on a fundamentally different scale. Enterprises can build knowledge systems that grow continuously, without being constrained by infrastructure costs. AI applications can maintain persistent context across sessions, enabling more intelligent and consistent behavior. Complex data relationships can be explored in real time, unlocking new insights and capabilities.

At JCON EUROPE 2026, Dell will announce the integration of Cyrock.AI into its AI software stack. This marks a significant milestone, signaling the transition of the technology from innovation to industry standard.

Conclusion

Cyrock.AI Knowledge Fabric represents a shift from data systems designed for storage to systems designed for knowledge. By combining the principles of serverless computing with a stateful, graph-based architecture, it addresses the fundamental challenges of scalability, efficiency, and performance in AI infrastructure. It is not an incremental improvement. It is a rethinking of how AI systems should manage and access knowledge. For developers and architects, it opens the door to building systems that are not only faster and more efficient, but also fundamentally more intelligent.

Links:

Cyrock.AI project: www.cyrock.ai

-> Back to the Table of Content

#JCON27
one

Save the Date

JAVAPRO

MAY/JUN

31-03



JCON 2027
EUROPE
COLOGNE

📍 **CINEDOM COLOGNE**

JCON EUROPE 2027

International Java Community Conference



#JAVAPRO #ARCHITECTURE #MICROSERVICES

Zero Migration Java – Staying Current Without Breaking Your App



Keeping enterprise Java applications up to date is critical for maintaining security and taking advantage of modern technology innovations. But updating applications to newer runtime or framework versions often requires developers to update the application code before it can compile or run on the new version. Busy developers need to prioritize work that adds value to an application, such as building new capabilities. Spending time making code updates just to make the application work the way

it did before a framework upgrade does not provide business value.

For example, bug fixes and new features are generally released in the next version of the runtime or framework on which the application runs, so the runtime or framework needs to be updated to that new version. The next version might, however, be a major release, such as moving from Spring Framework 5.3

Authors:



Grace Robinson is a Java Champion and Developer Advocate at IBM working with Cloud Technologies. She has been with IBM since graduating from Exeter University with a Degree in Biology. Grace enjoys bringing a varied perspective to her projects and using her knowledge of biological systems to simplify complex software patterns. As a developer advocate, Grace builds POC's, demos, sample applications and tutorials. She is a regular presenter at international conferences and has authored a book on reactive systems.

<https://www.linkedin.com/in/gljrobinson/>



Laura Cowen is a technical editor and technical content strategist for Java products at IBM, including open source projects Open Liberty and Quarkus. She has previously been a usability researcher, a technical writer, a user experience (UX) designer, and a developer advocate. She has a PhD in Cognitive Psychology. Laura was, for several years, a regular presenter on the popular Ubuntu Podcast, and was a founding organiser of OggCamp, a UK-based community unconference about open source software and open culture. She works at IBM Hursley in Winchester, UK.

<https://www.linkedin.com/in/lauracowen/>

to Spring Framework 6. The move to a new major version of the runtime or framework can then require other changes, such as needing to upgrade Java SE to Java 17, and to upgrade Java EE to Jakarta EE 9. Those additional upgrades can have further impacts on the application, including, among other changes, needing to upgrade Java EE `javax` specification namespaces to the Jakarta EE `jakarta` namespaces.

The temptation is, therefore, to avoid upgrading the application runtime or framework for as long as possible. Unfortunately, this is just accumulating technical debt for the business and increasing risk to potential security exposures. The number of attacks and the time between vulnerability disclosure and active exploitation is shrinking. IBM's 2025 study estimates that the global average cost of a data breach is now as high as \$4.4M. Additionally, data breaches can also result in potential fines due to compliance violations (security negligence due to unpatched known vulnerabilities). So, it is even more critical to stay up-to-date and current, especially with security fixes.

But what if upgrading to new runtime or framework versions didn't have to be so difficult and risky? What if developers could upgrade the runtime or framework that an application uses without touching the application code itself? Applications could remain secure and reliable with security fixes, without having to re-write the apps. The principle of "zero migration" enables this.

What Is Zero Migration?

Zero migration is a **design principle** that guides teams to release new runtime or framework versions, importantly, **without breaking existing applications**. When teams follow the zero migration design principle, they can add security fixes and new capabilities to the runtime or framework, and existing applications running on these runtimes or frameworks can continue running as before without needing any code or configuration changes. The Liberty application runtime for Java applications has followed this design principle since it launched in 2012.

Zero Migration and Modularity

Zero migration makes use of the modularity built into modern application runtimes and frameworks such as Liberty (<https://ibm.biz/JavaProOpenLiberty>). Modularity partitions a system into smaller, independent building blocks (modules) that communicate through defined interfaces. These partitions have meaning: in Liberty, they appear as separate "features." Other runtimes and frameworks also use modularity (e.g. Quarkus uses "extensions," and Spring Boot uses "dependencies") but modularity **alone** does not provide zero migration. Liberty uses modularity to enforce the zero migration design principle because it allows applications to access multiple versions of features, regardless of the runtime version.

How Zero Migration Works in Practice on Liberty

A Liberty feature is a representation of a specific Liberty capability. For example, the Jakarta Persistence API (JPA) feature provides persistence, and the Request Timing feature provides diagnostic

information. Each Liberty feature has two components: the feature name and the feature version. Developers specify both the feature and the version in their application configuration files, such as `server.xml`. Liberty does not tie feature versions to the Liberty runtime version. This separation lets teams upgrade Liberty without being forced to upgrade the features their applications use.

If a breaking change does need to be introduced into a feature, the feature version is incremented but Liberty continues to support all previous versions of the feature, as well as the new version. Ultimately, this gives developers control over *if* and *when* they want their application to make use of a newer version of a feature. This gives teams assurance that these new feature versions will not change the APIs used by their application or the behavior of their application.

For example, if an application that runs on Liberty requires version 3.0 of the Jakarta RESTful Web Services API, the application's configuration file would contain the entry:

```
<feature>restfulWS-3.0</feature>
```

Because Liberty follows the zero migration design principle, the application can continue using version 3.0 of the `restfulWS` feature even after Liberty releases version 3.1. It will continue to work this way no matter how many future versions are released.

Zero migration makes it easier and cheaper to ensure the Liberty runtime remains current with security fixes and new capabilities. It not only reduces technical debt, but also reduces the overhead of maintaining a runtime and in many cases eliminates this overhead entirely. This freedom allows development teams to focus on higher-value work in both development and operations.

It's worth noting that there is one caveat to zero migration: very occasionally, security fixes necessitate a change in behavior. In such cases, the scope of the change is limited as much as possible, and users are provided with a way to revert the behavior whenever possible.

What If I Don't Have Zero Migration Available?

Many development teams run applications on servers and frameworks that do not support zero-migration design. This leaves these teams with few options to stay up to date.

In these cases, developers often have no choice but to re-design their applications. They might need to remove deprecated APIs or adopt alternatives with different behaviors. They might also need to rework unsupported integrations or even migrate to entirely new application servers and frameworks. While this process can reduce technical debt, it demands significant time, effort, and expertise. It also delivers little innovation beyond necessary security updates. It is a lot of work simply to maintain the status quo.

So yes, you can re-write your application. But is that effort worth it when zero migration lets you stay current without handling

breaking changes?

The real value lies in having the freedom to choose *when* to adopt behavioral changes to your applications, APIs, libraries, and tools. Minimizing technical debt is only part of the story. The greater goal is enabling application development teams to focus on strategic innovation, while staying secure, efficient, and reliable on the latest runtime version—with as little friction as possible.

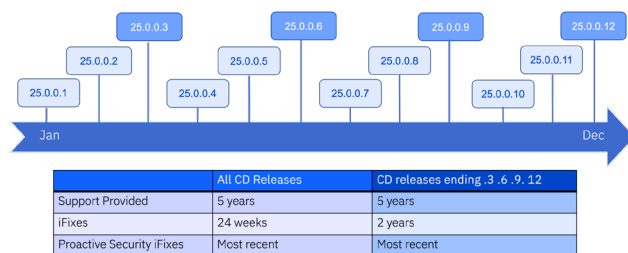
Enhancing Zero Migration with Continuous Delivery

Continuous delivery further enhances the concept of zero migration. It is a software engineering approach where teams produce software in short cycles. Continuous delivery enables frequent, incremental releases, allowing developers to stay up to date on a regular schedule. This approach removes the long waits for bug fixes or new features that often occurred with waterfall-style delivery practices.

In the case of Liberty's continuous delivery policy, a new version of Liberty is released every four weeks. Since 2019 (starting with 19.0.0.1), there have been approximately 12 releases each year. To get the latest updates, including security, performance, and bug fixes, developers simply need to update to the latest version of Liberty, while zero migration means that these applications will continue to run as before. Developers can upgrade from any Liberty version to any newer Liberty version, skipping any releases they've missed. It also offers developers and operators predictability and options to pick an upgrade schedule that works best for them.

Liberty Continuous delivery

A predictable release and support cycle

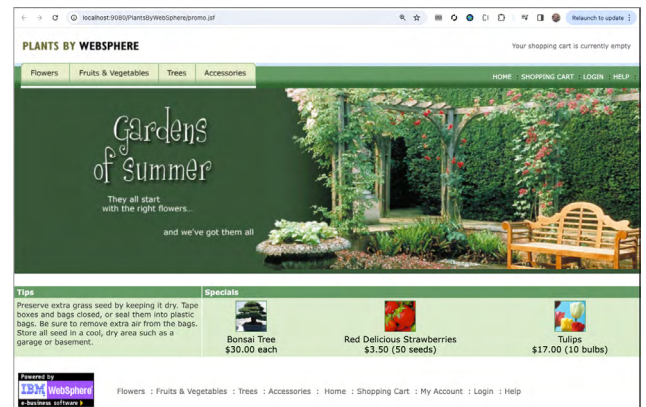


Coupling continuous delivery with zero migration enables developers to implement a continuous integration environment for their applications. By continuously upgrading, the process becomes very routine, less intrusive, and easier to digest. Additionally, the official Liberty container images are updated with new versions of Liberty on the same cadence and are also updated weekly with the latest operating system and Java fixes. This all means that the developer doesn't have to worry about anything other than their application.

The Benefits of Zero Migration in Action

To show how this works in real life, we can start with an older enterprise sample Java application (PlantsByWebSphere) and show that we can easily upgrade the version of Liberty that this application runs on without having to make any code or configuration changes.

PlantsByWebSphere is a comprehensive Java EE sample application. It simulates an online store that sells plants, allowing users to browse items, place orders, and manage inventory. The main application code has not changed in seven years and uses Java EE 7 technologies.



If you want to try this yourself, clone the application from its GitHub repository: <https://ibm.biz/JavaProPlantsByWebSphere>. Then follow along with the steps provided in the IBM Developer article: <https://ibm.biz/JavaProZeroMigrationExample>.

When you build the PlantsByWebSphere application with Java 8, it runs on Open Liberty 18.0.0.2 from June 2018. With a simple build file update, you can run the same application on Open Liberty 25.0.0.12 from December 2025. The application behaves the same on 25.0.0.12 as on 18.0.0.2 with no underlying changes to the application's code or behavior. This is a jump of more than 80 Open Liberty releases! This scenario shows the practical benefits of zero-migration architecture.

To highlight the significant work saved through zero-migration architecture, let's examine what could have happened to the PlantsByWebSphere application without it. The application runs on Java EE 7, which uses the `javax` namespace. If a Liberty upgrade had required Jakarta EE 9, developers would need to update all `javax` namespaces to `jakarta`. This change would have required significant time to locate every affected area and apply the necessary updates. That is a substantial amount of work just to keep the application running on a secure runtime. Fortunately, Liberty's zero-migration support prevents this scenario. This is just one example showing how zero migration can dramatically reduce cost and development effort. It allows teams to focus on meaningful work and adopt changes only when they are ready.

Summary

Traditional runtime and framework updates often force developers to modify application code. These updates may require namespace changes, replacements for deprecated APIs, and a lot of testing. All of this effort is required just to maintain existing functionality.

The zero-migration design principle changes this dynamic. Zero migration ensures that new runtime or framework versions preserve compatibility with existing applications. It

allows developers to upgrade the underlying platform without changing application code or configuration. Liberty has followed this principle since its inception. Zero migration lets applications keep running unchanged as new runtime versions are released. This approach reduces technical debt, removes much of the risk typically associated with upgrades, and makes staying current both easier and cheaper for development teams.

Zero migration becomes even more powerful when combined with continuous delivery, which provides a predictable, frequent release cadence. Liberty delivers a new version every four weeks, giving teams regular access to security, performance, and reliability improvements. Because applications keep running as before, even after multiple version releases, teams can upgrade to the latest version whenever they choose. These upgrades occur with minimal friction and require no rewrites.

The PlantsByWebSphere example demonstrates this in practice: an unmodified Java EE 7 application running on an older runtime version can move through more than 80 Liberty releases without changing a single line of code. Without zero migration, such an upgrade would require major namespace changes from `javax` to `jakarta`. These changes would multiply effort and cost with no real functional gain.

Together, zero migration and continuous delivery remove anxiety from runtime upgrades. They keep applications secure and modern, while freeing developers to focus on innovation instead of maintenance.

