IDC | ANALYZE THE FUTURE

Red Hat Quarkus Lab Validation

RESEARCH BY:

Arnal Dayaratna
Research Director, Software Development, IDC

Table of Contents

Click on any section titles or page numbers to navigate to each.

Executive Summary 3

Key Findings 4

Methodology 6

Introduction 8

Java Remains the World's Most Popular Programming Language 8

Quarkus Optimizes Java for Containers Using a Closed-World Assumption Approach for Immutable Containers 10

SCENARIO 1.1 – Quarkus Start-Up Memory 11

SCENARIO 1.2A – Memory Utilization for 10 Pods 12

SCENARIO 1.2B – Deployment Density 13

SCENARIO 1.3 – Quarkus Memory Usage Under Load 15

SCENARIO 2.1 – Scale an Existing App 19

SCENARIO 3 – Time to First Response On-Premises or in a Serverless Environment 20

SCENARIO 3.1 – Measure Time to First Response on a Docker Container on Bare Metal 20

SCENARIO 3.2 – Measure Time to First Response on a Serverless Environment 21

SCENARIO 4.1.1 – Live Coding 23

SCENARIO 4.1.2 – Live Coding a Kafka Stream 25

SCENARIO 4.1.3 – Reactive and Imperative Programming Modules 27

Conclusion 29

Appendix 31

Executive Summary

This report focuses on comparing Quarkus with another widely used Java framework for cloud native development referred to, in this document, as Framework A. The comparison between Quarkus and Framework A is based on a number of criteria that are important for container, Kubernetes, and cloud deployments and developer efficiency. This report validates that **Quarkus can save as much as 64% of cloud resources as compared to Framework A when running in native mode and 37% when running on a Java Virtual Machine (JVM).** Furthermore, the report validates that Quarkus improves developer productivity as compared to Framework A. Improvements in developer productivity lead to faster time to market and innovation that empower organizations to remain competitive by providing enhanced user experiences and new innovative solutions to their customers.

Key Findings

Cost Savings

- The use of Quarkus JVM and Quarkus Native lead to cost-savings-driven higher deployment density of Kubernetes pods and reduced memory utilization.
- Expenditure of $100K USD annually in memory consumption for Framework A on AWS translates to $63K with Quarkus JVM and $36K with Quarkus Native.

Deployment Density

- The number of pods that could be started using Quarkus exceeded the number of pods that could be started using Framework A by:
  - ~8x for Quarkus Native
  - ~1.5x for Quarkus JVM
- Because customers can deploy more applications given the same amount of memory, Quarkus users can do more with the same amount of resources.

Start-Up Memory Usage

On-premises

- Quarkus Memory Native reduced memory usage by ~90% in comparison to Framework A.
- Quarkus Memory JVM reduced memory usage by ~20% in comparison to Framework A.

Cloud (OpenShift on AWS)

- For 10 Kubernetes pods on OpenShift 4.2 on AWS, the memory utilization of Quarkus was:
  - ~1/8 that of Framework A for Quarkus Native
  - ~2/3 that of Framework A for Quarkus JVM

Throughput Given Increasing Load

Quarkus JVM vs. Framework A

- For 32 concurrent connections, the ratio of Quarkus JVM throughput to Framework A throughput was ~1.5.
- Between 40 and 56 concurrent connections, the ratio of Quarkus JVM throughput to Framework A throughput ranged from 1.7 to 1.8.
- Quarkus JVM–based applications perform better than their Framework A counterparts as load increases.

- Under peak load, Quarkus JVM used 81% of the memory used by Framework A (214 MB compared to 264 MB).
- Quarkus JVM optimizes memory consumption in comparison to Framework A as an application scales.

Quarkus Native vs. Framework A

- Under peak load, Quarkus Native used ~1/3 of the memory used by Framework A (80 MB compared to 264 MB).
- Quarkus Native optimizes memory consumption in comparison to Framework A as an application scales.

