Native Benchmark 2025 – Spring / Quarkus / Go / Rust

Ilyes Mehand Ouanezar
DevOps Engineer intern
TELECOM Nancy
Nancy, France
ilyes.ouanezar@telecomnancy.eu

***Abstract*—This paper presents a performance benchmark of modern backend technologies, comparing both JVM-based and natively compiled stacks, including Java (Spring Boot, Quarkus), Go, and Rust. Using a standardized REST API and a consistent testing methodology based on Docker and k6, we evaluate key metrics such as startup time, compilation time, executable size, request latency, CPU and memory usage. Our results reveal significant differences in performance characteristics between stacks, with Rust demonstrating exceptional speed and efficiency, and Go offering a strong balance of simplicity and performance. These insights can inform technology choices for developers and organizations seeking optimal backend performance.**

***Index Terms*—Native compilation, Benchmark, Java, Go, Rust**

I. Introduction

In recent years, the rise of native compilation and lightweight runtime environments has reshaped how backend services are architected and deployed. While the Java ecosystem remains a dominant force through mature frameworks like Spring Boot and Quarkus, alternative languages such as Go and Rust offer compelling advantages in terms of startup time, resource efficiency, and binary size. Native compilation technologies like GraalVM further promise to close the performance gap between traditional JVM-based applications and fully compiled binaries.

This project benchmarks multiple backend stacks, as known as:

TABLE I: Technology Stack Overview

Stack	Type	Technology
Spring Boot	JVM	Java 21 + Spring Boot
Spring Native	Native (GraalVM)	Java 21 + Spring AOT
Quarkus	JVM	Java 21 + Quarkus
Quarkus Native	Native (GraalVM)	Java 21 + Quarkus
Go	Native	Go 1.20
Rust	Native	Rust (actix-web, etc.)

comparing their performance in both JVM mode and native compilation (where available), along with two natively compiled languages: Go and Rust. This project is hosted on GitHub and can be easily tested locally as it only requires Docker / Docker compose.

These metrics are collected with k6 tests, a shell / powershell scrip and Docker. All metrics are logged in .csv files and can be exported / analyzed / compared ...

Micronaut was initially benchmarked and then deprecated due to difficulties to quickly get a REST API that compiles both natively and not.

Our goal is to provide actionable insights on which technologies perform best for latency-sensitive, resource-constrained, or high-throughput applications.

II. Benchmarked system

This diagram shows how all services are structured and how they interact in the benchmark setup:

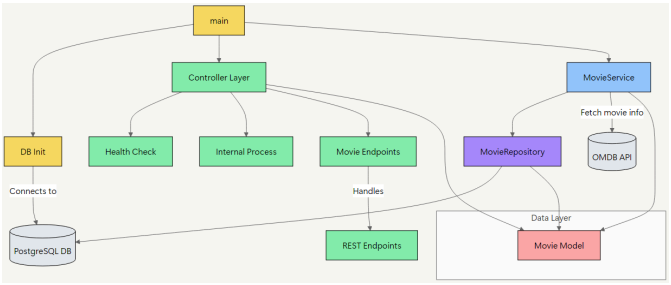


Fig. 1: Application Architecture

This diagram shows the structure of the application and how its components interact to handle requests and perform operations in the benchmark setup:

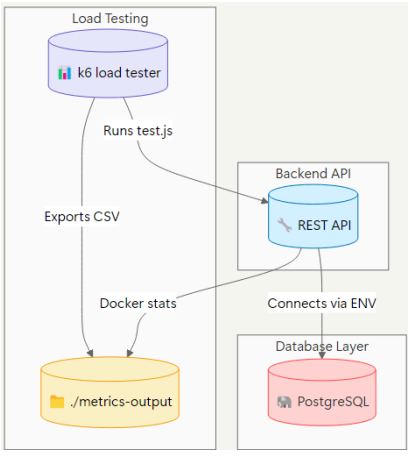


Fig. 2: Containers Architecture

A. Endpoints Tested and their purpose

TABLE II: API Endpoints Overview

Method	Function	Purpose
GET	Search for a movie by title via OMDB API call	Web request: search title in OMDB
POST	Add a movie directly to the database (JSON)	Database access (write)
GET	Read a movie from the DB by ID	Database access (read)
GET	Trigger a CPU-bound internal process	Benchmark heavy logic
GET	Check if API is alive ("pong")	Verify availability
GET	Get a random movie from DB	Database query + logic

III. Methodology

Due to an excessively high standard deviation between the measurements we run the tests 10 times for each Framework, compute their average and then compare it.

