report

August 17, 2025

1 Intro

The tests were ran on `Debian 6.1.140-1 x86_64 GNU/Linux` using `Oracle GraalVM 21.0.7` for Java. The environment featured an Intel(R) Xeon(R) CPU @ 2.20GHz (8 cores) and 32GB RAM.

The configuration files used for benchmarking are located in the **/report/configs/** folder.

The JVM options used for the benchmarks are specified in the configuration files in **/report/configs/**. For example: - For JVM with G1GC: `-Xmx2g -Xss1m -XX:+UseG1GC` - This means the Java process will use up to 2 gigabytes of memory for the heap, set each thread's stack size to 1 megabyte, and use the G1 garbage collector to manage memory. - For JVM Shenandoah: `-Xmx2g -Xss1m -XX:+UseShenandoahGC` - This means the Java process will use up to 2 gigabytes of memory for the heap, set each thread's stack size to 1 megabyte, and use the Shenandoah garbage collector, which is designed for low pause times. - For Native: `-H:MaxHeapSize=2g -H:StackSize=1m` - This means the maximum amount of memory the application can use for its heap is set to 2 gigabytes, and each thread will have a stack size of 1 megabyte. These options help control memory usage and prevent the application from using more resources than intended when running as a native executable.

These options control the maximum heap size, thread stack size, and the garbage collector used during the benchmarks.

All dependencies required to run the Jupyter notebook are listed in requirements.txt.

To ensure accurate results, we use WRK2 as the workload generator with the following key parameters:

```
# output copied from wrk2 help page
  Options:
     -c, --connections <N>  Connections to keep open
     -d, --duration    <T>  Duration of test
     -t, --threads     <N>  Number of threads to use
     ...
     -R, --rate        <T>  work rate (throughput)
                            in requests/sec (total)
                            [Required Parameter]
```

- **Connections**: Set to 100, which ensures enough concurrent connections without overwhelming the system.
- **Duration**: Set to 30 seconds to provide a stable testing window.

- **Threads**: Set to 2 to provide sufficient concurrency for the workload generator to utilize CPU resources efficiently without causing excessive context switching.
- **Rate**: Set to 2000 requests per second.

1.1 About the Benchmarks

The **Quarkus Tika** benchmark focuses on file serialization and deserialization tasks, leveraging the Tika library to process and extract metadata from files. This allows us to measure not only HTTP request/response performance but also the efficiency of handling real-world file I/O and content extraction scenarios in both JVM and native modes.

The benchmarks also include applications built with **Quarkus** and **Micronaut** frameworks. These represent modern, cloud-native Java stacks designed for fast startup and low memory usage, making them ideal for serverless and containerized environments. By comparing Quarkus and Micronaut in both JVM and native image modes, we can assess their relative strengths in terms of startup time, throughput, and resource efficiency under realistic workloads.

2 Startup and Response time

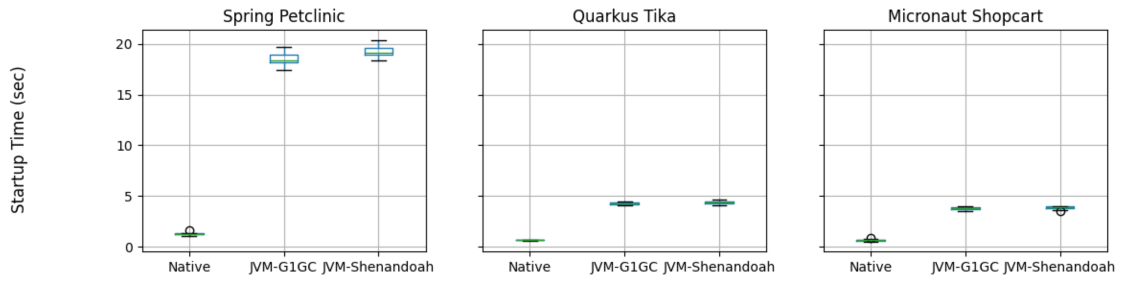
Startup time and response latency are critical metrics for evaluating serverless and microservice applications, especially in dynamic environments like cloud deployments or autoscaling systems. Fast startup ensures quicker availability of services (e.g., during cold starts), while low response time improves user experience and system efficiency.

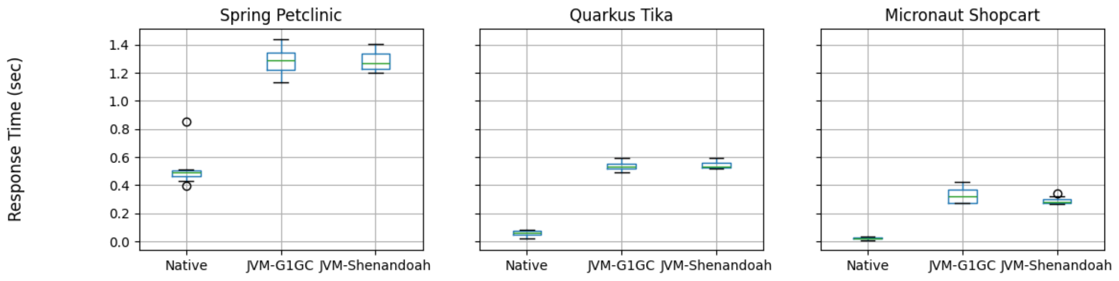
When comparing **JVM** vs. **Native Image** execution, these metrics highlight key trade-offs:

- JVM offers runtime optimizations via JIT but often has slower startup due to classloading and warm-up overhead.
- Native images (e.g., compiled with GraalVM) eliminate JIT and preinitialize application state, potentially reducing both startup and tail latency.