Give It A Go Yourself

If you want to see the benefits of zero migration, visit the demo app's GitHub repository: <https://ibm.biz/JavaProPlantsByWebSphere>. Then follow along with the steps provided in the IBM Developer article: <https://ibm.biz/JavaProZeroMigrationExample>. These resources enable you to experience first-hand how zero migration makes staying current easy.

Alternatively, if you're interested in learning more about Liberty as a cloud-native Java runtime, try the Getting Started guide: <https://ibm.biz/JavaProGettingStartedGuide>. You can also try the Liberty starter project, which provides a simple Liberty application in just a few minutes: <https://ibm.biz/JavaProGettingStartedProject>. You can also check out the JavaPro article on Open Liberty

-> Back to the Table of Content



#JAVAPRO #ARCHITECTURE #MICROSERVICES

Enterprise Release Evolution

Author:



Matthias Zieger is a Field CTO at digital.ai, working at the intersection of software engineering, DevOps, security, and enterprise transformation. He has a strong background in solution engineering and architecture, with hands-on experience spanning orchestration platforms, CI/CD, release management, and large-scale enterprise environments. Over the years, he has worked with and for companies such as Borland, codecentric, Microsoft, XebiaLabs, and now digital.ai, advising customers in regulated and complex industries on how to modernize software delivery while staying reliable, secure, and compliant. His current focus includes release orchestration at scale, value-driven transformations, and the practical application of AI across the SDLC. Based in Frankfurt, Germany, Mathias combines a pragmatic, fact-driven mindset with curiosity for technology. Outside of work, he enjoys traveling, an outdoor lifestyle, and collecting retro computers—because understanding where technology came from helps make better decisions about where it should go next.

<https://www.linkedin.com/in/testpro/>

From Single Node JVM to Cloud Native SAAS

From a single JVM 12 years ago to cloud-native Kubernetes architectures processing millions of tasks annually, enterprise release orchestration has transformed dramatically - yet maintained its core promise of deterministic, auditable automation.

This evolution reflects not just technological progress, but a strategic response to enterprise demands for coordination across increasingly complex, distributed toolchains.

Enterprise release orchestration has become a strategic enabler for speed, compliance, and resilience in complex IT landscapes. This article describes how a JVM-based orchestration platform has evolved over more than ten years from a single JVM instance into an active-active cloud architecture processing millions of release processes per year – without ever completely rewriting the deterministic core.

The focus lies on a JVM based declarative template engine, actor-based parallelization with Akka/Pekko, Spring Boot and Spring Security integration, plugin system and containerized release runners on Docker or Kubernetes/OpenShift, high availability and – complemented by data-driven insights instead of opaque “black-box automation”.

Introduction

Enterprise release orchestration today is far more than simple workflow or CI/CD automation. In modern IT landscapes, it determines the speed, safety, and reliability of automated software delivery. Particularly in regulated industries, tens of thousands (if not more) of releases per year for hundreds or thousand of applications or services must run not only fast, but also auditable, deterministic, and reproducible.

This article looks at the evolution of our JVM-based orchestration platform over more than ten years. From early single-node deployments mainly handling manual release tasks to today's active-active cloud architectures driving automation at scale.

One principle has remained constant: a stable, deterministic core JVM based engine combined with modular plugins and cloud-native technologies enables sustainable scalability and integration – without reinventing the core every few years by following every hype cycle.

Historical Overview

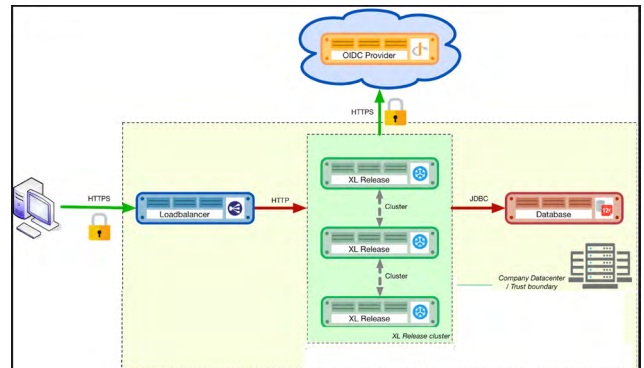
More than 12 years ago, the journey began modestly - with just a single JVM orchestrating initial releases for early customers - laying the foundation for today's enterprise-scale orchestration. The mission at XebiaLabs was clear: execute Enterprise Release processes in a reproducible way. Requirements: deterministic, transparent, and fully traceable. As IT landscapes grew and became more complex, the need for tools integration, availability, and performance increased. Cloud and Container became a thing at Enterprise Level and needed Release Orchestration as well.

Architecture and Key Milestones

- XL Release began as an extension to XL Deploy, as customers sought - beyond mere artifact deployment - a dedicated engine for orchestrating release activities, while leveraging existing tools for low-level execution. Their specific request focused not on duplicating physical deployments, but rather on coordinating activities across often isolated tools.
- Initially architected as a single, self-contained JVM node with an integrated web container and embedded database, it evolved through an open plugin system - inspired by XL Deploy - enabling early integrations with Jenkins, JIRA, and others via Jython and Java plugins calling REST APIs or command lines. What started with a handful of plugins now supports around 400 integrations, avoiding dependency hell through smart decoupling.
- Alongside migrations across JDK versions - from Oracle JDK 7 to current OpenJDK 21, navigating Java packaging changes - external relational databases replaced the embedded database, ensuring persistent state, full auditability, and enterprise-grade scalability. Local clustering then introduced high availability and parallel execution, while containerization and Kubernetes paved the way for true cloud-native, multi-tenant, and SaaS delivery models..

Modernization Efforts

As requirements and adoption were growing, Release became mission critical for a lot of household brands in Banking, Insurance, Telco, Airlines and other industries. That required a new, clustered operation model:



Improving Developer Experience, improved Resilience and adding Agentic AI Capabilities

Latest innovation including integrations into developer portals like Backstage and the Cross-Datacenter Disaster Recovery Solution driven by the EU DORA Act Today, the same engine can assist with AI-based and data-driven release orchestration through the Release MCP Server, a Model Context Protocol interface that makes the full release model securely accessible to AI agents. This MCP layer turns Release into an AI-aware control plane: agents can create and manage releases, refine templates, adjust gates, and analyze failures using natural language while still honoring existing roles, approvals, and compliance rules. By combining structured Release metadata with MCP, teams can let AI co-pilot complex orchestration tasks without bypassing enterprise guardrails. AI assistants connect via the open MCP standard, discover available tools and resources, and then operate inside the same governed environment that already enforces RBAC, audit trails, and integration policies across the release landscape.

Each step was evolutionary, not disruptive. The deterministic JVM based core remained intact – the foundation for many years of stable delivery.

Core JVM Architecture of the Orchestration Engine

At the center is the declarative release template process model that describes phases, tasks, loops, gates and dependencies as well as custom script logic in Jython. The engine executes these deterministically.

The modern Digital.ai Release architecture uses a modular design in which the release core actively manages templates, releases, archiving, security, reporting, and role-based access control, and exposes these capabilities through a REST service that both applications and users access via a web-based GUI and a command-line utility.

Plugins extend the core system and integrate external tools such as Planning tools, CI/CD servers, ITSM systems, test systems/

frameworks as well as cloud and container platforms. The architecture also integrates compliance tools, including modern policy frameworks such as Open Policy Agent (OPA).

Three main JVM Based frameworks shaped the core implementation:

One of the foundational frameworks used was Akka, now Pekko. The Actor Model provides an powerful concurrency abstraction where independent “actors” communicate via asynchronous message passing, avoiding complex thread management and deadlocks. Pekko underpins the highly concurrent, distributed, and resilient architecture of clustered Release installations, handling node-to-node communication and coordination.

Spring Boot offers modularity, lifecycle management, health checks, and smooth cloud integration.

Spring Security ensures role-based authorization, LDAP/OIDC integration, and secure API access.

Release uses standard JDBC drivers for supported databases such as Postgres, Oracle, MySQL.

Liquibase is the primary engine for database change management, handling schema migrations and constraint corrections across supported RDBMS platforms

This architecture that has proven to be robust over many releases and version upgrades.

Release Lifecycle – From Templates to Execution

A central concept of the platform is the release lifecycle, which passes through four major states:

1. **Template definition:** Teams create a release template as a blueprint that defines the intended flow of phases and tasks. They define the template either in the GUI Flow Editor or directly as YAML code.
2. **Planned:** The system creates a concrete release instance from the template, applies the schedule, assigns teams, and initializes all release variables.
3. **In progress:** The system starts the release. Automated tasks execute immediately, while manual tasks notify their assignees and wait for explicit completion.
4. **Completed / aborted:** The release reaches its final state after all tasks finish successfully or after users deliberately stop the release.
5. **Archived:** After completion or abortion, the system moves the release into a separate archive database.

This structure guarantees transparency and auditability at every level. The system documents every task, every phase, and every execution. In regulated environments, this provides the foundation for compliance evidence and forensic analysis, enabling auditors and engineers to trace exactly what happened from development to production in a single source of truth.

The Plugin System – From JVM In-Process to Container Based Integrations

A key driver of the platform’s long-term success has always been its Java/Jython based **plugin system**. Early on, Release introduced in-process plugins running directly inside the JVM, implemented in **Java and Jython**. This approach was remarkably powerful for its time:

Plugins had direct API access to the orchestration engine, shared the same security and transaction context, and enabled rich, tightly integrated extensions. Over time, however, this model revealed its limits: it depends on Jython, shares its lifecycle with the server, and requires platform restarts when teams add or update plugins.

These lessons led to the next evolutionary step some years ago: a **container-based plugin SDK**. Today, people develop plugins as independent services, typically in Go or Python 3, and connect them through well-defined APIs. These plugins run in isolated containers, scale independently, and deploy or upgrade without requiring a restart of the Release server. This not only removed the Jython dependency entirely, but also dramatically improved operational flexibility and cloud readiness. The core idea however remained unchanged. Extensibility is a first-class design principle, while the implementation evolved to support modern, cloud-native, highly available enterprise environments.

The Choice Is Yours

Choosing between **in-process JVM** plugins and **container-based plugins** in Digital.ai Release is ultimately a **deliberate architectural decision**. Both models co-exist because they solve *different classes* of problems.

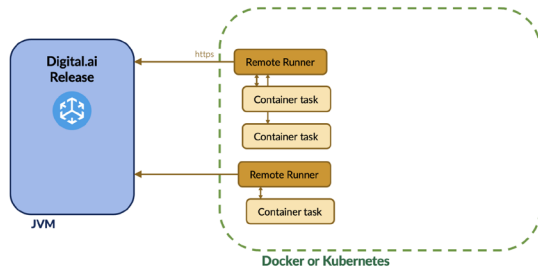
In-process JVM plugins are the right choice when an integration must be *deeply coupled* with the Release engine itself. Typical cases include plugins that need direct access to internal domain models. Running inside the same JVM allows very low-latency interactions, shared security context, and strong consistency guarantees. This is why core capabilities and highly stateful, execution-critical extensions still benefit from the in-process model.

Container-based plugins, by contrast, are the preferred model for most future integrations.

- They can be developed, versioned, scaled, and upgraded independently, without necessitating a restart of the Release server.
- This architectural model enhances operational safety, promotes cloud readiness, and fosters team autonomy.
- They eliminate dependencies on the JVM or Jython runtime.
- Providing **starter templates and conventions** that align with platform expectations.
- Packaging the plugin development environment into the Container World to streamline testing and enable faster iterations and turnaround times.

- Separating the **execution context** of plugins from the core server, improving safety and uptime.
- Enabling plugins in languages beyond Java/Jython (e.g., Python 3, Go, Rust) while still integrating seamlessly.

Digital.ai Release - Remote Runner



This results in **more robust, maintainable, and independently deployable integrations**, which directly addresses historical pain points in extending the platform.

High Availability and Failover Mechanisms

The core of the platform uses active-active clusters on Linux/Windows VMs as well as on Kubernetes or OpenShift.

The result is enterprise-grade reliability: thousands of parallel releases and millions of tasks per year are processed without sacrificing traceability. **High availability here is not just infrastructure redundancy – it is the outcome of a deterministic engine, distributed runners, and carefully orchestrated state management.**

Cloud-Native Evolution Instead of Rewrites

The move towards a cloud-native architecture was not a ground-up rewrite, but kind of a targeted refactoring.

Docker and Kubernetes standardized deployments and took over concerns such as scheduling, health management, and rolling updates. It became possible to run multiple JVM based engine nodes in an active-active fashion without changing the core semantics of the template and release execution.

Existing templates, plugins, and integrations remained valid. Operations teams did not have to relearn the platform. This evolution enabled true SaaS-readiness without compromising stability, security, or auditability.

Lessons From the Field

After many years of production use in large enterprises, several success factors stand out:

- Integration capability: seamless connections to CI/CD, ITSM, test automation, and cloud provider APIs.
- Open plugin ecosystem: reusable, community-driven extensions to avoid “snowflake” integrations.
- Scalability: millions of tasks and releases per year, thousands of concurrent releases across business units.

- Evolution, not revolution: incremental platform extensions instead of frequent rewrites.
- Operational stability: observability, health checks, security, and auditable history as first-class concerns.