A. Metrics

Table III presents all the measured metrics in this benchmark.

B. Description of K6 Tests

The objective here is to evaluate the stability, responsiveness, and robustness of the REST endpoints by simulating user traffic and measuring performance and errors.

a) *General Configuration:*
- VUs (Virtual Users): 10
- Total iterations: 1000
- Setup timeout: 1300s Performance thresholds:
  - p(95) < 500ms for each endpoint
  - Failure rate < 1%

b) *Setup (Preparation):*
Send requests to /health until receiving "ok" or exceeding 60 attempts with a 20s pause between requests. If the API is not ready, the setup loop waits and retries automatically (up to 20 minutes).

c) *Test Groups:*
1) Healthcheck
- Endpoint: GET /health
- Verifies that the API is alive
- Checks:
  - Status 200
  - Content: "ok"

2) Create Movie
- Endpoint: POST /movies
- Sends a dynamically generated movie with a unique name
- Checks:
  - Status 200
  - The movie ID is returned

3) Get Movie by ID
- Endpoint: GET /movies/{id}
- Uses the previously created movie ID
- Verifies that the movie can be retrieved

TABLE III: Performance Metrics Overview

Metric	Description	Purpose	Tool/Method
Compilation time	Time to build code into executable or bytecode	Evaluates dev cycle and CI/CD impact	Docker
Compression time	Time to compress build with upx	Estimates packaging/distribution duration	Docker
Start up time	Time for application to start	Important for cold starts (e.g., Kubernetes)	Docker
Executable size	Final binary or executable size	Affects transfer and disk space	Docker
CPU usage/s	CPU used per second during execution	Evaluates CPU performance efficiency	script
Physical memory usage	Peak RAM usage	Detects leaks and sets memory limits	script
Memory while running	Average RAM during execution	Measures stability and efficiency	script
http_req_duration{tag:health}	Latency of /health endpoint	API health check latency	k6
http_req_duration{tag:create}	POST /movies response time	Performance of resource creation	k6
http_req_duration{tag:get_by_id}	Get by ID response time	Resource read speed	k6
http_req_duration{tag:search}	Search by title response time	Performance of search	k6
http_req_duration{tag:random}	Random movie fetch response time	Random access latency	k6
http_req_duration{tag:internal}	/internal/process response time	Internal logic performance	k6
http_req_failed	Failure rate of HTTP requests	Measures reliability (<1%)	k6
check()	Assertions on response content/status	Validates correctness	k6
iteration_duration	Full iteration execution time	Simulated user performance	k6
http_reqs	Total HTTP requests sent	Measures load generated	k6
data_received / data_sent	Data transferred during test	Analyzes bandwidth usage	k6
vus, vus_max	Virtual users active/max	Concurrency during test	k6
checks_total, checks_succeeded	Assertions performed/passed	Checks functional correctness	k6

4) Search Movie by Title
- Endpoint: GET /movies?title=Inception
- Searches for a movie by its title
- Checks:
  - Status 200
  - The movie is returned

5) Random Movie
- Endpoint: GET /movies/random
- Retrieves a random movie
- Verifies that the movie is returned correctly

6) Internal Process
- Endpoint: GET /internal/process
- Simulates an internal process
- Verifies that the process is completed (status 200)

Each group tracks request duration in a dedicated trend metric to isolate and limit the impact of timeouts on performance measurements.

These tests simulate realistic user load, measure the precise performance of each endpoint, and can be integrated into a CI/CD pipeline to detect regressions.