Start-Up Time

On-premises

- Start-up time for Quarkus Native was roughly 12x faster than Framework A.
- Start-up time for Quarkus JVM was roughly 2x faster than Framework A.

Serverless

- The median Quarkus Native start-up time was ~4.5x faster than Framework A.
- The median Quarkus JVM start-up time was ~2x faster than Framework A.

Developer Productivity

- Quarkus improves developer efficiency by reducing the number of operational steps required to update applications.
- By unifying imperative and reactive programming, Quarkus gives developers the freedom to seamlessly choose between the two different styles of programming.
- Quarkus reduces maintenance time due to a smaller number of projects and source files to manage on the part of developers.

Methodology

These labs evaluate the benefits of using Quarkus in comparison to a non-Quarkus Java application, with a specific focus on cloud-native applications that leverage the conjunction of microservices, containers, container orchestration frameworks, and DevOps development practices.

In this document, labs are described as "Scenarios" that enumerate the findings derived from a specific deployment environment and methodology. All scenarios were executed by Red Hat in environments that Red Hat created and managed. The findings were observed by IDC via webcam contemporaneously with the execution of the labs.

Key components of our methodology were as follows:

- The comparison of Quarkus to a non-Quarkus Java application featured three application types, namely, Quarkus JVM, Quarkus Native, and Framework A.
- No optimizations to Java were made for the purposes of this lab. All heap memory settings are equivalent for Quarkus JVM, Quarkus Native, and Framework A.
- Framework A uses Tomcat by default in all of the scenarios discussed in this document. In the case of the developer productivity measurements in Scenario 4.1.1, 4.1.2 and 4.1.3, we used the Framework A Tools Suite.
- The environments that we examined were non-cloud-based and cloud-based and are detailed in each individual lab.
- These comparisons of a Quarkus to a non-Quarkus Java application focus on application-related costs. In quantifying costs, the labs assumed that the cost of RAM for an on-premises deployment was commensurate to that of a cloud-based deployment once maintenance costs are included. Another working assumption was that applications are not monolithic but instead feature a conjunction of discrete services that experience varying gradations of load and performance requirements.
- Our methodology measured cost savings by quantifying the number of nodes dedicated to workloads for Quarkus and non-Quarkus-based Java applications. In quantifying the number of nodes, infrastructures nodes (otherwise known as master nodes) were not considered. Because reductions in the number of application nodes that run containerized applications lead to reductions in the number of master nodes required, any cost savings attributable to decreases in applications nodes will be greater than what we have assumed here.
- Given that most production environments have associated environments for development, integration, and testing/QA, the number of instances of a service within an organization needs to be multiplied by the number of associated environments. IDC estimates that one service may have a minimum of seven to 10 instances at any one time within an organization.

METHODOLOGY CONTINUED

This means that the cost savings and developer productivity delivered by Quarkus for aproduction-grade application need to be multiplied by a factor that corresponds to the number of related environments.[1]

In calculating cost savings, IDC did not account for the way in which one production-grade service has a multitude of associated services that are likely to increase the cost by a multiple of seven to 10.

→ These labs examine the impact of Quarkus on developer productivity by examining how the number of operational steps required of developers to update applications changes for Quarkus versus non-Quarkus-based Java applications.

1. Typical instances of an application service in an enterprise use one instance for development, one instance for test, one for integration testing, one for pre-production, two or more for production (HA and load balancing) and two or more for disaster recovery. This amounts to a total of 8 instances.

Introduction

Java Remains the World's Most Popular Programming Language

As noted by IDC research, Java remains the most popular language used by developers worldwide: 9.3 million of the 13.5 million professional developers worldwide are "heavy to moderate" users of Java in comparison to 8.1 million for C/C++ and 7.9 million for JavaScript in 2020. The popularity of Java partly involves the ability to "write once, run anywhere" (WORA) as a result of the JVM infrastructure that renders Java applications portable across different operating systems, platforms, and devices.