Hypotheses: - *Startup Time*: We expect native images to start significantly faster than JVM applications. - *Response Time*: We hypothesize that native images will exhibit lower and more consistent latency, especially during cold starts.

```
Text(0.5, 1.0, 'Micronaut Shopcart')
```





The results show that native images clearly outperform JVM execution across all applications (Spring Petclinic, Quarkus Tika, Micronaut Shopcart). Native executables deliver 5×–20× faster startup and 2×–5× lower response times, with far less variance, making performance more predictable. JVM runs are slower and more variable due to JIT warm-up and garbage collection.

Native runs are also more predictable, while JVM runs vary more due to GC behavior (G1GC vs. Shenandoah) and JIT warm-up.

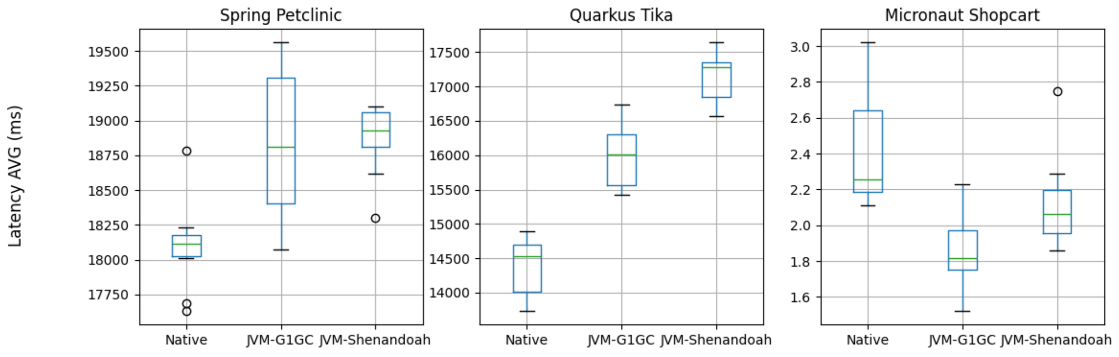
3 Latency and Throughput

Latency reflects how quickly an application handles individual requests, which directly impacts user experience and system responsiveness. **Throughput** (measured in requests per second) indicates how well the system handles load and scales under pressure. Together, these metrics give a holistic view of runtime efficiency.

In microservice and serverless environments—where services must handle bursts of traffic or maintain low tail latencies—these performance factors are critical.

Hypotheses - *Latency*: We expect GraalVM native images to offer lower and more stable request latency due to ahead-of-time optimization and reduced GC activity. - *Throughput*: We expect native images to match or slightly improve throughput compared to JVM, although in some cases the JVM's JIT may deliver marginal gains under sustained load.

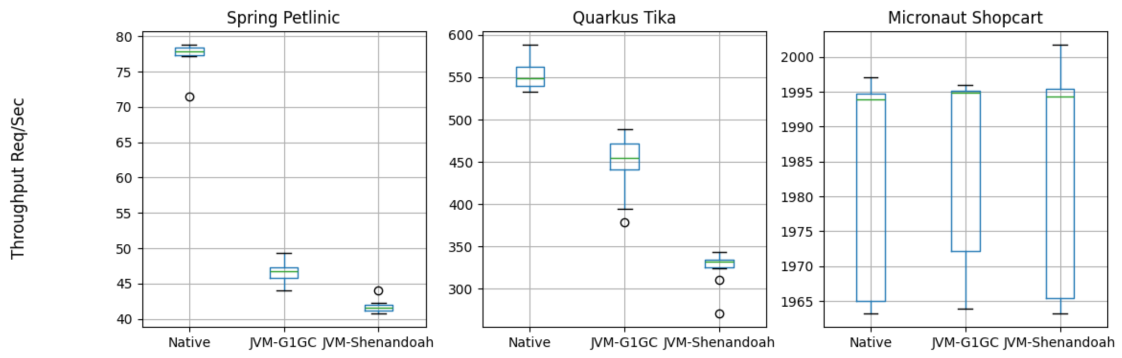
```
Text(0.5, 1.0, 'Micronaut Shopcart')
```



The results show mixed latency behavior across workloads. In Spring Petclinic, native execution achieves ~1,000 ms lower latency and tighter variance than JVM. In Quarkus Tika, however, JVM (both G1GC and Shenandoah) delivers lower latency and fewer outliers, outperforming native. For Micronaut Shopcart, differences are small, with JVM slightly ahead but all modes showing similar variance.

Overall, latency benefits of native images are workload-dependent: while native helps in Petclinic, JVM with JIT and GC optimizations (G1GC/Shenandoah) can outperform in other cases. The presence of outliers in native runs also highlights potential variability in tail latency.

```
Text(0.5, 1.0, 'Micronaut Shopcart')
```



Throughput results show workload-dependent differences between native and JVM execution.