C. Description of Monitoring via Docker

Docker container monitoring allows tracking system resource usage during the execution of load tests. This real-time monitoring is essential for analyzing the impact of load tests on system performance.

a) *Collecting Docker Statistics:*

Using Docker to run the tests allows us to collect data on the performance of the containers hosting the api-benchmark API. We use Docker tools to monitor the following resources:
- CPU Usage: Tracks the processor load used by the container, helping to understand the impact of the tests on CPU performance.
- Memory: Memory usage by the container, which helps detect memory leaks or excessive resource usage.

These data are collected continuously, allowing us to visualize the load evolution during test execution.

b) *Exporting Data:*

The collected statistics (CPU usage and memory) are exported to a CSV file, which enables visualizing this data over a given period. This provides an overview of the resources used and helps identify trends or anomalies during the tests.

c) *Using Docker Desktop Tools:*

Docker Desktop also provides several useful metrics that are not limited to test execution. These metrics complement the overall performance monitoring of the application by providing additional information on:
- Compilation Time: The time required to transform the source code into an executable. This helps evaluate the efficiency of the build process and continuous integration (CI).
- Compression Time: The time taken to compress the application before distribution, an important factor in fast deployment cycles.
- Startup Time: The time required to start a Docker container, critical for applications that need to scale rapidly, such as in serverless architectures or with Kubernetes.
- Executable Size: The final size of the binary or executable file generated. This can impact network transfer times and the disk space required on servers.

IV. Results

A. HTTP requests

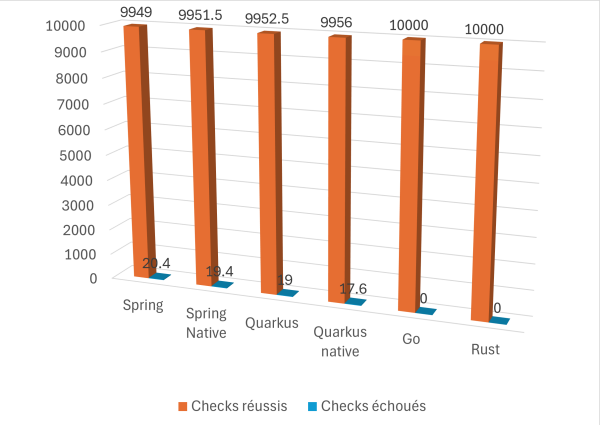


Fig. 3: HTTP requests' checks

The test results clearly show that Go and Rust deliver outstanding reliability, with 0% failed checks across the board. These natively compiled languages stand out for their consistent stability, even under heavy load. In contrast, Java-based frameworks—whether running on the JVM or compiled natively with GraalVM—exhibit a slightly higher failure rate, although still very low (around 0.2%). Among them, Quarkus Native proves to be the most stable. While the differences are minor, they highlight the advantage of natively compiled languages for systems that demand high reliability and predictable response times.

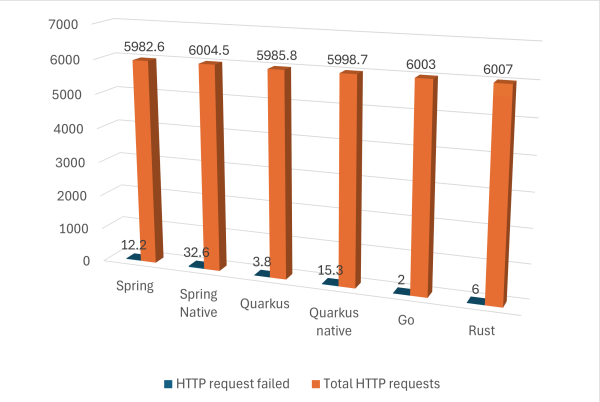


Fig. 4: HTTP requests failures

The data shows that Go and Rust once again demonstrate excellent reliability, with the lowest number of HTTP request failures—only 2 and 6 failures respectively, out of over 6000

requests. On the other hand, Spring Native experienced the most instability, with an average of 32.6 failed requests, followed by Spring Boot and Quarkus Native.