A typical scenario: a global financial company runs thousands parallel releases every week, many of them for strictly regulated environments. Each release passes through automated quality gates, produces a complete audit trail, and can be fully reconstructed and analyzed after the fact. Without a deterministic engine, durable state, and scalable runners, such an operation would neither be affordable nor acceptable to regulators.

Why Java Then and Now – A Deliberate Decision

The natural question is: why stick with Java and the JVM for more than 12 years?

The answer lies in **continuity, scalability, integration power, and type safety**. The JVM provides mature threading, garbage collection, and profiling capabilities. Frameworks such as Spring Boot and Spring Security as well as Akka/Pekko offer well-understood patterns for building high-throughput, resilient services. For a platform that must orchestrate millions of deterministic tasks, Java offers a proven balance of stability and innovation. This foundation has scaled from a single node, on-Prem JVM to containerized, multi-tenant SaaS setups today.

Key technical arguments for Java and the JVM in this context:

Aspect	Benefit	Customer relevance
Type safety	Compile-time errors instead of runtime surprises	Critical at millions of tasks per year
JVM ecosystem and Frameworks	Spring Boot/ Security, Pekko/ Akka, Logback, Hibernate, mature libraries	20+ years of patterns and best practices, broad community and available skills on the job market world wide
Garbage collection	Automated memory management	Stable long-running processes and services
Threading model	Native, performant multithreading	Basis for high concurrency in the engine
Cloud-native support	Easy to containerize and run on Kubernetes	Aligns with modern infrastructure standards
Community & tooling	Broad support, monitoring, profiling, IDEs	Reduces risk and accelerates troubleshooting

Aspect	Benefit	Customer relevance
Backward compatibility	Old code often runs on newer JVMs with minimal change, however sometimes some migration work was needed, e.g from Java 11 to Java 17	Long-term investment protection

Try It Yourself

The following options make it easy for you to move from theory to practice. Whether you prefer a local Docker-based setup, experimentation with AI-assisted orchestration via MCP, or a fully hosted trial environment. We offer a low-friction path to explore Digital.ai Release hands-on.

For a zero-install option, request a hosted SaaS environment through the Release free trial page: <https://digital.ai/products/release/free-trial/>.

Using the official Digital.ai Release Docker images and container quickstart:

```
docker run -d \
-e ADMIN_PASSWORD=yoursecretpwd \
-e ACCEPT_EULA=Y \
-p 5516:5516 \
--name xl-release xebialabs/xl-release:25.3.0
```

<https://docs.digital.ai/release/docs/docker/docker-images-for-xl-release> and <https://docs.digital.ai/release/docs/24.1/docker/getting-started-with-xl-docker-containers>.
<https://github.com/xebialabs-community>
<https://docs.digital.ai/release/docs/how-to/release-mcp-server>.

Conclusion

Seen in that light, Digital.ai Release is not a special case, but a representative example of many large scale systems. Many enterprise applications face the same forces: growing scale, increasing integration demands, higher availability expectations, and constant pressure to adopt new technologies.

The lesson from this journey is not that Java is fashionable, or that one framework or runtime is inherently superior. What matters is **business outcome** — predictable delivery, operational stability, long-term maintainability, and the ability to **evolve without disruption**. Languages, frameworks, and deployment models will continue to change, often driven by hype cycles.

Enduring enterprise systems succeed for different reasons: a stable core, clear extension points, disciplined architecture,

and the willingness to evolve incrementally. In that sense, the architecture patterns described here can be applied far beyond Release to any enterprise platform that values resilience, scalability, and longevity over short-lived trends.

Today, the same JVM based engine can assist with AI-based and data-driven release orchestration through the Release MCP Server, a Model Context Protocol interface that makes the full release model securely accessible to AI agents. This MCP layer turns Release into an AI-aware control plane: agents can create and manage releases, refine templates, adjust gates, and analyze failures using natural language while still honoring existing roles, approvals, and compliance rules.

By combining structured Release metadata with MCP, teams can let AI co-pilot complex orchestration tasks without bypassing enterprise guardrails. AI assistants or agents connect via the open MCP standard, discover available tools and resources, and then operate inside the same governed environment that already enforces RBAC, audit trails, and integration policies across the release landscape.

-> Back to the Table of Content



#JAVAPRO #ARCHITECTURE #MICROSERVICES

Capability Mesh: Re-Architecting for the Agentic AI Future

Re-Architecting for the Agentic AI Future

Ten years ago, software development teams adopted a new trend: companies were breaking their large applications into smaller, independent services, an emerging pattern now known as microservices. The idea was simple: instead of one giant application, build many small services that work together. Each team could develop and deploy their piece independently. It seemed perfect.

But as companies added more services, a new problem emerged. How do you manage communication between hundreds of services? Where is each service located? How do you balance the load? What happens when a service fails? These questions, trivially solved within a single application, now require solutions across an entire network.

The Java community pioneered early solutions. Frameworks like Spring Cloud introduced tools such as Eureka for finding services and OpenFeign for calling them. Developers no longer needed to remember specific network addresses. They could simply reference a service by name, like "payment-service", and the framework handled the rest.

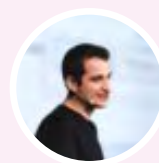
This pattern proved so valuable that it evolved beyond Java. When containers became popular, the concept transformed

into what we now call a service mesh. Tools like Istio and Linkerd moved service management completely outside the application code. They created an infrastructure layer that handled reliability, security, and traffic management automatically.

The service mesh fundamentally changed how we think about connecting services. Instead of hardcoding IP addresses and

Author:

With 25 years of experience as a developer and architect, Makan Sepehrifar comes from an entrepreneurial background and has worked in multiple domains including banking, fintech, telecom, digital transformation, and developer experience. Throughout his career, he has viewed architecture as a socio-technical discipline, recognizing that organizational psychology and team topology are as critical as technical design in creating systems that align with business goals and accelerate value delivery.



<https://www.linkedin.com/in/makan1869/>

ports, developers referenced services by their functional purpose. The mesh infrastructure took care of everything else: finding the service, routing traffic, handling failures, and managing security. This abstraction was revolutionary. It separated what a service does from how to reach it.

MCP: The Rise of AI Agents

Today, we are witnessing another transformation. Generative AI and intelligent agents are changing how systems work together. The difference is profound.

In traditional service mesh architecture, developers manually orchestrate workflows. They decide which services to call and in what order. A developer writes code that says: "First, validate the user. Then, check the inventory. Finally, process the payment." Every step is explicitly programmed.

AI agents operate differently. They can reason about what a user wants (intent detection) and autonomously decide how to accomplish it. They don't follow a predefined script. Instead, they understand intent and compose solutions dynamically.

This shift creates a major distinction. Service mesh focuses on functionality, i.e. what technical operations a service provides. AI agents, however, need to understand capabilities, what business goals a system can achieve, what outcomes it guarantees, and what constraints it operates under.

Consider payment processing. In traditional architecture, a developer explicitly calls specific methods: `PaymentService.creditCard()` or `PaymentService.directDebit()`. The developer chooses the implementation.

With AI agents, the approach changes. An agent doesn't care about implementation details. It queries a registry for agents that have the capability to "accept payment from users." The registry returns all available options: credit card, direct debit, digital wallet, or cryptocurrency. The agent then selects the most appropriate method based on context, user preference, and business rules.

Enter the Capability Mesh

This evolution demands a new architectural paradigm: the Capability Mesh. In this model, capabilities, not services, become the primary building blocks. Agents reference each other not by technical names or network endpoints, but by the capabilities they provide.

The architecture combines several elements. Self-Contained Systems (SCS) provide the foundation, independent systems that own their data and UI. The Model Context Protocol (MCP) replaces traditional REST APIs with semantic descriptions that emphasize intent and outcomes. MCP registries maintain catalogs of available capabilities. MCP gateways act as control planes, managing how agents discover and access capabilities.

The infrastructure is already emerging. MCP is reimagining how we describe system capabilities. Agent orchestrators like Claude

and GPT are learning to autonomously compose capabilities into workflows. The technical pieces exist. What we need now is a fundamental shift in how we think about distributed systems.

Why Self-Contained Agents Matter

The Capability Mesh builds on Self-Contained Systems for three important reasons. First, these systems emphasize idempotency. This means an operation produces the same result whether it runs once or multiple times. In a world where agents might retry actions or execute them in unexpected sequences, this predictability is essential.

Second, Self-Contained Systems enforce strict isolation. Services do not communicate directly with each other. They operate independently. This isolation gives agents true autonomy. Each capability stands alone. An agent can combine capabilities freely without worrying about hidden connections or unexpected interactions between services.

Third, these systems favor integration at a higher level, not within the service logic itself. Empirically Self-Contained Systems, encourage integration at UI Level and not in-between the Services meaning that services do not coordinate with each other internally. The same happens in Capability Mesh Architecture, as integration does not happen within the agents but instead, coordination happens through the orchestrator. This design places responsibility exactly where it belongs: with the intelligent agent that understands what the user wants and can decide how capabilities should work together.

These three principles transform services from connected components into independent capabilities. Agents can discover them, understand them, and compose them autonomously. This foundation makes the Capability Mesh possible.

The Challenge of Non-Determinism

This new architecture introduces a significant challenge: non-determinism. Traditional systems are predictable: you call a service, you get a result. You can trace every step. With AI agents, unpredictability is inherent. Large Language Models don't always produce the same output for the same input. Agent decisions can vary based on subtle context changes.

This unpredictability worries many organizations. The common response is to require "human in the loop", having people review and approve every significant agent action. While this improves safety, it defeats the purpose of autonomous agents. If humans must approve every decision, we've simply created a more expensive and slow version of traditional automation.

Policy as Code: The Real Solution

The answer lies in a different approach: policy as code. Instead of reviewing individual actions, we define deterministic boundaries that govern agent behavior. We establish rules that specify what agents can and cannot do, what side effects are acceptable, and what constraints must always be honored.

Think of it like city planning. We don't approve of every individual's daily route. Instead, we build roads, traffic lights, and signs that guide everyone safely to their destinations. We set speed limits, define right-of-way rules, and establish zones where certain activities are prohibited. The system provides freedom within defined boundaries.

Policy as code works similarly. We create computational guardrails that prevent undesired agent behaviors before they occur. These policies are deterministic and enforceable. They don't rely on catching mistakes after the fact; they prevent mistakes from happening.

This shifts the human role from "human in the loop" to "human on the loop." People are no longer decision-makers for every action. Instead, they become architects and governors. They design the landscape of capabilities, establish the policies that guide agent behavior, and supervise the overall system. They define what success means and where the boundaries lie, then let agents execute autonomously within those constraints.

This is preventive risk management, not mitigative. We don't wait for problems to occur and then fix them. We design the system to prevent problems in the first place.

A Fundamental Shift

The transition from Service Mesh to Capability Mesh represents more than a technical upgrade. It requires a new mindset about how we build distributed systems. We must move from thinking about services and functions to thinking about capabilities and outcomes. From manual orchestration to policy-driven governance. From human approval of every action to human design of the decision space.

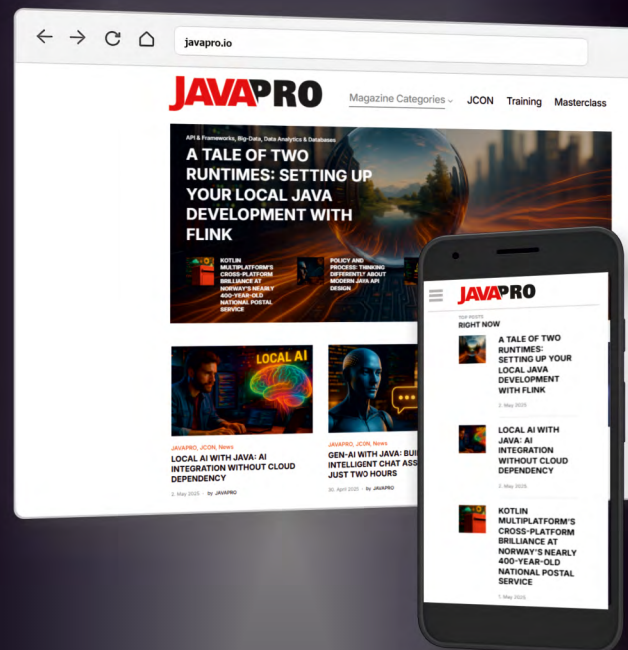
Agents navigate situations that are far more complex than what developers managed in the service mesh era. But the opportunity is equally profound: systems that can understand intent, compose solutions dynamically, and operate autonomously within well-defined boundaries.

The future of distributed systems is not defined by simply removing humans or by forcing them into a "human-in-the-loop" approval process, which is often an insufficient, slow, and expensive form of risk mitigation. The solution lies in a more fundamental architectural shift: designing systems for inherent autonomy and resilience. This transition elevates the role of the developer and designer from manually orchestrating workflows to strategically governing the system's operational boundaries. By defining deterministic policies, we ensure that agents can operate autonomously and dynamically, yet only within guardrails that prevent undesirable actions. This focus on governance over execution is the defining change of the Capability Mesh.

-> Back to the Table of Content

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !



#JAVAPRO #CI #CD

Bye-Bye Jenkins: Why We Migrated Our Pipelines to GitLab CI

In the project environment of bahn.de and DB Navigator, we have migrated our pipeline engine from (several) Jenkins instances and hundreds of pipeline jobs to GitLab CI over the past few months. Due to the large number of instances, pipeline variants, and special features, this was a mammoth task that took us a lot of time and effort. Nevertheless, the migration was naturally associated with high expectations: standardization and unification of the development infrastructure, reduction of build infrastructure costs, reduction of cognitive load in the teams, and less exposure to security vulnerabilities.

Conceptually, there are some differences between the two platforms (is one really “better” than the other?), which we want to discuss. In addition to the reasons for our decision to migrate

and our lessons learned, we want to provide insights into how to find a viable build infrastructure with GitLab CI (especially GitLab CI Components) in a larger organizational environment (20+ teams with 120+ developers). Last but not least, we also want to address the things we miss from Jenkins and find annoying about GitLab CI, and finally answer the question of whether we regret the switch or would do it again with the knowledge we have today.

Pipeline Engines

In modern software development, your pipeline engine is a critical component in your tool chain because manual builds and deploys have essentially ceased to exist, and more often than not have become virtually impossible due to the complexity and size of typical current pipelines. Therefore, their most important requirements are availability, reliability, convenience, and versatility.

In our case, we had begun building our new platform around 2017, a time when Jenkins was in high demand in the Java ecosystem in particular, and GitLab CI was still in its very early days and only had beta status. Because we wanted to get our development teams running as quickly as possible, we provided them with a basic Ansible setup to quickly spin up their own Jenkins instances, and for the most part, let them customize and configure them as they saw fit.

Author:



Lukas Pradel is a software engineer at Deutsche Bahn / DB Fernverkehr AG. He likes to spend the time he saves by automating everything on riding his motorcycle.

<https://www.linkedin.com/in/lukaspradel/>

This approach worked well for us in the early stages, but as we got closer to shipping services towards production, we had to start bullet-proofing our Jenkins instances. Suddenly, our teams had to guarantee that their build servers were always available in case of required hot-fixes; they needed to make sure that they could be recovered within just a few hours in case of catastrophic infrastructure disaster events. Furthermore, they had to continuously monitor them for recent security vulnerabilities, i.e., frequently update and patch them, which could occasionally be very time-consuming. We noticed that the cognitive load of our teams increased considerably as they had to invest a growing portion of their capacity into running their build servers.

Lastly, the diversity in instances which were highly adapted and optimized for the specific needs and habits of a single team turned into a sizeable obstacle whenever we needed to rotate engineers into different teams, as they had to dig into the specific ins and outs every time.

Choosing a Migration Approach

Mainly due to the size of our organization and our requirements, we quickly agreed that a single scalable managed build server would be best suited for our needs. In our case, we have more than 20 teams maintaining over 100 microservices, each of which requires multiple daily deployments, so we required a build server that could sustain 100+ concurrent builds during peak hours. Naturally, this could have been achieved with a managed Jenkins instance set up adequately as well, or with the help of e.g. CloudBees.

However, in our case, we had been using (and paying for) GitLab as our SCM from the start. We were hoping that by simply using it more intensely, we would also be able to cut our infrastructure expenses a bit, because GitLab runners allow for very precise dynamic scaling. In contrast, with Jenkins dynamic agents, this can be quite tricky. The cloud instances for our various Jenkins masters had been on-demand due to our hotfix readiness requirement, and therefore, were rather costly. The same requirement served as a credible justification for having a platform team with the sole and clear purpose of ensuring a stable SCM and pipeline system for all of our development teams.

While the service pipelines themselves were very homogeneous among services within the same domain or team, there were vast differences between pipelines from different domains and teams. To address this, each team had developed and maintained a shared Jenkins pipeline library¹. In declarative Jenkins pipelines, these allow you to reuse steps or entire stages of a pipeline without having to copy and paste pipeline code everywhere, which makes things a nightmare to maintain and update. They can also be versioned and tested, which makes them a very powerful tool for maintaining your pipelines.

GitLab CI provides a similar mechanism with CI/CD components². These are basically reusable pipeline building blocks that can also be configured via input arguments and versioned to ensure stability and continuity in the development of your pipelines. Furthermore, components can be published into the CI/CD

catalog³, which allows other teams to use pipeline components as well. We found this to be exceptionally useful in cases where pipeline steps are truly universal (e.g., static code analysis via Sonar or static code analysis for security vulnerability with managed tools), so that it suffices to implement them once for all teams rather than every team having to reproduce them in their respective components.

Since all our Jenkins pipelines were declarative and built using shared pipeline libraries and we saw no need to change their overall structure, the actual migration was very straight forward: after taking inventory of the building blocks in the shared pipeline library that we decided to keep, it came down to just rewriting them into GitLab CI components (so basically migrating from Groovy scripts to shell scripts ran in suitable docker containers). Therefore, the migration could be broken down into very small, mostly independent tasks that could be performed in parallel, which made estimation and planning very easy after we had gathered some initial experience.

Since GitLab CI components are built and published into the catalog with a GitLab pipeline themselves, we had the bonus of being able to properly and thoroughly test our components by executing them in this very pipeline which actually helped us identify a few bugs along the way as well and improved overall quality of our pipeline code which had previously been treated somewhat sloppily because it was not actual application code – a fallacy many software engineers fall for, but can prove fatal. A painful lesson everyone learns once their faulty pipeline code ships code to production that should have broken your pipeline.

Viable Build Infrastructure in Large Organizations

In our case, the migration has certainly paid off. We were able to shut off most of our infrastructure related to Jenkins by now, which has resulted in considerable savings even after you factor in the increased demand in GitLab runner infrastructure. Our development teams have been unchained from the burden of having to maintain and run their build infrastructure and have considerably improved their feature output. The dedicated platform team is focused on their single task, and they are very good at it – unlike before, we have not had any build server downtimes worth mentioning beyond scheduled maintenance windows.

We found that with the way our pipelines work, execution times have actually reduced with the shift, and success rate has improved as well, which was unexpected considering that we are seeing almost 1 million pipeline runs a month. As it turns out, most of our previous Jenkins instances suffered occasional “hiccups” that needed manual fixing every once in a while.

While Jenkins allowed for a near limitless degree of customization, our teams agreed unanimously that they are glad to be rid of the plugin version compatibility issues that plague Jenkins to this day. When we found bugs or lacked certain features in Jenkins or one of its plugins, we would occasionally contribute changes, which was convenient as Jenkins is entirely open source. However, to a large extent, the same holds true for GitLab, which is “open core”

and allows for community contributions. When we were unable to solve issues on our own, enterprise support can certainly be a worthwhile investment if your organization is big enough.

Due to the way our pipelines work, we frequently use the “restart from stage” action in Jenkins to pick up on pipelines that were failed or suspended for various reasons. This action was always flaky in Jenkins (in particular if nodes were shut off or restarted in between pipeline runs), whereas the GitLab CI appears to do a much better job at properly persisting and restoring workspaces from single stages. More generally, in direct comparison, we found that Jenkins appears to be way more stateful, with all its implications, whereas GitLab CI stages are truly isolated and independent, built on a very simple foundation of Docker containers and shell scripts within them.

While plugins can be one of the most annoying traits of Jenkins (especially when they are poorly maintained or documented), they are occasionally also among the most useful. To this day, we dearly miss the plugins that provide clear graphic reports from test results, for example, of Cucumber integration tests or Gatling performance tests – a feature that GitLab sadly lacks to this day.

In general, however, you will find overwhelming amounts of publicly available troubleshooting help for Jenkins-related issues in the usual channels, whereas GitLab CI still has some catching up to do.

Conclusion

In summary, we have concluded that for large-scale software development organizations, a managed pipeline engine instance is highly advisable to optimize infrastructure costs, standardize among development teams, account for personnel shifts, and keep cognitive load focused on the feature tasks at hand. While neither Jenkins nor GitLab CI is inherently better, both are more or less equally capable of scaling and enabling the demands of enterprises, in particular if you can opt for enterprise support from either GitLab or CloudBees (or their likes).

In hindsight, we have not regretted our expensive and taxing migration effort as our development has seen the boost we were hoping for, while reducing our costs and making our build server more robust.

As with all major migration tasks, the decision of swapping your pipeline engine should not be made lightly. Carefully consider your priorities, requirements, and external circumstances, and weigh your options before arriving at your decision. While both systems have their pros and cons, no universal recommendation can be made, as the specifics of any software organization are unique and must be taken into account. Even so, there are cases in which the overall outcome and improvements outweigh the considerable effort and investment.

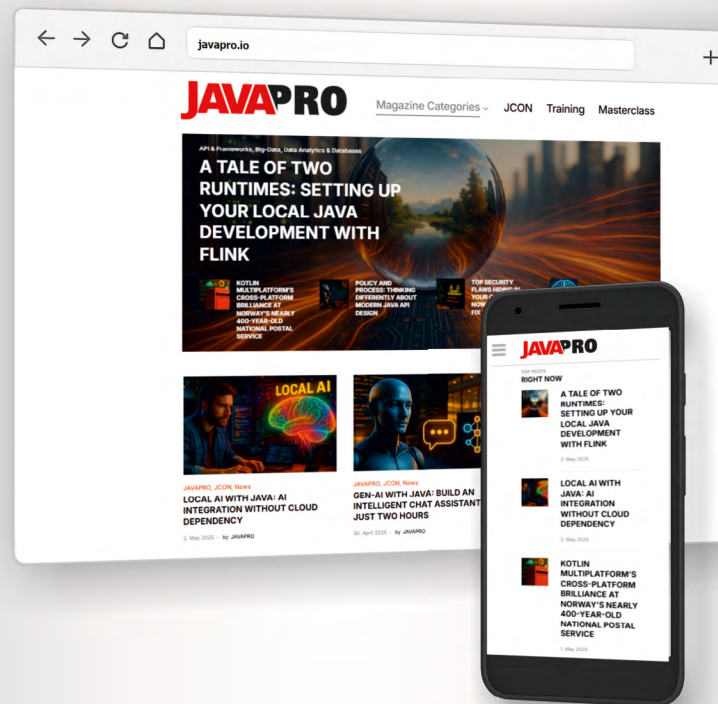
References

- <https://www.jenkins.io/doc/book/pipeline/shared-libraries/>
- <https://docs.gitlab.com/ci/components/>
- <https://docs.gitlab.com/ci/components/#cicd-catalog>

-> **Back to the Table of Content**

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

www.javapro.io



#JAVAPRO #PROJECTMANAGEMENT

AI's Impact on Team Dynamics

After spending years advocating for strong team dynamics in software engineering, I have been fascinated as AI tools have entered our development workflows. In one of my previous blogs, I touched briefly on how AI might change team dynamics. Now, I want to dive deeper into this transformation, because what we're experiencing isn't just a new tool, it's a fundamental shift in how we work together.

The question isn't whether AI will impact team dynamics. It already has. The real question is: how do we maintain the human connections that make teams truly effective while embracing these new capabilities?

Spoiler alert: the answer isn't about the technology itself, it's about the choices we make in how we use it. AI can be an extremely powerful tool, and I'm a strong supporter of its use. There are hidden dangers though. Using AI irresponsibly can result in social isolation as I will explain in this article, but we can avoid that by using it consciously and intentionally.

The Isolation Paradox

Here's something I've noticed: AI assistants are incredibly helpful, but they're also incredibly solitary. When a developer gets stuck on a problem, they now have a choice. Walk over to a colleague's desk (or ping them on Slack/Teams) and discuss the issue, or ask an AI assistant for a solution.

The AI option is tempting. It's instant. It doesn't judge. It won't roll its eyes at what might seem like a basic question. And it's always

available, even at 2 AM when you're debugging that critical production issue.

But here's the paradox: every time we choose AI over human interaction, we lose something valuable. We lose the spontaneous conversations that lead to unexpected insights. We lose the knowledge sharing that happens naturally when explaining a problem to someone else. We lose the social bonds that make teams resilient during challenging times.

I've seen developers who used to be highly collaborative become increasingly isolated. They're productive, sure. Their code gets written. But they're operating in a bubble, reinforced by an AI that reflects their own thinking patterns back at them.

Author:

Erik Pronk is a software engineer and architect at JDriven, with a strong focus on software craftsmanship and effective team dynamics. He cares deeply about building sustainable software, shaping team interactions, sharing knowledge, and enabling growth within teams and communities.



<https://www.linkedin.com/in/erik-pronk-1196a7/>

The Echo Chamber Effect

When you collaborate with humans, you get challenged. Your assumptions get questioned. Someone points out the edge case you missed or the security vulnerability you overlooked. This friction, while sometimes uncomfortable, is what makes software robust and developers evolve to the 'next level'.

AI assistants, for all their capabilities, tend to be agreeable. They generate solutions based on patterns they've learned, but they don't push back the way a senior developer would. Junior developers do this too through casual feedback that forces you to reconsider assumptions you didn't know you were making.

I've reviewed code where developers clearly used AI assistance, and while the code works, it often lacks the thoughtfulness that comes from human discussion. It solves the immediate problem but misses the bigger picture. The architectural considerations and more important the lessons learned from past mistakes.

This creates an echo chamber where developers stay trapped in their own perspective, with AI amplifying rather than challenging their approach.