- In Spring Petclinic, native achieves nearly double the throughput (~78 req/sec vs. ~45 req/sec on JVM), with less variability.
- In Quarkus Tika, native also leads, sustaining ~550 req/sec compared to ~450 (G1GC) and ~330 (Shenandoah).
- In Micronaut Shopcart, all modes reach the system's ceiling (~1990 req/sec), with negligible differences.

Overall, native images deliver higher throughput in Petclinic and Quarkus, while Micronaut saturates equally across execution modes.

4 99th percentile

While average latency gives a general sense of performance, it often hides performance spikes that affect real users. The 99th percentile latency captures the worst-case (tail) response times experienced by 1 in every 100 requests. This is critical for:

- User experience: In many applications, a small number of slow responses can significantly degrade perceived performance.
- Serverless and autoscaling environments: Tail latencies determine how quickly new instances can respond under load or cold starts.
- SLA compliance: Many service-level agreements are based on tail metrics rather than averages.

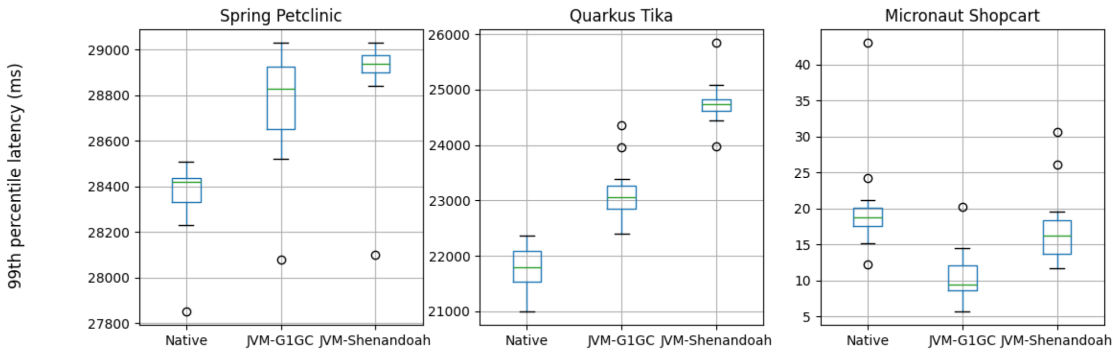
JVM-based applications may suffer from outliers due to JIT compilation, GC pauses, or thread contention—making the 99th percentile a key metric for comparison.

Hypothesis:

We hypothesize that:

- Native images will exhibit lower and more stable 99th percentile latencies compared to JVM, due to the absence of JIT warm-up and more predictable memory behavior.
- The difference will be most visible in latency-sensitive frameworks (like Micronaut and Quarkus), while more complex frameworks (like Spring Boot) may still suffer from inherent overheads regardless of execution mode.

```
Text(0.5, 1.0, 'Micronaut Shopcart')
```



The 99th-percentile latency results show mixed outcomes across workloads.

- In Spring Petclinic, native achieves slightly lower tail latency (~28,300 ms vs. ~28,800–29,000 ms on JVM), though differences are modest.
- In Quarkus Tika, native is clearly better, sustaining ~21,500 ms vs. 23,000–25,000 ms on JVM, with fewer extreme outliers.
- In Micronaut Shopcart, all modes maintain very low tail latency (~10–20 ms), though JVM occasionally produces higher spikes.

Overall, native images reduce tail latency in Petclinic and Quarkus, while Micronaut remains efficient across both execution modes.

5 RSS

Resident Set Size (RSS) represents the actual portion of memory a process occupies in RAM during execution. It's a key indicator of runtime memory footprint and is especially important in:

- Serverless and containerized environments, where memory resources are limited and billed per usage.

- Autoscaling systems, where smaller footprints allow more instances per host and faster cold starts.

- Microservices architectures, where dozens or hundreds of services may run in parallel, amplifying memory impact.

Comparing RSS between JVM and Native Image is essential because:

- The JVM includes extra overhead from the JIT compiler, metadata, and dynamic class loading.
- Native images eliminate JIT and precompile application logic, which should lead to lower and more stable memory usage.