While the overall failure rates remain relatively low, these results confirm that natively compiled languages like Go and Rust offer more consistent and resilient request handling, especially under load. Conversely, some JVM-based frameworks exhibit slightly higher variability and error rates, which may impact performance in high-throughput or latency-sensitive applications.

Table IV: Rust consistently delivers the best performance across all endpoints, with sub-millisecond latencies on critical routes like /internal, /create, and /health. Go also performs well, particularly on lightweight endpoints such as /random and /search, though it lags on CPU-bound tasks (it could be easily increased with a better implementation). Java-based frameworks—both JVM and native variants—exhibit higher latencies overall, with native compilation improving startup but not consistently reducing response times. Overall, Rust stands out as the fastest and most efficient stack, followed by Go, with Java frameworks trailing in responsiveness.

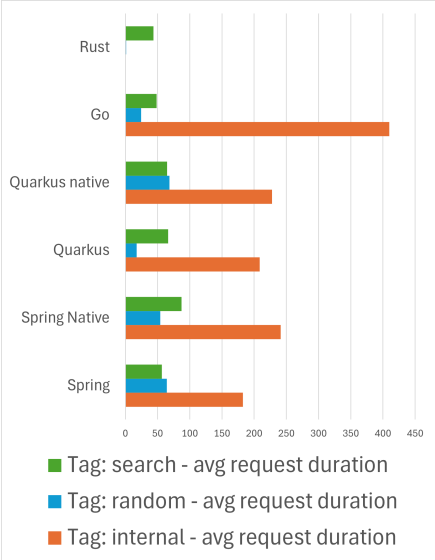
Table V: When comparing the overall average HTTP request duration, Rust significantly outperforms all other stacks with an average of just 7.89 ms, far ahead of every other technology tested. Go follows with 81.62 ms, showing decent performance but still an order of magnitude slower than Rust. Note that the gap between Rust and Go is a bit inflated by the very bad result of the Go's /internal route.

Among the Java-based frameworks:

- Spring Native has the highest average at 65.36 ms,
- Quarkus Native is the slowest of the group at 68.92 ms,
- Spring Boot and Quarkus (JVM) perform slightly better at 52.42 ms and 58.57 ms, respectively.

These results underline Rust's exceptional efficiency for handling HTTP traffic, making it ideal for latency-critical applications. The JVM stacks, while still responsive, show the expected overhead from runtime or native compilation, whereas Go offers a good balance between performance and simplicity but can't match Rust's ultra-low latency.

TABLE IV: Average HTTP Request Duration in details



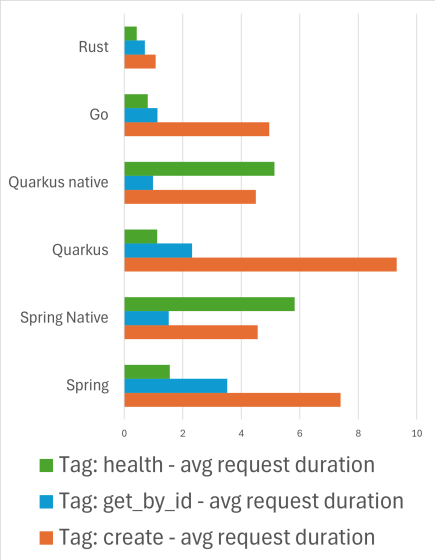
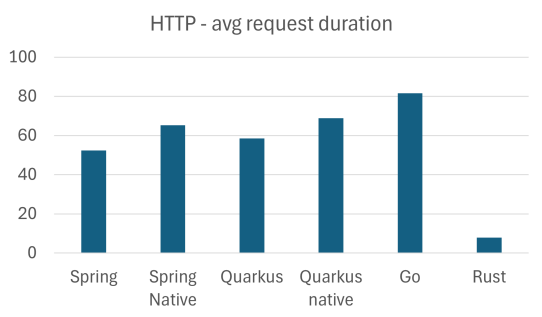
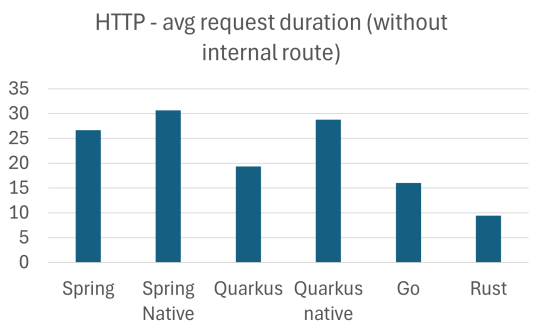