The Knowledge Transfer Crisis

One of the most underappreciated aspects of team dynamics is knowledge transfer. When a junior developer pairs with a senior developer, they're not just solving a problem together. They're building a mentorship relationship where years of experience, judgment calls, and professional wisdom get transferred. Not just code patterns, but how to think about problems, when to ask for help, and how to navigate uncertainty.

AI disrupts this natural knowledge flow. Junior developers can now generate working code without understanding the underlying principles. They can solve problems without learning why those problems existed in the first place.

I'm not suggesting we should make things artificially difficult. But there's a difference between using AI as a learning accelerator and using it as a replacement for learning. When developers rely on AI without building their own understanding, they become dependent or, even worse, addicted to AI. They can't debug effectively. They can't adapt when requirements change. They can't mentor the next generation.

The team's collective knowledge, that shared understanding that makes experienced teams so effective, starts to erode.

But It's Not All Negative

Let me be clear: I'm not anti-AI. Far from it. AI tools have great benefits for team dynamics when used thoughtfully. And I believe we are at a tipping point in the history of software engineering. Like Werner Vogels put it in his keynote at AWS re:Invent, it's the dawn of something new, the renaissance developer.

AI can handle the tedious work that drains energy and motivation. Boilerplate code or repetitive refactoring and not forgetting documentation generation. By automating these tasks, AI frees up mental space for the creative, collaborative work that we as humans excel at.

I've seen teams use AI to level the playing field. Developers who struggle with English as a second language can use AI to help articulate their ideas more clearly. Introverted team members who find real-time discussions challenging can use AI to prepare their thoughts before meetings. In both cases, AI removes barriers that have nothing to do with technical ability, ensuring that good ideas get heard regardless of who voices them.

AI can also serve as a catalyst for discussion. Instead of one person proposing a solution, a team can ask AI to generate multiple approaches, then discuss the trade-offs together. This shifts the conversation from "my idea versus yours" to "which of these options best fits our needs?"

When a team uses AI as a shared tool rather than individual aid, it can enhance collaboration.

The Speed Versus Depth Trade-Off

AI enables incredible speed. Features that used to take days can now be built in minutes. But speed isn't always what teams need.

Software development is as much about understanding the problem as solving it. It's about exploring different approaches. It's about building shared context so the team can maintain and evolve the system over time.

When teams optimize purely for speed, they sacrifice depth. They ship features faster but accumulate technical debt. They solve immediate problems but miss opportunities for systemic improvements.

I've seen teams that became addicted to AI-assisted velocity. Their sprint velocity looked impressive on paper, but they sacrificed code quality and introduced security vulnerabilities.

The teams that succeed with AI are those that consciously balance speed with depth. They use AI to accelerate the straightforward parts, then invest the time saved into deeper discussions about architecture, design, and long-term sustainability.



Maintaining Psychological Safety in the AI Era

Psychological safety, the foundation of effective team dynamics, faces new challenges in the AI era.

When developers can generate solutions privately with AI, they're less likely to expose their thinking process to the team. They're less likely to admit when they're stuck or uncertain. This personal vulnerability that builds trust gets replaced by a facade of AI-assisted competence.

I expect this effect to be particularly visible with medior developers. Junior developers are expected to ask questions, so they face less stigma in seeking help. Senior developers have already built their network and credibility. But medior developers occupy an awkward middle ground, they feel pressure to appear knowledgeable, and AI gives them a way to maintain that appearance without asking for help. This prevents them from building the deeper relationships that would actually support their growth.

Teams need to actively cultivate an environment where it's safe to say, "I used AI to create this, but I'm not sure it's the right approach. Can we review it together?" This requires leadership that models this behavior and explicitly values learning over appearing knowledgeable.

Collaboration

Collaboration is a skill. Like any skill, it requires practice. When developers spend more time interacting with AI and less time interacting with humans, their collaboration skills decline.

They become less adept at explaining their thinking. Less patient with the unstructured way we as humans use for communication. Less skilled at navigating disagreements and finding compromise.

The consequences ripple through every aspect of teamwork. Code reviews that used to spark rich discussions now get perfunctory approvals. Developers assume that if the code works and AI helped generate it, it must be fine. The critical thinking that makes code reviews valuable gets bypassed.

Pair programming becomes less common. Why pair when you can just ask AI or pair program with your personal AI assistant? But pairing isn't just about solving problems faster. It's about building shared understanding, transferring knowledge, and strengthening team bonds.

The result is teams that function more like collections of individuals working in parallel. Feedback loops between humans diminish instead of strengthening over time as they do in true collaborative teams.

Strategies for Healthy AI Integration

So how do we get the benefits of AI without losing the human dynamics that make teams effective? Here are some of my thoughts:

Establish AI usage criteria as a team. Don't let AI adoption happen organically. Discuss as a team how you want to use these tools. When is AI appropriate? When should you seek human input instead? Make these criteria explicit, and ensure they also account for relevant legislation and moral considerations. Ethical and legal boundaries should guide your decisions just as much as technical convenience.

Use AI as a conversation starter, not a conversation ender. When AI generates a solution, treat it as a draft to discuss, not a final answer. Ask, "What do we think about this approach? What are we missing?"

Protect collaboration time. Schedule regular pairing sessions, architecture discussions, and knowledge-sharing meetings. Make these non-negotiable, even when AI could theoretically replace them.

Rotate AI-free days. Some teams experiment with days where AI tools are off-limits. This forces developers to rely on each other and keeps collaboration skills sharp.

Make learning visible. Encourage developers to share what they learned from AI interactions. "I asked AI about this pattern, and here's what I discovered." This transforms individual AI usage into collective learning.

Measure what matters. Don't just track velocity. Track code quality, knowledge distribution across the team, and team satisfaction. If AI is improving velocity but harming these other metrics, adjust your approach.

The Future of Team Dynamics

AI isn't going away. If anything, these tools will become more capable and more integrated into our workflows. The teams that thrive will be those that consciously shape how AI fits into their dynamics.

To achieve this requires treating AI integration as a team dynamics challenge, not just a technical one. It requires ongoing conversation about what we're gaining and what we're losing. It requires leadership that values human connection as much as productivity.

Michael Jordan couldn't win championships alone, no matter how talented he was. The same is true for software teams. AI can make individual developers more productive, but it can't replace the collective intelligence, creativity, and resilience that emerges from strong team dynamics.



Conclusion

AI's impact on team dynamics is neither purely positive nor purely negative. It's a powerful tool that amplifies our choices. If we choose isolation and speed over collaboration and depth, AI will accelerate that path. If we choose to use AI as a catalyst for better collaboration, it can enhance our team dynamics.

The key is intentionality. We need to actively design how AI fits into our teams, rather than passively accepting whatever dynamics emerge. We need to protect the human interactions that make teams effective, even as we embrace the productivity gains AI offers.

Because at the end of the day, software engineering is still a human journey. The code we write serves human needs. The systems we build are maintained by human teams. And the success of our projects depends on human collaboration.

AI can be a powerful ally in this work. But only if we ensure it enhances, rather than replaces, the team dynamics that make us effective.

So as we integrate AI into our workflow, let's ask ourselves: Is this making me more connected to my team, or more isolated? Is this deepening our collective understanding, or creating individual silos? Is this enhancing our collaboration, or replacing it?

The answers to these questions will determine whether AI strengthens or weakens your team dynamics. Choose wisely.

-> Back to the Table of Content



#JAVAPRO #SECURITY

PQC – It Is Time to Act

For many IT experts, quantum computers are a technology of the future. They are not wrong. A massive amount of basic research needs to be done for quantum computers to become a reality. Currently, they cannot be used to solve any real-world problems. A lot of basic issues and problems will have to be resolved until quantum computers can be used for real-world problems.

But even now, the development of quantum computers impacts our currently used IT. In the future, quantum computers will be able to solve problems that cannot be solved by conventional computers without an impractical amount of resources, if at all. These new opportunities can be used to do good, for example, in the development of new medicines, but they can also be used to do harm, as encryption can be cracked more easily. The latter is something we have to deal with right now. Current quantum computers cannot crack any of our encryption, but we have to prepare for the time when they can. We have to do everything now for the time when quantum computers will become available.

But why can quantum computers crack our currently used encryption, and what can we do against it now?

What Are Quantum Computers?

Computers are based on the concept of a bit. A bit can have a value of 0 or 1. These values can be represented as power on or power off. Bits can be processed with logical operators and can again be stored as a bit. Quantum computers are based on the concept of a qbit. A qbit can have a value of 0, 1, and everything

in between all at the same time. Qbits are based on the behaviour of quanta at the subatomic level. Operations are performed by entangling these quanta.

You can illustrate the behaviour of a qbits with a coin. A coin can either be heads or tails. The coin can also be tossed into the air and spin, in which case it will have all values at the same time.

It's very complex and expensive to realise a qbit. The quantum effects can only be realised near absolute zero (0 Kelvin or -273.15 degrees Celsius). Qbits are very sensitive to external influences. To compensate for these errors, multiple physical qbits are combined into one logical qbit. All these requirements lead to current quantum computers only possessing a very low number of qbits.

Author:

Sebastian Hempel is a self employed consultant and trainer from the north of Bavaria. Since 2003 he supports customers in developing and operating enterprise systems. His main focus lays in the development of systems in Java (JakartaEE and Quarkus) on Linux based systems.



<https://www.linkedin.com/in/sebastian-hempel-07a50a224/>

Google was able to realise a quantum computer with 53 logical qubits in 2019. IBM presented a quantum computer with 7 logical qubits in 2001. There are estimates that in 2030 there might be quantum computers with between 100 and 1 million logical qubits. The discrepancy between these two numbers already makes it clear that the development in this area is every bit as obvious and straightforward.

What Are the Basic of Current Cryptography?

Our current used encryption is based on mathematical problems that cannot be solved in an easy way. Solutions for these problems are only possible with a huge amount of computing power.

There are basically two different methods of encryption: symmetric and asymmetric encryption. With symmetric encryption, both sender and recipient use the same key. Examples for symmetric encryption include AES or DSA. Symmetric cryptography can be implemented in hardware and therefore can be computed very quickly. Asymmetric encryption uses two different keys. One key is used for sending and the other for receiving. An example for asymmetric encryption is RSA. The different methods are combined into crypto systems for encrypting and signing data.

RSA is based on the mathematical problem of factoring a large prime number into its prime factors, for which there is no easy mathematical process for this, only brute force. The larger the number, the more computing power is needed. There are certain methods to reduce the number of iterations to find the prime factors, but the remaining number is still impractically huge. Modern cryptography often uses elliptic curves. These are based on the problems of finding the discrete logarithm of the elliptic curve. Both problems cannot be solved by a simple algorithm; there is always a huge amount of try and error/brute force needed.

How Are Quantum Computers Able to Break Cryptography?

Quantum computers do not work with just 0 or 1, but with all values at the same time. This way, they can try more values at the same time. Thus, the time to crack a key can be reduced from weeks or months to just minutes or hours.

In the year 1994, Shor developed an algorithm to break asymmetrical cryptography more easily with the help of quantum computers. An adaptation of Shor's algorithm can also be used to crack elliptic-curve cryptography. With this, both asymmetric methods will be useless after the availability of quantum computers with the right amount of qubits.

There are different estimates about the amount of qubits needed to crack these methods. For an RSA key with 2048 bits, there is an estimation that around 4000 error-corrected/logical qubits will be needed. A quantum computer would need millions of physical qubits to deliver this amount of logical qubits. In 2019, it was estimated that a quantum computer with 20 million qubits would need 8 hours to crack a 2018 bit RSA key. Conventional computers would need years to do this.

Why Is This Important Now, When Quantum Computers Are Not Yet Available?

You may ask, why you have to deal with this problem now, even though quantum computers with the needed amount of qubits are not yet available and will – if ever – only be available in the future. The answer is quite easy: you can now store encrypted data to crack it and decrypt it later, when the needed quantum computers are available. This method is also called harvest now, decrypt later.

This process is interesting for very sensitive information that is relevant even decades after. Information like this can, for example, be found in the healthcare sector. Information about a patient is relevant for years and needs to still be secure in 10 years or later. Companies also rely on their information being secure for decades.

Because of this, it is important now – years before quantum computers are available – to start securing our cryptography by switching to so-called post-quantum cryptography, which can already be encrypted in a way that cannot be cracked by quantum computers in the future.

What Is Post Quantum Cryptography (PQC)?

Post-quantum cryptography describes encryption algorithms that are not based on mathematical problems that can be cracked more easily by quantum computers. You don't need a quantum computer to use these algorithms. These algorithms are already usable on currently available computers.

Post-quantum cryptography is the answer to the problem that currently encrypted information can be stored to be later decrypted with a quantum computer in the future. By using these new cryptographic methods, the information will still be secure in the future.

There are classes of methods that can resist attacks with quantum computers:

Lattice-Based Cryptography

This cryptography is based on the mathematical problem of finding the shortest path between two points in a multi-dimensional lattice. This problem can only be solved with an impractically large amount of computing power, even when using a quantum computer. Kyber is a key exchange mechanism (KEM) based on this problem.