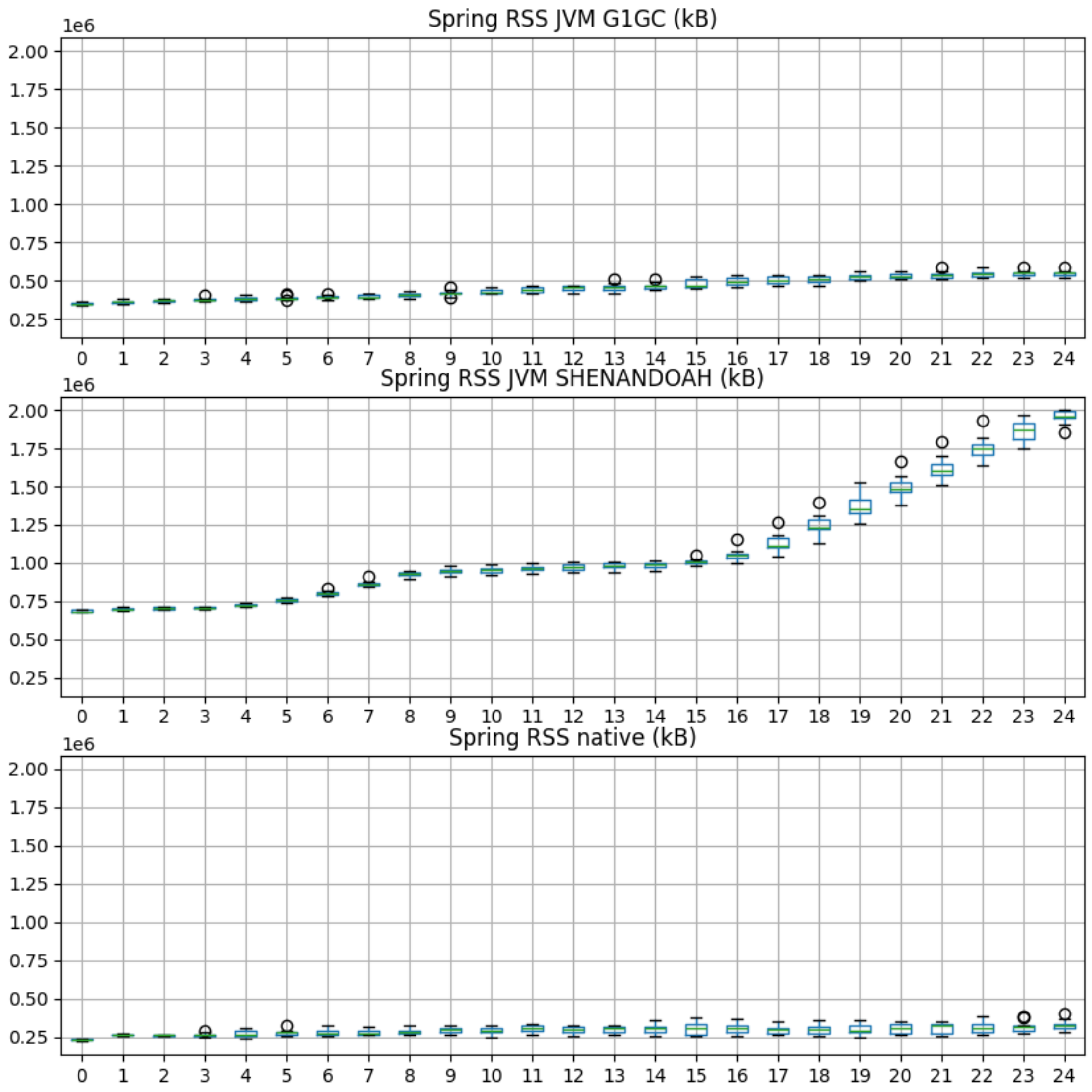
Hypothesis

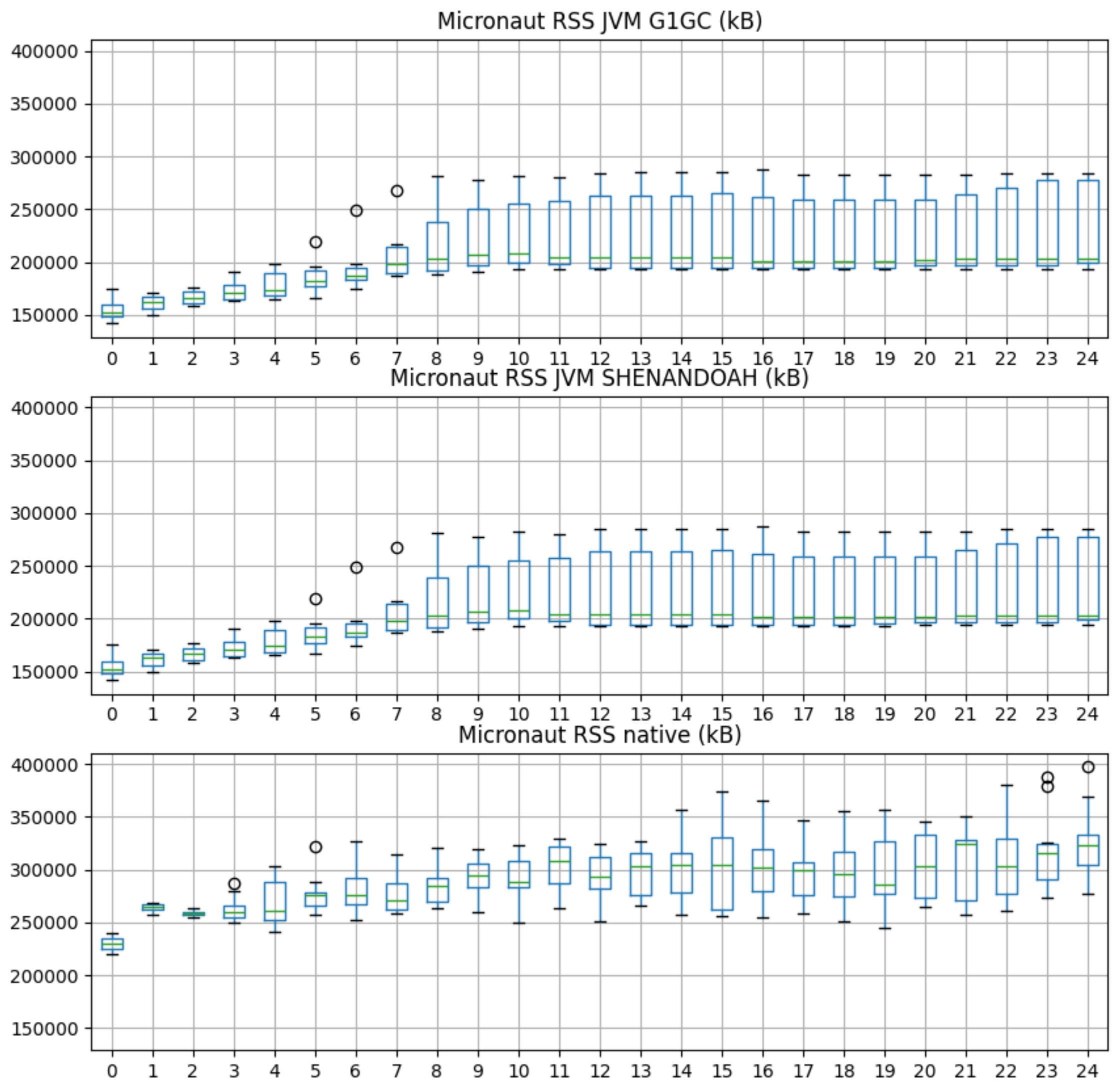
We hypothesize that:

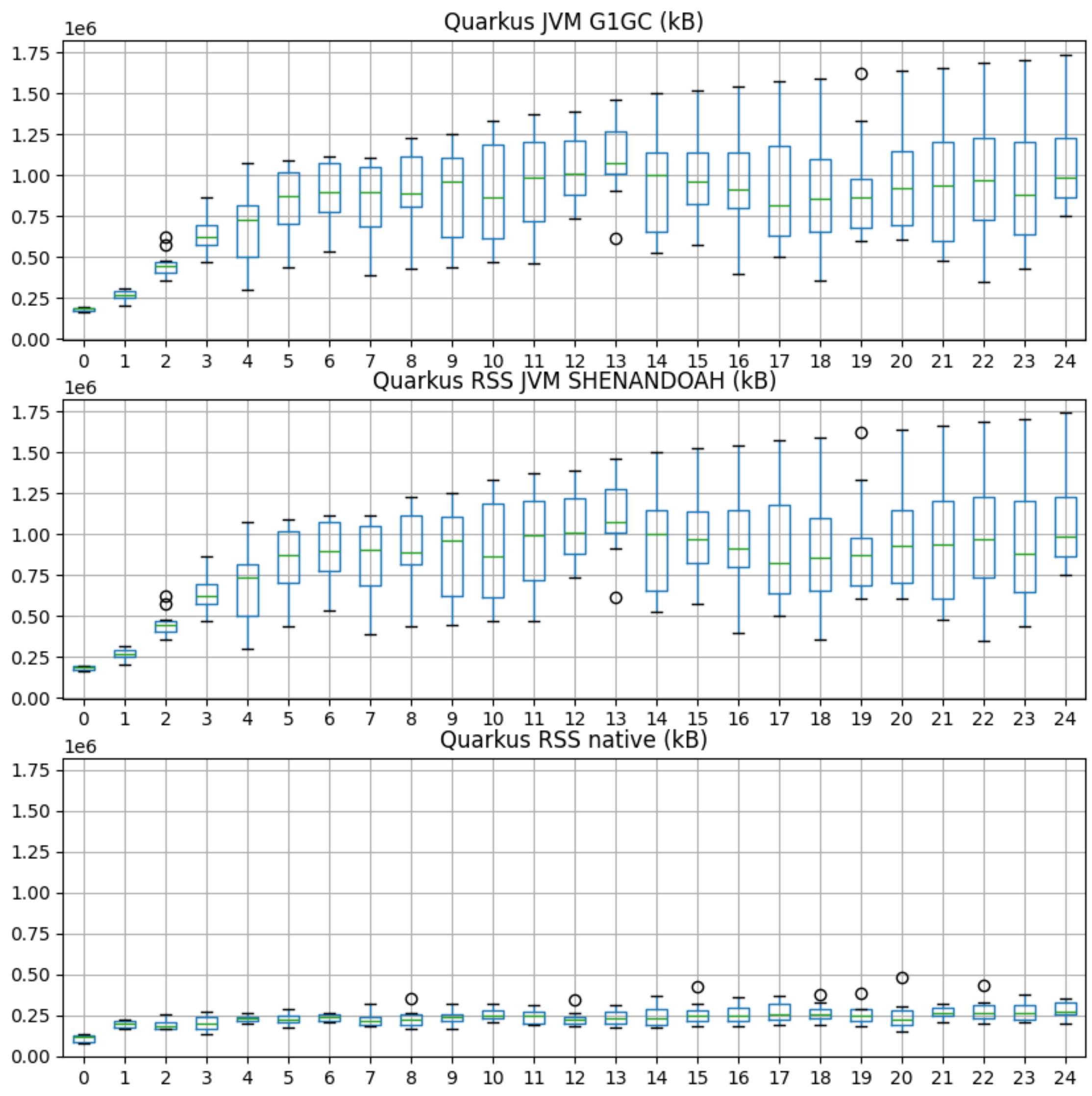
- Native images will use significantly less RSS memory than their JVM counterparts, due to reduced runtime overhead and statically initialized state.
- The difference will be more pronounced in simpler frameworks like Quarkus and Micronaut, and still noticeable in heavier frameworks like Spring Boot.