TABLE V: Average HTTP Request Duration (with and without internal route)





B. CPU / Memory

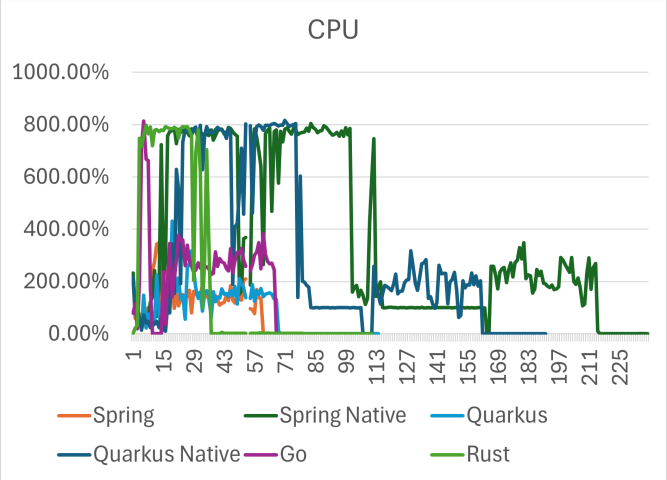


Fig. 5: CPU usage pourcentage over time

Spring and Quarkus exhibit moderate and stable CPU usage, making them well-suited for environments where energy efficiency is a priority. Go consumes slightly more but remains balanced, effectively leveraging parallelism. In contrast, Rust, Quarkus Native, and Spring Native aim for peak performance, with high CPU spikes: Rust for a short duration, Quarkus Native for a longer period, and Spring Native combining both intensity and duration.

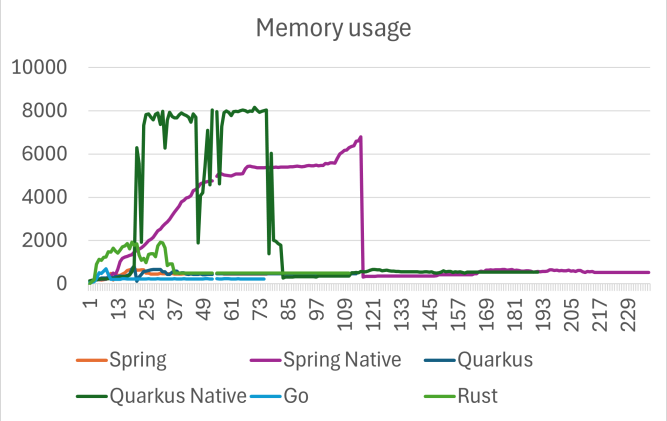


Fig. 6: Memory usage (in MB) over time

Go has the lowest and shortest RAM usage, making it highly efficient. Spring and Quarkus consume slightly more memory for a bit longer, staying within reasonable bounds. Rust uses more RAM than the others and maintains it for a similar duration. Spring Native shows a continuous memory increase, reaching nearly 7 GB over a long period before dropping sharply. Quarkus Native goes even further, peaking at 8 GB for almost as long, then also dropping to minimal usage.

Note that these graphs include CPU and Memory usage during compilation, compression (if compressed) and finally during runtime.