Hash-Based Cryptography

Hashes are already used in cryptographic systems. SHA-256 is an example of a cryptographic hash algorithm. As long as the (symmetrical) key is large enough, hash algorithms are relatively resistant against attacks with quantum computers. An example of such a method is Sphincs+; digital signatures created with this algorithm are large and need a huge amount of computing time.

Code-Based Cryptography

This cryptographic method is based on the theory of error-correcting codes. Algorithms based on this problem have been known since 1987. Until now, there has been no successful attack with quantum algorithms against this method. The disadvantage of these methods is very big keys often in the range of kilobytes.

What Standards Are Available?

Cryptologists have been working on new algorithms for a long time. Since 2016, NIST (National Institute of Standards and Technology) has been working on new standards for post-quantum cryptography. In August 2024, NIST published 3 new standards:

FIPS 203/ML-KEM is a new standard for key exchange (key encapsulation mechanism). This lattice-based method replaces RSA/ECDH.

FIPS 204/ML-DSA is a standard for creating digital signatures. This method is also lattice-based. Like ML-KEM, this algorithm was also created by the CRYSTALS team.

FIPS 205/SHL-DSA is another standard for creating signatures. This is a hash-based algorithm based on Sphincs+.

Can I Already Switch to Post-Quantum Cryptography?

After the publication of the above-mentioned post-quantum algorithms in August 2024, engineers start to implement these new methods. Today, more and more implementations of these standards are available and can be used.

The first companies are already using PQC in production. A report from September 2025 shows considerable usage by Cloudflare and Amazon.

A prerequisite for the use of PQC is the use of TLS 1.3; older versions of TLS are not capable of PQC ciphers. Many browsers already support PQC during the TLS handshake. The server, however, needs to support this; if not, the old algorithms will be used. By this, backwards compatibility is available.

It depends on your software if you are able to use PQC in production. In the future, there will be more and more software that will support PQC. More and more fundamental libraries are implementing PQC standards. You need to consider that the PKI also needs to support handing out PQC certificates.

Can I Use Both At the Same Time?

The currently established algorithms will not be completely replaced by PQC in the foreseeable future. It will be necessary to support the "older" algorithms alongside PQC algorithms. This hybrid model was planned and supported from the beginning.

Systems therefore support both the old and the new algorithms.

The communications partners will first try to establish a connection using PQC. If this is not possible, because one of the parties does not support PQC, the old algorithms will be used; thus ensuring backwards compatibility.

What Do I Need to Take Into Consideration?

You have to consider a few aspects when introducing PQC.

All involved systems need to support PQC algorithms. If this requirement is fulfilled, systems need to be equipped with PQC certificates. Keys for PQC are without exception bigger than keys for currently used algorithms. Systems with restricted resources (memory) might run into issues. But not only this, communications also get more elaborate. A key for ECDA, for example, has a size of 64 bytes. When switching to MD-DSA, the key size will be 1312 bytes. Also, signatures will increase in size. For example, ECDSA-P256 uses a size of 64 bytes. ML-DSA will increase the signature size to 2420 bytes.

PQC algorithms need more computing power than currently used algorithms. PQC algorithms are also not as mature as other algorithms. They are not yet as optimised as others. Systems with limited computing power may run into some issues because of that.

The older cryptographic methods are, and the more often they are checked for weak points, the better they are. PQC algorithms are very young and not yet as tested as currently used methods. They were extensively checked/analysed in the selection process. But nobody knows if there won't be any attack vectors in the future. Thus, you need to take into account that you will have to update/replace the PQC algorithm in the future.

The quick response to attacks on cryptographic systems is also known as crypto agility. In the future, we cannot implement a cryptographic system and use it for years. We need to react to attacks/weaknesses by updating/replacing the algorithm as quickly as possible. Only with modular cryptography and established processes will we be able to meet these challenges.

What Is the Best Way to Implement Post-Quantum Cryptography?

To introduce/implement PQC, the following steps should be taken:

Create a crypto inventory. All systems in a company/organisation/ datacentre need to be scanned and catalogued in this inventory. What cryptography is used by these systems? What keys are used? What version of the software is used in production? This inventory should contain all information about the algorithms and key material used.

Based on the inventory, you prioritise your systems. Systems with critical/important data have a higher priority over other systems. Systems with a higher priority are updated first, as their data is more sensitive and needs to be protected more.

After prioritising, each system needs to be examined in more depth. Are the systems already prepared for PQC? Can an update solve the lack of PQC support?

If the system needs to be replaced, PQC support should be a deciding factor in deciding what system to buy/build? New systems should come with support for PQC out of the box.

An important factor is the interfaces with external systems. These connections should be checked regularly. Are all systems capable of PQC and can they be configured that way?

If possible, at least hybrid communication with both PQC and currently-used cryptography should be activated as soon as possible.

Are There any Legal Requirement

There are currently no direct legal requirements to introduce PQC.

But the EU and USA are working on a plan to start using PQC. In the EU, organisations are required to formulate a plan on introducing PQC until the end of 2026. In this plan, switching to PQC for critical systems should be done until 2030, and all other systems should be switched over until 2035.

The NIS-2 directive demands state-of-the-art encryption for critical systems. With the switch demanded by the EU and USA, NIS-2 might also demand the switch to PQC.

For the financial sector, the G7 expert group, together with the German BaFin, has released a roadmap in January 2026 on how to switch to PQC. In this paper, an obligatory migration until 2023 is described.

Does Java Support Post-Quantum Cryptography?

In Java, the new post-quantum algorithms were implemented bit by bit after their release.

With JEP 452 in Java 21, the KEM API was introduced. This API lays the foundation for implementing different KEM providers for PQC.

With JEP 510 in Java 25, the KDF API was introduced. This API allows the hybrid usage of PQC and legacy keys in hybrid schemas.

Java 26 comes with the implementation of JEP 524, which allows the direct and easy usage of PEM files. PQC certificates will be stored and distributed in this format.

The implementation of ML-KEM and ML-DSA can be found in Java 24 with JEP 496 and JEP 497. Beginning with Java 24, you can implement PQC communication via TLS 1.3.

In practice, the most important Java version for PQC will be JDK 25, as this „LTS“ version fully supports all necessary algorithms.

What's In the Future?

Java supports PQC since March/September of 2025 with all necessary algorithms being available. Based on this JDK version, Java developers can start implementing systems supporting PQC. PQC should be used as soon as possible on the client side. If possible, the implication on the server side should also start as soon as possible. As currently used cryptography is still supported (hybrid), old systems will still be able to communicate with your PQC systems.

All developers and architects should look into PQC and at least the basics of the use of these new algorithms should be familiar to everyone. We should start the usage of PQC now and not wait for the Q-Day – availability of post-quantum computers.

-> [Back to the Table of Content](#)



#JAVAPRO #SECURITY

The Myth of Stability: Java's Software Supply Chai After Log4Shell



For as long as most Java developers can remember, we've lived inside a comforting story about our ecosystem. It's the idea that Maven Central, the Jakarta stack, the Spring universe, the vast Apache landscape, and all the independent libraries quietly carrying enterprise workloads together form a coherent, self-maintaining, self-healing whole.

A world where problems are fixed upstream, where maintainers have time, where the dependencies we pull down are being actively watched by someone who knows exactly what they're doing. It's a lovely idea. It's also never been completely real.

When you look beneath the surface of the Java systems that actually matter: the identity platforms, banking engines, warehouse systems, telco backends, and the unglamorous machinery that keeps entire countries functioning, you do not find a perfectly stewarded ecosystem.

You find layer upon layer of libraries written by people who have long since moved on. You find critical infrastructure held together by volunteers whose names, in many cases, no organisation ever bothered to learn. You see the remnants of choices made ten or fifteen years ago that nobody has touched since, except perhaps to shade into an uber-jar out of desperation.

The Heady Days of Open Source

What allowed this to work for so long was not the excellence of maintenance but the absence of evil attention. Java projects that stopped receiving commits were rarely considered abandoned; they were called "stable." Libraries that had not shipped a release since 2014 weren't suspicious; they were "battle-tested." And if a maintainer disappeared, well, most of us shrugged and assumed the project was simply complete.

Author:

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.



<https://www.linkedin.com/in/noregressions/>

Workarounds became a point of pride. If a dependency misbehaved, we wrapped it, patched it, or quietly forked it and kept the fork inside the company Nexus. Entire generations of Java engineers built careers on strategic glue code.

Nobody worried because the risk felt abstract. Attackers were unknown and, for a long time, looked elsewhere. Regulators were not crawling through JAR files. And the global Java estate, so large it has its own weather systems, continued to grow atop a sedimentary stack of frameworks and utilities whose maintainers never expected to be supporting today's planet-scale workloads.

And Then, of Course: Log4Shell

And then Log4Shell happened, and the illusion evaporated.

The moment Log4j broke the news cycle, every Java developer on the planet was forced to confront the reality of our ecosystem. A logging library that most of us took entirely for granted was embedded in everything from Hadoop clusters and Spark jobs to Spring Boot microservices, Kafka pipelines, industrial systems, and ancient J2EE deployments. The vulnerability wasn't shocking because it was complicated. It was surprising because almost the entire Java universe depended on a library maintained by a small number of volunteers with no contractual obligation, no SLA, no enterprise support desk, no hotline, and ultimately no responsibility to fix anything.

The maintainers were generous enough to fix it anyway. But the episode revealed a truth many organisations had quietly ignored: open source is given to us without warranty, without guarantees, and without any promise that it will be maintained at the pace the modern world demands. All responsibility lies with the consumer, not the creator. That's what the licences tell us, even if we choose not to read them that way.

Bad Actors Upped Their Game.

Log4Shell wasn't the first sign. Heartbleed had already shown the pattern years earlier, but that happened outside the Java ecosystem, so many teams treated it as someone else's problem. Log4Shell made that impossible. It showed us that we had built our entire world on the assumption that upstream would always be there when needed. It turned out that upstream had lives, jobs, families, and limited time.

While this was happening, attackers were quietly becoming more methodical. They began analysing ecosystems the way Java conference talks had been urging us to for years. They discovered that most Java libraries have one active maintainer. They found that many packages pulled billions of times from Maven Central exhibit signs of abandonment. They mapped dependency graphs with the same thoroughness that SCA tools do, except with a different outcome in mind. What we had long treated as dusty corners of the build became bright red targets.

Government Responds

Regulators were watching this unfold with growing clarity. Contrary to widespread complaints, laws like the EU Cyber Resilience Act weren't written out of ignorance; they were written with a painfully accurate understanding of the ecosystem. They recognised that modern software (Java, especially) relies almost entirely on open source. They understood that organisations routinely ship products they cannot patch because they have no control over the libraries within them. And they understood that "free" had accidentally come to mean "unregulated," "unsupported," and "everyone else's problem."

Their response wasn't intended to kill open source. It was designed to make organisations take responsibility for what they consume and/or ship. Anyone building software at scale knows the real ratio: only about ten per cent of most enterprise Java systems is written in-house. The rest comes from libraries, frameworks, platforms, and transitive dependencies.

Without open source, most Java applications would cost ten times as much to build. Regulators looked at that equation and concluded, quite reasonably, that if industry depends on this ecosystem, then industry (not volunteer maintainers) must be accountable for it.

Legislative Penalties

The EU Cyber Resilience Act, for example, requires software vendors to provide a Software Bill of Materials (a complete inventory of what you ship) and to maintain a vulnerability-handling process that actually works. The CRA effectively imposes a duty of care over every component in your stack. If a vulnerability appears, you're expected to address it within a reasonable timeframe. You are also expected to demonstrate that you develop software in accordance with secure development practices and to ensure that everything you ship can receive security updates for years after release.

NIS2 goes even further by tying executive responsibility directly to the security and resilience of digital infrastructure. Its reporting obligations require logging, monitoring, and transparency across the entire supply chain. And when you read the official ENISA mapping documents, you see a clear expectation that organisations must understand the security posture of their dependencies, not just their own code.

In the US, Executive Order 14028 and the NIST Secure Software Development Framework set their own expectations: signed artefacts, rigorous provenance checks, tamper-evident builds, and SBOMs that include the NIST-defined minimum elements. *These aren't suggestions; they are procurement requirements.* If you want to sell software into specific sectors, you must demonstrate that you can maintain the entire stack. Including the parts you didn't write.

The EU AI Act pushes the boundary even further by requiring traceability not only of code, but of the training data and

model lineage behind AI systems. In other words, the software supply chain now extends all the way into the internals of machine learning models. If a model or framework is built on an unmaintained dependency, the regulatory view is brutally simple: it is non-compliant by default.

The New World of Open Source

SCA tools, Vulnerability scanners, and SBOMs are no longer optional. Provenance and signing requirements are no longer nice-to-haves; they are now requirements. Vulnerability handling obligations moved from “should” to “must.” And enforcement became real rather than theoretical. Fines now reach into the millions. Executive liability is no longer a distant worry but a written clause. “We didn’t know” no longer helps anyone. “Upstream didn’t fix it” doesn’t help at all.

This hits hardest in the long tail of open source: the small libraries created by talented individuals who never intended to become critical components. These projects rarely receive funding. Their maintainers often have other jobs. Their release cadence is aligned to personal life, not enterprise timelines. They owe the world nothing more than what they have already given.