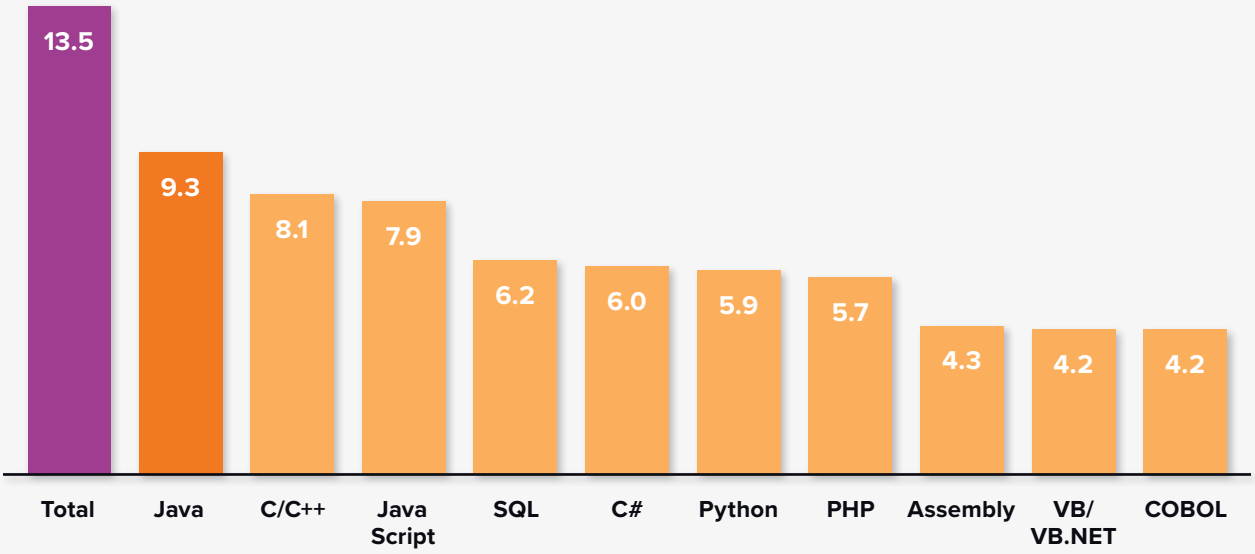
Figure 1 below illustrates how more than two thirds of the world's developers are "heavy to moderate" users of Java.

Even the most sophisticated enterprises are experiencing significant challenges in managing facilities and operations, resulting in serious impact on their organizations.

FIGURE 1

Number of full-time developers worldwide by language in 2020 (in millions)

B1. Please indicate how much you (or the team you are responsible for) used the following languages as a software developer in the last 12 months.



n = 2,500 | Source: PaaSView and the Developer 2019

INTRODUCTION CONTINUED

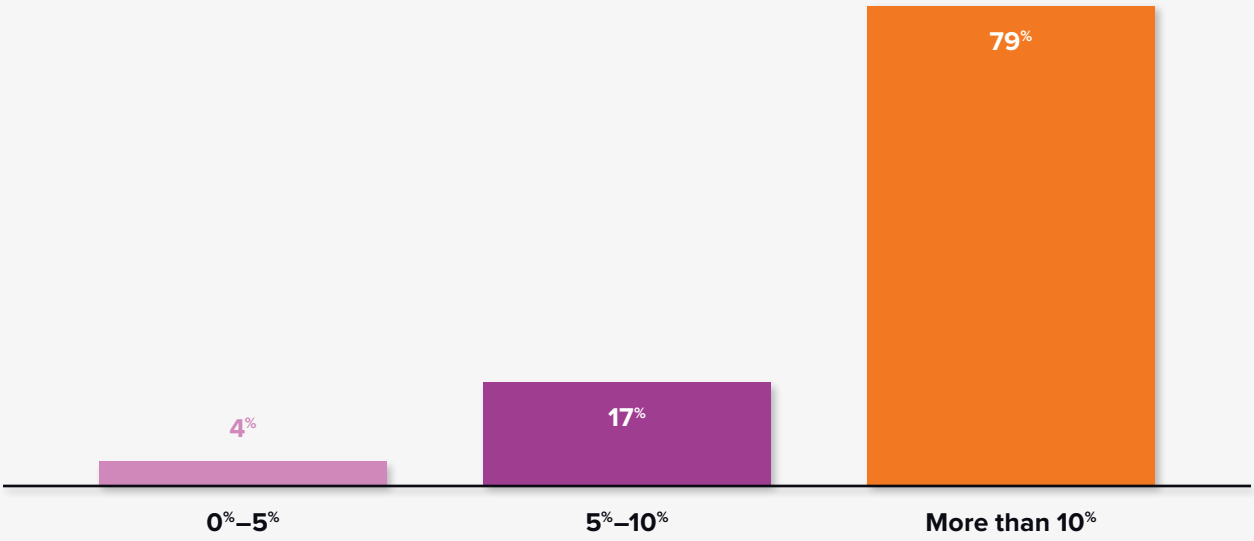
Increased adoption of containers has posed a challenge for Java, because containers deliver the portability that was previously provided by the JVM. JVMs running inside containers represent an extraneous infrastructure that brings operational inefficiencies to the performance of container-based Java applications. Java applications deployed on containers are notorious for large memory requirements and lengthy start-up times.