C. Time / Size

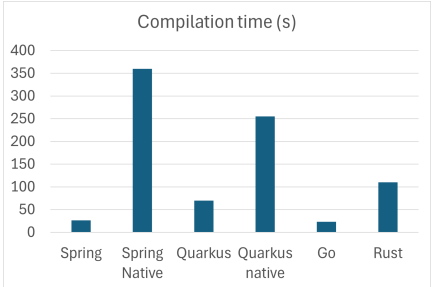
Table VI: Go has the fastest compilation time at 23 seconds, followed by Rust at 110 seconds, offering a balance of speed and powerful optimizations. Java native frameworks, such as Spring Native (360 seconds) and Quarkus Native (255 seconds), have significantly longer build times, while their JVM counterparts are slower than Go and on par or behind Rust.

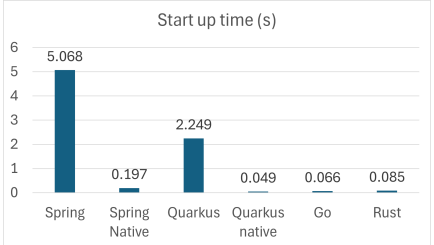
For startup times, Rust leads at 0.05 seconds, followed by Go at 0.08 seconds. Java frameworks, including Spring (5.07 seconds), Quarkus (2.25 seconds), and Quarkus Native (0.07 seconds), have notably slower startup times, impacting rapid service restarts or scaling.

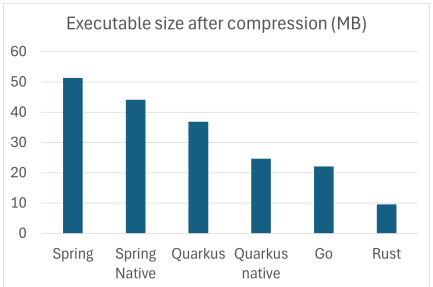
Regarding executable size, Rust (9.6 MB) and Go (22.1 MB) offer the smallest sizes. Java native frameworks, even with UPX compression, are larger, with Spring Native at 44.14 MB and Quarkus Native at 24.62 MB. Non-native Java frameworks have significantly larger sizes, with Spring at 51.3 MB and Quarkus at 36.9 MB, reflecting JVM overhead.

In summary, Go and Rust excel in fast compilation, startup, and small executable sizes, making them ideal for rapid development and low-latency applications. Java frameworks, while offering runtime speed, incur higher build and startup overheads.

TABLE VI: Comparison of Compilation Time, Startup Time, and Executable Size Across Technologies







V. Discussion

The results highlight substantial differences between the technologies studied, both in terms of raw performance and resource consumption, as well as compilation time. Several key insights can be drawn:

A. HTTP Performance and Latency

Rust clearly stands out as the most performant technology in terms of latency, with average response times below 10 ms—even on complex routes. Go follows with decent, though less consistent, latency, particularly on CPU-bound routes. Java frameworks, whether native (via GraalVM) or JVM-based, show higher latencies, confirming the overhead introduced by the virtual machine or the complexity of ahead-of-time (AOT) compilation.

However, the performance gap between JVM and native modes in the Java ecosystem is often smaller than expected, especially with Spring and Quarkus. This suggests that native compilation with GraalVM does not always deliver a significant latency advantage under all types of workloads.

B. Stability and Reliability

Go and Rust demonstrate near-perfect reliability, with virtually no HTTP failure rates, making them strong candidates for critical environments requiring predictable behavior under heavy load. In contrast, Spring Native and Quarkus Native show slightly higher failure rates (up to 0.5%). These errors remain low but can accumulate in high-throughput systems.

C. Compilation Time and Executable Size

Compilation times vary greatly between stacks. Go compiles in under 30 seconds, Rust takes around 2 minutes, while native Java solutions can require several minutes (up to 6 minutes for Spring Native). This can negatively impact the developer experience, CI/CD cycles, and responsiveness in cloud-native environments.

Executable sizes follow a similar trend: Rust and Go produce compact binaries (9–22 MB), while Java native binaries are significantly larger (up to 50 MB), though still much lighter than their equivalent .jar files. This has direct implications for distribution, deployment speed, and scalability in microservices or serverless functions.