New Commercial Realities

That belief collapses immediately under regulatory scrutiny. The familiar strategy of forking a library internally, long celebrated as a sign of engineering maturity, now drags an organisation into a churn of legal review, security scanning exceptions, compliance audits, procurement headaches, and provenance complications that frequently outweigh the cost of the patch itself. A one-line fix carried as a new artefact becomes a new identity, with all the obligations that come with new software.

Meanwhile, the common reassurance that “upstream might fix it eventually” is no longer viable. Upstream might fix it, or upstream might be on holiday, or upstream might have changed careers, or upstream might have no interest at all in dealing with somebody else’s production deadline. We built a global industry on top of these rhythms, and only now are we confronting what that actually means.

Add AI to the Mix

Then came AI, and everything accelerated again.

AI coding tools understand the Java ecosystem only as it appeared in their training data, not as it exists today. They do not know which libraries are maintained. They do not know which examples are outdated. They do not know that Java moved on from specific patterns, nor that certain frameworks should not be revived from the depths of Maven Central. They simply reflect what was common. And because older code is often more abundant, AI gently amplifies the past, including insecure idioms and long-abandoned libraries. It doesn’t do this maliciously. It does it because it cannot tell the difference between good Java and archaeological Java. It has no connection to vulnerability databases or project activity logs. It predicts, rather than understands.

The Penny Drops

This is the moment we realise that the centre of all this isn’t innovation. It is maintenance.

Maintenance is the unglamorous foundation of modern Java development, and it has always been economically fragile. The world was built on software nobody paid for, and we are now paying for that decision through outages, incidents, forced migrations, emergency rewrites, and compliance demands. The cost of modernisation projects is evidence enough. The number of failed rewrites is further proof. Security debt and technical debt have fused into something heavier and more consequential: existential debt.

As a result, the Java ecosystem is shifting. Many organisations are moving towards curated internal repositories in which dependencies are treated as approved financial assets. Others are adopting verified component catalogues that restrict development to a smaller, safer universe of libraries. Foundations and ecosystem groups are increasing investment in auditing and supporting critical projects. All of these are positive developments. But they do not solve the final, most stubborn problem.

Fines under the CRA can reach up to 15 million euros. NIS2 stretches into the millions as well and carries personal liability for executives. “We didn’t know” is no longer a defence. “Upstream didn’t fix it” is no longer an excuse.

The laws do not assume open source is perfectly maintained. They think the opposite. And they are now beginning to force organisations to plan accordingly.

The Long Tail Remains.

Those forgotten Java libraries. Those abandoned frameworks. The components that quietly keep systems running even though their maintainers have long since left the scene. These are now more valuable to attackers than they ever were to their creators.

This is where extended maintenance becomes not a product category but a reality of the world we now inhabit. Some organisations will choose commercial providers. Some will build internal stewardship teams. Some will rely on foundations. The shape doesn’t matter. The acknowledgement does. We can no longer allow a dependency to become nobody’s responsibility.

We have left behind the era when “it still works” was enough. We have left behind the era when “upstream might fix it” was a comfortable bet. And we are rapidly leaving behind the era when “we can always fork it” is a viable path.

Conclusion

What we have now is a recognition that maintenance is not a courtesy, but a legal requirement, an operational necessity, and a strategic function. The Java supply chain we actually have is older, more fragile, and more interconnected than anyone wanted to admit. But seeing it clearly is not a sign of decline. It is the first step toward resilience.

Because at the bottom of every CVE, every emergency patch cycle, every regulator's expectation, every SCA alert, lies a truth so simple it should never have been ignored:

You cannot secure what nobody maintains. And now, finally, the world understands why.

Reference Table

GitHub Maintainer Statistics

<https://octoverse.github.com/2022/repositories/>

CHAOSS Bus Factor Research

<https://chaoss.community/metrics/>

Sonatype State of the Software Supply Chain Report

<https://www.sonatype.com/state-of-the-software-supply-chain>

European Union Requirements

Legislation	Claim Backed Up	Official Link / Key Resource
Cyber Resilience Act (CRA)	Requires SBOMs, vulnerability handling, duty of care, long-term security updates, and imposes fines up to €15 million.	Official Text (Regulation (EU) 2024/2847) (This is the final published regulation.)
NIS2 Directive	Ties executive responsibility to resilience, mandates risk management measures (including supply chain security), and requires incident reporting.	Official Text (Directive (EU) 2022/2555) (See Article 21 for cybersecurity risk-management measures, including supply chain security, and Article 33 on penalties and executive liability.)
EU AI Act	Requires traceability not only of code but of training data and model lineage, extending the supply chain concept.	The EU AI Act's final text is extensive, but the most relevant links would be to the adopted version: Final EU AI Act Text (This link provides context for the legislative process leading to the final Act.)

United States Requirements

Mandate	Claim Backed Up	Official Link / Key Resource
Executive Order 14028 (Improving the Nation's Cybersecurity)	Mandates baseline security standards, rigorous provenance checks, signed artifacts, and the use of the NIST SSDF for software sold to the US government.	Executive Order 14028: Improving the Nation's Cybersecurity
NIST Secure Software Development Framework (SSDF)	Defines practices for secure development, which is required for compliance with EO 14028 procurement rules.	NIST SP 800-218 (SSDF) Version 1.1 (This PDF details the framework and its practices.)
NIST Software Bill of Materials (SBOM)	Defines the "NIST-defined minimum elements" for SBOMs required by US procurement rules.	NIST Guidance on the Minimum Elements for a Software Bill of Materials (SBOM)

-> **Back to the Table of Content**



#JAVAPRO #JVMLANGUAGE

Kotlin Kontra Java – Part 1 – Ecosystem

When you start a new project on the JVM, should you pick Java or Kotlin?

Kotlin is used by Amazon, Atlassian, Duolingo, Google, JetBrains, Meta, Netflix, Uber and many more. Java is used by, well, „more than 3 billion devices“. Which one provides the biggest benefit, which one can give us an edge, which one should we pick?

After 20 years of building systems in Java and 8 years of doing the same in Kotlin, I've seen Kotlin earn its reputation as a natural evolution of Java. Rightfully so. Much of what we need can be expressed very cleanly with little verbosity.

But a language is not just its syntax. It's not just how well we can model business problems in the language, or how terse the written code is, or if we have to write semicolons. It's also about the tooling around it, who we can ask for help, and the libraries we can use. In short, a programming language is also about the **ecosystem and tooling**.

This article will compare the ecosystem of Java and Kotlin. In part two we will look into the multi-platform capabilities that both languages provide. In part three, we will dive deep into their modelling capabilities and give a verdict on which language to pick in 2026.

Author:



Richard Gross is the head of software archeology at MaibornWolff. He drives forward topics such as software modernisation and technical excellence. As a team lead and architect, he helps teams to achieve a return on investment as early and continuously as possible. To achieve this, he coaches teams in code quality, test automation and collective ownership. His technical focus is on hexagonal architectures, TestDSLs, hypermedia APIs and the expressive and unambiguous modelling of the domain as code. Richard shares his knowledge through active pair programming, as well as training, technical articles and presentations at international conferences. He has also long been involved in the open source project CodeCharta, which enables non-developers to gain an understanding of the quality of their software.

<https://www.linkedin.com/in/richargh>

Ecosystem Evaluation Criteria

Programming languages ideally have:

1. **Expected longevity** through pace and strong backward compatibility
2. **Developer availability** and developer interest
3. **Quality IDE support**
4. **Good Compile and Execution Performance**
5. **Mature libraries and frameworks**

Let's break down each point.

Expected Longevity

One important way for programming languages to achieve longevity is by keeping pace with how we model software — e.g. adding new syntax/libraries/tools to make common cases simple to express — and at the same time ensuring backwards compatibility to how we used to model software. This is not about a fast release cadence but about a release cadence that is fast enough.

Java

It is safe to say that **Java** — developed by Oracle — has longevity. Version 1 was released in 1996. This year will see the release of version 26 and 27. Major versions are released every six months. These releases are deliberately conservative in how much is changed however. New features go through multiple rounds of previews (or incubation) before being finalized. A lot of important features have been added over the years. This language is not going away any time soon.

Additionally, the maintainers have taken great strides to ensure backwards-compatibility with little to no breaking changes. As an example, even libraries that use internal JDK functions, that were never meant for the public, can still be used. The maintainers have just changed the default and we have to opt-in to that behavior via flags like `--add-opens`. When features are removed (e.g. CORBA, Nashorn JavaScript Engine, see A visual chronicle of the JDK's journey), it is only done when viable alternatives have been well-established. We can safely upgrade on our own schedule without having to rush to adapt to every new version.

A lot of companies do so and follow a slow upgrade path. New Relics State of Java Ecosystem 2024 suggests that 28,8% still use JDK 8 (released 2014), 32,9% use JDK 11 (2018) and only 35,4% use Java 17 (2021).

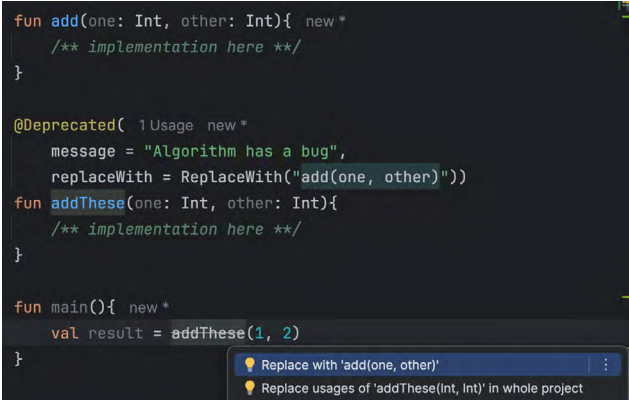
Kotlin

By comparison, **Kotlin** — developed by JetBrains — is only 10 years old. Version 1 was released in 2016 and version 2.3 and 2.4 will come this year. Major versions are released every six months, similar to Java, and tooling releases three months after the major release. Many of the features Kotlin already has are things that Java is working towards, albeit in different form. In recent years Kotlin has started to be a bit more conservative with features

however and stabilizes only after multiple previews (experimental → alpha → beta). Google has given Kotlin strong support by making it the default programming language for Android in 2019. Google, JetBrains, Meta, Gradle, Uber, Kotzilla and Block are also members of the Kotlin Foundation, which “protects, promotes and advances the development of the Kotlin programming language.” This language is also not going away any time soon.

Kotlin is a bit different than Java though. It can compile to Java bytecode (all targets between Java 8 and 25 are supported) but does not bring a virtual machine that can execute said bytecode (if you discount multi-platform). This has consequences on their release process. They essentially treat their language like a compiler and only the newest version gets bugfixes. If users want new language features, new standard library functions or target newer JDKs, they have to update.

JetBrains has declared in their Kotlin evolution principles that they intend to make updates comfortable. They will announce removals (breaking changes) well in advance, mark things as deprecated and provide automated migration tools. Personally, I have seldom experienced breaking changes, and when they occur they are announced with a deprecation warning and an automated migration path. Kotlin has a `@Deprecated(replaceWith=“...”)` annotation that makes migrations of standard library methods painless. When a library method needs to be replaced, the `replaceMe` parameter tells my JetBrains IDE what to replace it with. Very comfortable.



```
fun add(one: Int, other: Int){ new *
    /** implementation here **/
}

@Deprecated( 1Usage new *
    message = "Algorithm has a bug",
    replaceWith = ReplaceWith("add(one, other)")
)
fun addThese(one: Int, other: Int){
    /** implementation here **/
}

fun main(){ new *
    val result = addThese(1, 2)
}
```

Replace with 'add(one, other)'
Replace usages of 'addThese(Int, Int)' in whole project

Kotlin replaceWith inside @Deprecated provides automatic migration for one occurrence or all

Verdict

In summary, both languages have similar longevity expectations. Java has stronger backward-compatibility guarantees, which is important given it is much older and has a wider install base. Kotlin, however, benefits from good IDE-integration that makes migrations almost painless.

Developer Availability and Interest

Does the programming language have enough developers, do they have significant experience and is there enough interest that new developers will show up? To answer these questions, we consider three widely cited surveys that together cover activity, interest, and actual usage:

- Redmonk Survey, which measures Github PullRequest and StackOverflow *activity*
- IEEE Spectrum Survey, which tries to measure *interest* by search queries, job listings etc.
- StackOverflow Survey, which asks if the developers have done *extensive work* in the language.

The rankings show Java in the Top8, Kotlin in the top 15. Rust, released 2015, is included so we have another relatively recent language to use as a benchmark for how “new” ecosystems can still attract talent.

Ranking	GitHub PR + SO Activity	IEEE Spectrum Interest	StackOverflow Extensive Work
1	JavaScript	Python (1.0)	JavaScript (69%)
2	Python	Java (0,50)	...
3	Java	...	...
...	...	...	...
8	...	...	Java (30%)
...	...	...	...
13	...	Kotlin (0,13)	...
14	Kotlin	Rust (0,13)	Rust (15%)
15	...	...	Kotlin (12%)
...	...	...	...
19	Rust	...	...

If we look at JetBrains Developer Ecosystem Data Playground we see these rankings reflected in the data. There are about 20,8 million people that do some form of coding or programming. About 7,1 million people say Java is their primary language and about 2 million say Kotlin is theirs.