RSS is therefore a vital metric for assessing resource efficiency, which is a major factor in deployment scalability and cost in cloud environments.

```
Text(0.5, 1.0, 'Quarkus RSS native (kB)')
```


Spring RSS JVM G1GC (kB)
1e6
Spring RSS JVM SHENANDOAH (kB)
1e6
Spring RSS native (kB)
1e6


Micronaut RSS JVM G1GC (kB)
Micronaut RSS JVM SHENANDOAH (kB)
Micronaut RSS native (kB)



The RSS measurements confirm a consistent memory efficiency advantage for native images across all frameworks.

- In Spring Petclinic, JVM memory steadily increases, with Shenandoah climbing close to 2 GB and G1GC reaching ~0.5 GB, while the native footprint stays stable around 250–300 MB.
- In Micronaut Shopcart, all modes consume less memory overall, but JVM processes grow toward ~300 MB, whereas the native footprint remains slightly lower and more stable (~250 MB).
- In Quarkus Tika, the gap is most striking: JVM memory grows beyond 1.5 GB with high variance, while native execution remains consistently below 300 MB throughout.

These results highlight that native images avoid the overhead of JIT compilation, class metadata, and GC tuning structures, leading to a smaller and more predictable memory footprint. The effect is especially strong in larger workloads like Spring and Quarkus.

6 CPU Usage

CPU utilization reflects how much processing power an application consumes under load. It is a key metric for understanding runtime efficiency and is particularly important when comparing JVM and native image execution because:

- JVM applications may use more CPU due to JIT compilation, background threads (e.g., for GC and classloading), and warm-up phases.
- Native images, compiled ahead of time, eliminate JIT and often use simpler memory and thread management, potentially leading to lower CPU consumption.

By measuring CPU usage, we can determine:

- Which runtime is more efficient under sustained load
- How much overhead is introduced by the JVM's dynamic optimizations
- Whether native images deliver comparable performance with fewer CPU resources

This is especially valuable in cloud environments, where CPU usage affects autoscaling, instance density, and operating costs.