D. CPU and Memory Usage

Rust utilizes the CPU intensively but only for short bursts, reflecting fast and optimized execution. Go shows a balanced profile in terms of CPU and memory usage. Java stacks—especially in JVM mode—exhibit more stable but generally less efficient runtime resource usage.

The native versions of Spring and Quarkus (compiled via GraalVM) are characterized by very high CPU and memory usage during the compilation phase (up to 8 GB of RAM), which can be a limiting factor in some build environments. However, this one-time cost is largely offset at runtime: native applications start much faster, consume less memory, and occupy less disk space than their JVM counterparts. This trade-off is especially relevant for use cases where startup time, container footprint, or runtime resource constraints are critical (e.g., serverless architectures, massively deployed microservices).

E. Use Cases and Recommendations

Rust is particularly suited for systems that require fine-grained control over resources, real-time performance, and maximum efficiency—such as embedded systems, critical APIs, or high-frequency services.

Go offers an excellent balance between simplicity, performance, and development speed. It is well-suited for microservices, cloud-native applications, and network-intensive projects.

Spring Boot / Quarkus in JVM mode remain relevant for enterprise projects, thanks to their rich ecosystems, high productivity, and compatibility with existing Java tooling. However, their runtime overhead makes them less competitive in highly constrained environments.

GraalVM Native (Spring Native, Quarkus Native) is promising but still maturing: the startup time improvements are undeniable, but the complexity of implementation and variability in performance must be carefully assessed on a case-by-case basis.

F. About UPX Compression

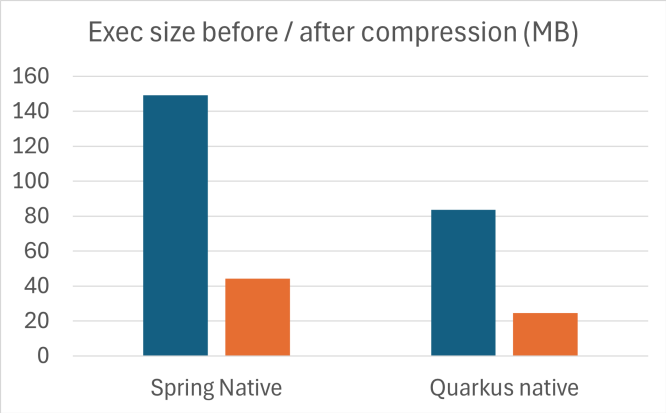


Fig. 7: Memory usage (in MB) over time

The JVM-based frameworks (Spring and Quarkus) have much larger sizes before compression, with Spring being the largest. However, Quarkus Native and Spring Native benefit from UPX compression, which reduces their sizes significantly. Quarkus Native (24.62 MB) and Spring Native (44.14 MB) are the compressed sizes, showing the effectiveness of UPX in optimizing native executables. The compression time is relatively quick for Quarkus Native (60 seconds) but takes longer for Spring Native (120 seconds), reflecting the additional overhead in size.

VI. Conclusion

This comparative study highlights that no single technology emerges as a universal winner; each stack presents distinct advantages and trade-offs. Rust delivers unmatched execution performance and reliability, making it ideal for performance-critical environments. Go provides a pragmatic

balance of speed, simplicity, and resource efficiency, positioning it as a strong choice for scalable cloud-native services.

Java-based frameworks like Spring Boot and Quarkus remain highly valuable in enterprise contexts due to their maturity and ecosystem, although their JVM-based runtime introduces a performance overhead. GraalVM Native compilation offers an exciting path forward for Java, particularly for lightweight, fast-starting applications—provided that teams can accommodate the higher build complexity and resource requirements.

Ultimately, the choice of stack should be driven by the specific needs of the project—whether it prioritizes startup time, developer productivity, runtime performance, or ecosystem compatibility. As native compilation technologies continue to evolve, they may shift the balance further, but today's decisions still require a careful case-by-case evaluation.

VII. Appendix