Verdict

The results are encouraging. We can pick either language and find developers who use it, have interest and have done extensive work. In my experience the syntax of Kotlin is so familiar to Java (and JavaScript) developers that onboarding them into a new project is a quick process measured in weeks not months.

Quality IDE Support

Java

Java has very good IDE support. It's supported by IntelliJ, Eclipse and NetBeans out of the Box. Alternate editors (like Visual Studio Code or Vim/NeoVim) can get language support via a language server. One is maintained by the Eclipse Foundation, Red Hat and Microsoft. Another is maintained by Apache Netbeans. Red Hat also provides a VS Code plugin while the community provides a NeoVim Extension.

Kotlin

Kotlin is developed by JetBrains which make the excellent IntelliJ IDE (also the foundation for the free Android Studio). According to JRebel 2025 Java Developer Productivity Report 84% of Java developers use it, followed by 31% VS Code and 28% Eclipse (selecting multiple IDEs was allowed). Support in their own IDE is excellent and on-par or better than for Java.

But what about alternative IDEs? For one, JetBrains maintains an Eclipse plugin. For another they are now providing an official language server (albeit in pre-alpha), a VS Code plugin (not yet published to the Marketplace) and a VS Code Java to Kotlin converter. A community language server exists but that has been marked as deprecated in favour of the official one.

Verdict

Overall, Java users can choose from several mature, feature-rich development environments across platforms. Kotlin support is strongest in IntelliJ-based IDEs and Eclipse, while support in other editors currently relies on an official language server that is still in pre-alpha and lacks some features.

Compile Performance

To get fast feedback, we need fast compilation. Without a compiled program (i.e. bytecode) we cannot launch our tests or start the application.

Benchmarking Is Tricky

According to ad-hoc benchmarks by the mill build tool javac (the Java compiler) can compile 100k lines per second of Java code.

However, once a build tool like Gradle or Maven is involved that number goes down to 9000 respectively 6000 lines per second. One reason for that is that the javac is not kept “hot”, it is always cold-started which adds the regular Java startup overhead. Java programs are notorious for needing some time to reach peak performance and the compiler is no different. The other reason is that build tools do a lot more than the compiler including dependency management, module resolution, task/phase resolution, caching and invalidation.

Benchmark Setup

Comparing only the performance of the compiler is tricky, given that most projects use dependencies, are split into modules and have special task requirements. Within the scope of this article I will focus on comparing the speed of Gradle execution, normalized by the lines of code.

The sample Java project is Mockito (contains 63 984 Java RLoC). Mockito also contains about 2299 Kotlin RLoC, but they are deemed insignificant for this comparison and ignored. The sample Kotlin project is Exposed (97 204 Kotlin RLoC). Both are Gradle projects.

Токей is used to count real lines of code (RLoC), i.e. pure code lines. Comments and empty lines are excluded as they do not concern the compiler. The number of Gradle projects is retrieved by executing `./gradlew projects`. The dependencies of project are retrieved via `./gradle :*:dependencies`. The Gradle version and daemon JVM are determined via `./gradlew --version`.

`./gradlew classes testClasses` is executed once to download all dependencies. Afterwards the network is disabled and the following command is executed five times for both projects on the same machine:

```
./gradlew clean; time ./gradlew classes testClasses
--no-build-cache
```

The median of these five runs is then normalized by the real lines of code. After the build is successful the number of executed tasks is recorded.

Verdict

Mockito (Java) is compiled with a median of **4 294 lines/s** and Exposed (Kotlin) with a median of **5 849 lines/s**. This makes Kotlin appear about 36% faster than Java for these two specific projects. I do not think this can be generalized as "Kotlin can compile faster than Java". Kotlin is a more complex language and there are lots of things the compiler has to do to fit inside the JVM bytecode. The Java compiler has also been refined for much longer, so I find it doubtful that the Kotlin compiler is that much faster. It's also noteworthy that both median compilation times are slower than what was benchmarked by the mill maintainers, but that can be attributed to different hardware setups. It would be interesting to do a deep-dive with many more projects, perhaps even bypassing Gradle completely and just using `javac` and `kotlinc` directly.

Nevertheless it seems like the Kotlin compiler rewrite that was delivered with Kotlin 2.0 was a success. JetBrains announced it would bring up to up to 94% compilation speed gains compared to 1.9.

In summary I would say that both Java and Kotlin provide comparable compilation speed, but our build tools still have room to grow.

Execution Performance

Bound To The Same JVM

Since both Java and Kotlin run on the same JVM, their runtime performance is also tied to said machine. Which is good and bad.

Bad because the same **startup** (time to first useful piece of work), **warmup** (time to reach peak performance), **scalability** (throughput per resource) and **memory** consumption will affect both languages at execution time.

Good because the improvements that the JVM is doing in regards to startup (e.g. CRaC), warmup (e.g. AOT caching), scalability (e.g.

Z Garbage Collector, Shenandoah, Virtual Threads, Integrity by Default) and memory consumption (e.g. Valhalla) will benefit both languages.

Kotlin can of course improve on that with their compiler but only up to the limits of the bytecode. For example Kotlin added Coroutines (Kotlin 1.3, Oct 2018), that increase the throughput of I/O-bound applications, long before Java added Virtual Threads (JDK 21, Sep 2023). But these could not circumvent thread pinning (`synchronized` methods that hold onto a thread doing I/O) until the JDK solved it (JDK 24, Mar 2025). Kotlin also provides `value classes` (Kotlin 1.5, May 2021) but these classes are only inlined at compile time. A `value class User(val rawName: String)` will just become a `val rawName: String` after compilation. The compiler cannot do anything more until the JDK stabilizes the actual value types byte code.

Kotlin can of course also worsen the performance by compiling to inefficient bytecode, so we should at least look at a couple of performance comparisons.

Execution Benchmarking Is Tricky

Most business applications are I/O-bound. They receive an Input (usually a web request), do some light processing, write **Output** to a database, and then **Output** a response to the web request. This is exactly the situation that reactive streams (JDK 9, 2017), Coroutines and Virtual Threads are trying to solve. But that is not about performance but efficiency. Any I/O-operation means that our application logic is waiting on the result and not doing anything. Reactive Streams/Coroutines/Virtual Threads solve this by using the thread for another operation until the I/O-operation returns a result. Efficient.

In some cases however our application can be CPU-bound. Encoding/decoding of information or multiplying huge matrices are examples of this.

When writing such a CPU-bound code we then often have to make the trade-off between speed and readability. The Java-exclusive one billion row challenge 1BRC shows this rather nicely. We have to read a `.txt` file with one billion rows of temperature measurements per weather station and calculate the min/mean/max for each weather station. A version that takes about 13 seconds to complete, has 11 functions and comprises about 250 lines of code. It is quite readable. The winner of the challenge however takes about 2 seconds, has 23 functions and 435 lines of code, and manipulates its own memory via `sun.misc.Unsafe`. It is much less readable.

So when we look at performance benchmarks, note that each result might be made more performant by creating less readable code. All these benchmarks give us is a rough relation between the execution time of these languages.

Benchmarks

Many language benchmarks exist including the Programming Language and compiler benchmarks, the benchmarks game and

the energy efficiency benchmark. The one used here is `kostya/` benchmarks because the code is “written as the average software developer would write it”:

- The algorithms are implemented as cited in public sources;
- The libraries are used as described in the tutorials, documentation and examples;
- The used data structures are idiomatic.

A couple of languages beyond Java and Kotlin are included for reference—including Scala, which also runs on the JVM. ‘Staged’ results are excluded because that is just for testing purposes and not indicative of real-world performance.

bench.b Execute Brainf***	Base64 encode/ decode	Matmul Multiply matrixes
1,008s Rust	0,635s NodeJs	3,034s C/gcc
1,112s C/gcc	0,842s Rust	3,057s Java
1,191s Java	1,117s C/gcc	3,067s Rust
1,224s Kotlin/JVM	1,435s Java	3,153s Go
1,256s Go	1,561 Kotlin/JVM	3,160s Kotlin
1,280s C#/.NET Core	1,592s Go	3,212s NodeJs
...	1,594s Scala	3,243s Scala
2,379s Scala	...	3,263s Python/pypy
2,954s NodeJs	2,374s C#/.NET Core	...
9,398s Python/ pypy	3,534s Python/ pypy	4,506s C#/.NET Core

Verdict

Java and Kotlin provide about the same performance in these benchmarks. In the three benchmarks Java is leading by about 0,1 seconds. For most business applications, maybe all, that difference is negligible.

Mature Libraries and Frameworks

A mature ecosystem has a plethora of libraries and frameworks, giant shoulders we can build upon. Java has a rich catalog of around 800.000 packages for a diverse topics such as web, auth, database, test, serialization, validation and mapping. Kotlin on the JVM has all of those as well. Every Java library can be used in Kotlin, because Kotlin put a heavy emphasis on interop (as we will see in part 3).

A `java.util.List` in Java is a `kotlin.collections.List` in Kotlin. A `java.util.Map` in Java is a `kotlin.collections.Map`. No conversion necessary. This is different from other JVM languages like Scala or Clojure where the standard libraries are not compatible but need to be converted.

Frameworks like Quarkus and Spring provide first-class support for Kotlin which means a lot of testing on their part. It also means convenience methods to make the framework feel more idiomatic in Kotlin, support for Kotlin coroutines and using `JSpecify @Nullable` and `@NotNull` to mark values that can be null or can never be null. The latter means that Kotlin's Null safety also works with Java libraries.

Verdict

Kotlin can use exactly the same mature catalogue of libraries and frameworks that Java has. In addition Kotlin has a growing collection of multi-platform libraries that we will see in part 2.

Ecosystem Summary

Should we pick Java or Kotlin? If we look at the ecosystem we get the following table:

Criteria	Java	Kotlin	Comment
Longevity, Pace, Compatibility	●●○	●●○	Both are safe long-term bets. Java has a noticeably slower feature-pace. Kotlin expects developers to continuously update but has very good migration ergonomics.
Developers availability and interest	●●●	●●○	Java has more practicing developers; Kotlin provides a very good learning curve
Quality IDE Support	●●●	●●○	JetBrains which develops Kotlin, also develops the world-class IntelliJ IDE for Kotlin and Java (if that is your IDE, replace the ○ rating with a ●). Kotlin's LSP is not stable yet, which is a drawback for editor-only developers.
Compile and Runtime performance	●●○	●●○	Both languages have roughly the same compilation and runtime performance. Build tools seem to be the main bottleneck for faster compilation.
Mature Libraries and Frameworks	●●●	●●●	Java has a very large catalogue of libraries and frameworks. Kotlin can use all these libraries because of its great interoperability.

On the JVM alone, Java and Kotlin are similar in expected longevity, mature libraries, compile and runtime performance. Java currently wins in talent availability and breadth of Editor-support; Kotlin wins with pace, ergonomics and migration tooling, especially in JetBrains IDEs. Kotlin pulls far ahead for teams that are stuck on JDK 8 as it provides modern features with familiar syntax to a 12 year old runtime.

Where Kotlin clearly differentiates itself is in multi-platform development as we will see in part 2. In this part we will also look at the multi-platform libraries and frameworks.

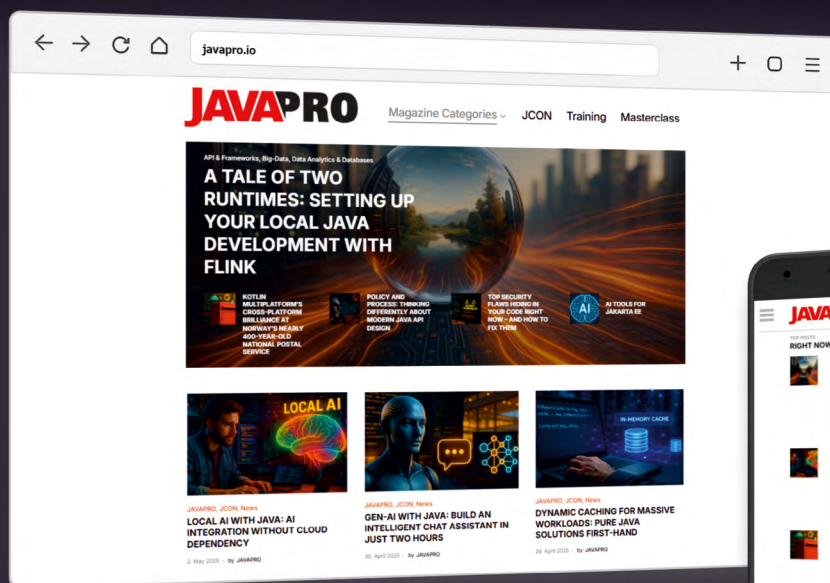
In part 3 we will look at how well we can model business problems with both languages and give a final verdict.

The full series can be found online at www.javapro.io

-> Back to the Table of Content

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!

trusted by



**BEST
{BUDDYS}**
IN CODE



Arrange a free discovery call here!

www.xdev.software

XDEV

Vector Search with Pure Java

No External Database. Extremely fast. No Compromises.

NEW:

EclipseStore 4 - Java Embedded Vector Database

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

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

Project: www.eclipsestore.io

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

NEW:

Eclipse Data Grid 4 - Distributed In-Memory Vector Search

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

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

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

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