In recent years, container adoption has grown as shown in figure 2 below, which enumerates how close to 80% of developers use containers for at least 10% of their applications.

FIGURE 2

Percent of developers who use containers for net new apps

I5a. What percentage of net new applications that you developed within the last 12 months leverage the following?



n = 2,500 | Source: PaaSView and the Developer 2019

The popularity of Java and the emerging use of containers suggest the need to optimize the deployment of Java in container-native and Kubernetes-native environments. Quarkus optimizes Java for containers by increasing the deployment density of containers due to reduced memory utilization. Additionally, Quarkus decreases the start-up time for container-based applications, improves application throughput, and enhances developer productivity by means of its unification of imperative and reactive programming.

INTRODUCTION CONTINUED

Quarkus Optimizes Java for Containers Using a Closed-World Assumption Approach for Immutable Containers

Quarkus is a Kubernetes-native Java framework that leverages GraalVM Native Image to compile Java applications into a native executable. This means Quarkus combines the resource efficiencies associated with compiling an application into a native executable with the ability to create native binaries at build time. As such, Quarkus brings the benefits of native executables to Java applications in conjunction with support for deployment to containers, which facilitates application portability. Quarkus supports the immutable quality of containers and subsequently enables the WORA attribute of the JVM that was responsible for its popularity.

When containers started to become popular, the Java community solved some of the most common issues associated with running Java in containers, but the frameworks used to develop container-based applications remained largely unchanged. Many frameworks were designed for multi-application deployment environments like application servers, where applications needed dynamic class loading to address their differing needs on a shared JVM infrastructure. Many runtimes, like Framework A and Thorntail, continue to optimize for and use frameworks that perpetuate those earlier design decisions. However, when deploying an application to a Kubernetes-based environment in which containers are immutable, much of this runtime dynamic behavior has been resolved at build time instead.

One significant difference between Quarkus and Framework A is that Quarkus takes a closed-world assumption approach to Java in that it resolves as much as possible of the runtime dynamic features at build time and, in doing so, saves memory and start-up time. Not only do these benefits apply to the JVM, but they also make it easier to move workloads to a statically linked runtime by leveraging GraalVM Native Image.

SCENARIO 1.1

Quarkus Start-Up Memory

Background

This lab quantified the start-up memory of a Quarkus application in comparison to a Framework A application. The methodology used to measure start-up memory was to measure resident set size (RSS) in comparison to heap size. Resident set size refers to the amount of memory used by a process as opposed to the amount of memory that has been swapped out. To determine start-up memory, the resident set size was measured after the first request to the application.