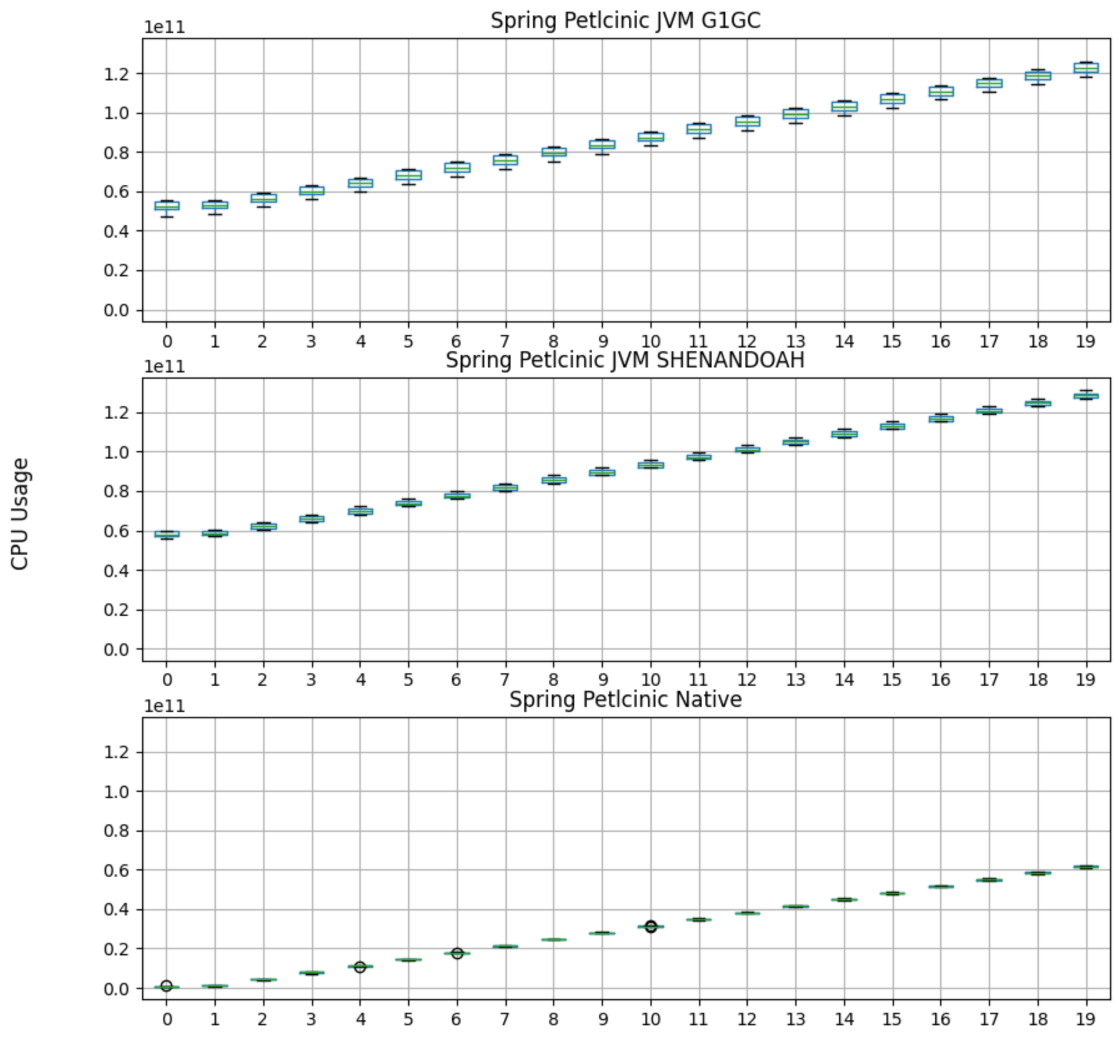
Hypothesis

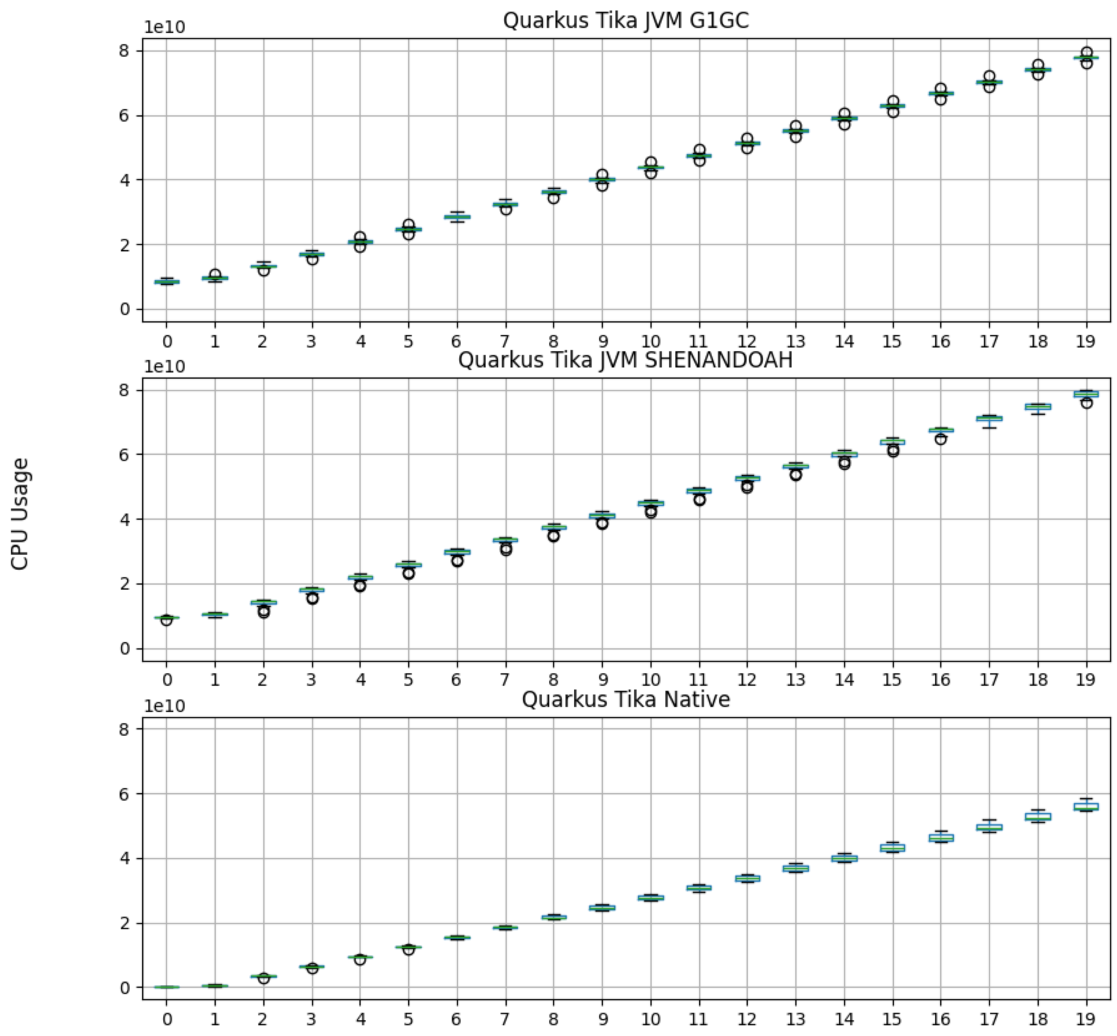
We hypothesize that:

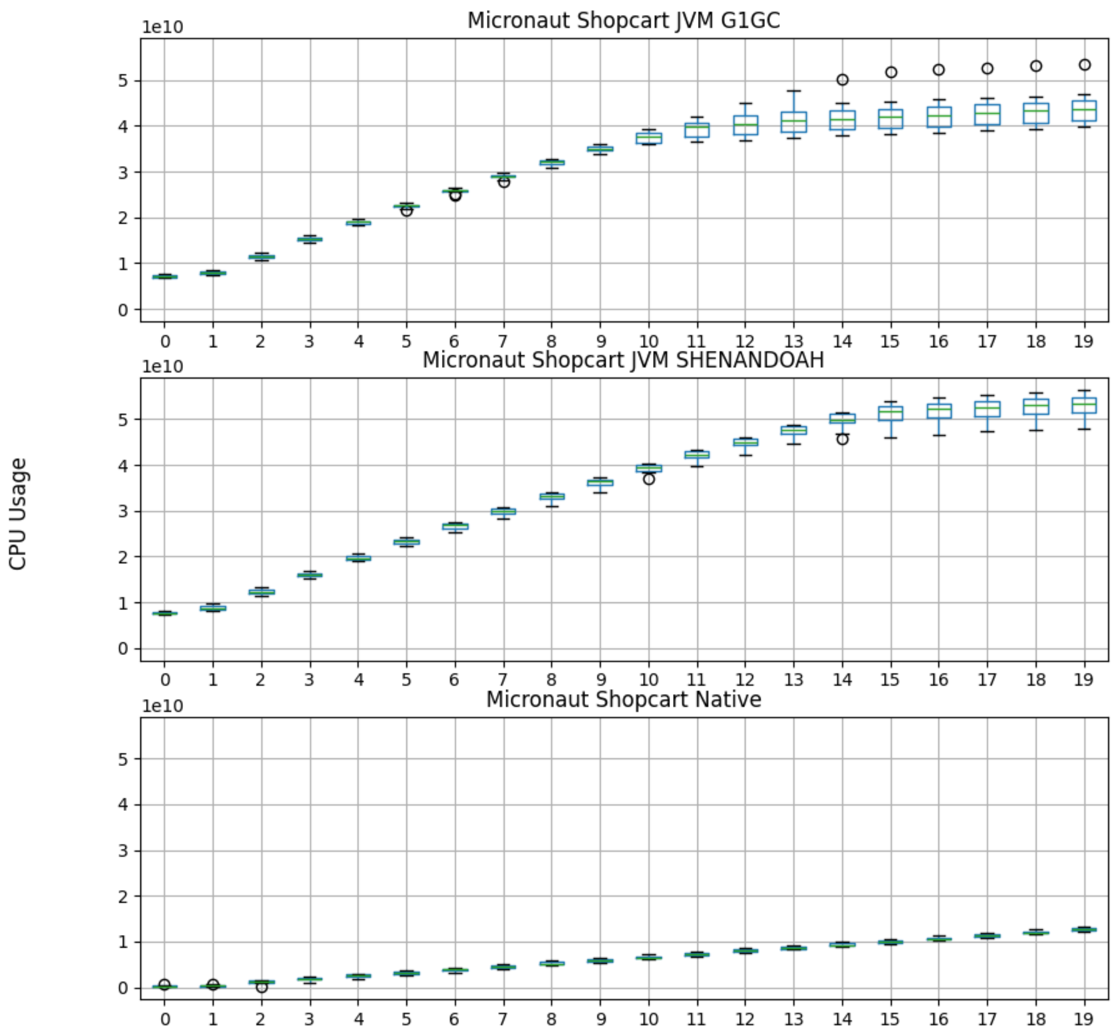
- Native images will consume less CPU than JVM applications under equivalent load, especially during startup and steady-state operation.
- The difference will be more visible in complex applications like Spring Boot, where JVM overhead is higher.

CPU metrics help assess not just performance, but cost-efficiency and scalability—key concerns for modern microservice and serverless deployments.

```
Text(0.5, 1.0, 'Micronaut Shopcart Native')
```


Spring Petlcinic JVM G1GC
1e11
Spring Petlcinic JVM SHENANDOAH
1e11
Spring Petlcinic Native
1e11
CPU Usage
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


Quarkus Tika JVM G1GC
1e10
Quarkus Tika JVM SHENANDOAH
1e10
Quarkus Tika Native
1e10
CPU Usage
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 2 4 6 8



The CPU usage measurements, expressed as cumulative CPU cycles, highlight clear differences in computational overhead between native and JVM execution.

- In Spring Petclinic and Quarkus Tika, both execution modes scale linearly with workload, but the JVM consistently consumes significantly more CPU across all intervals. This reflects JVM runtime overhead from garbage collection, JIT compilation, and thread scheduling, whereas native images avoid these costs.
- In Micronaut Shopcart, CPU demand is substantially lower overall. The gap between JVM and native is also much smaller, suggesting that for lightweight frameworks, native execution offers only modest CPU savings. Micronaut's efficient runtime minimizes differences between execution models.

Overall, native images demonstrate more efficient CPU utilization, especially in larger workloads, while lightweight applications like Micronaut narrow the gap.

7 Conclusion

The analysis shows that GraalVM native images consistently outperform JVM execution in startup time, memory usage, and CPU efficiency across all frameworks. These strengths make native execution especially well-suited for serverless and resource-constrained environments.

In terms of latency, native images often achieve lower or comparable averages and tail latencies, though results vary by workload. For throughput, however, the JVM sometimes performs better (e.g., Quarkus Tika), benefiting from JIT optimizations and adaptive GC strategies. Between the two JVM garbage collectors, G1GC and Shenandoah show broadly similar performance, but Shenandoah occasionally incurs higher memory usage under load.

Overall, native images provide faster startup and lower resource consumption, while the JVM—depending on GC choice and workload—can still excel in sustained high-throughput scenarios.