Deployment Environment and Relevant Code

The deployment environment was on-premises and featured containers on bare metal. For more details regarding the deployment environment, see "Bare-Metal Environment A" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 1.1

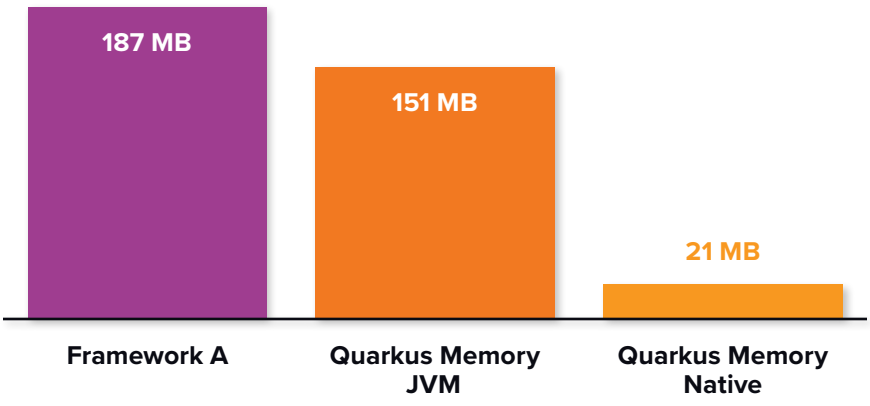
Key Findings

On-premises start-up memory usage of Quarkus on a JVM and natively was lower than Framework A as shown below:

Memory usage after the first request

Application stack	Memory utilization
Framework A	187 MB
Quarkus on JVM	151 MB
Quarkus on Native	21 MB

Memory (RSS) usage after the first request



- Quarkus Memory Native reduced memory usage by ~89% in comparison to Framework A.
- Quarkus Memory JVM reduced memory usage by ~20% in comparison to Framework A.

SCENARIO 1.2A

Memory Utilization for 10 Pods

Background

This lab quantified the ability of the Quarkus development framework to optimize memory utilization by measuring the following:

- Memory utilization after starting 10 pods

Deployment Environment and Relevant Code

OpenShift 4.2 hosted by AWS. For more detail about the environment, see "Container Orchestration Cluster" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 1.2

Key Findings

For 10 Kubernetes pods on OpenShift 4.2 on AWS, the memory utilization of Quarkus was:

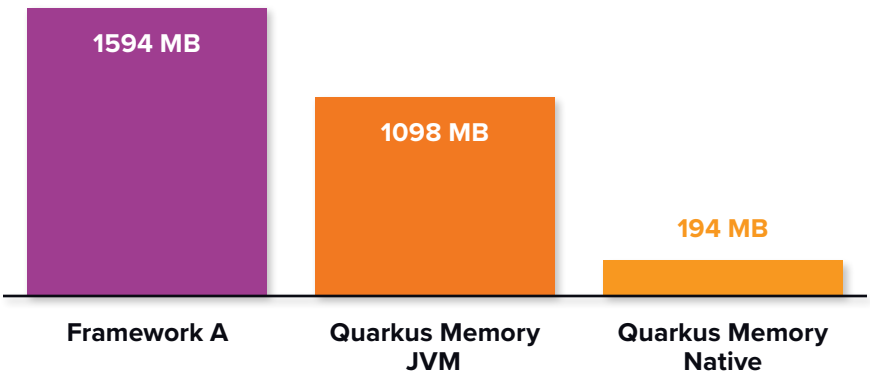
- ~1/8 that of Framework A for Quarkus Native
- ~2/3 that of Framework A for Quarkus on JVM

Public-cloud-based memory usage of Quarkus was lower than Framework A as shown below:

Memory utilization after starting 10 pods

Application stack	Memory utilization
Framework A	1594 MB
Quarkus on JVM	1098 MB
Quarkus on Native	194 MB

Memory usage for 10 instances



SCENARIO 1.2B

Deployment Density

Background

This lab quantified the following:

- The number of pods that can be started with 2 GB

Deployment Environment and Relevant Code

OpenShift 4.2 hosted by AWS. For more detail about the environment, see "Container Orchestration Cluster" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 1.2

Key Findings

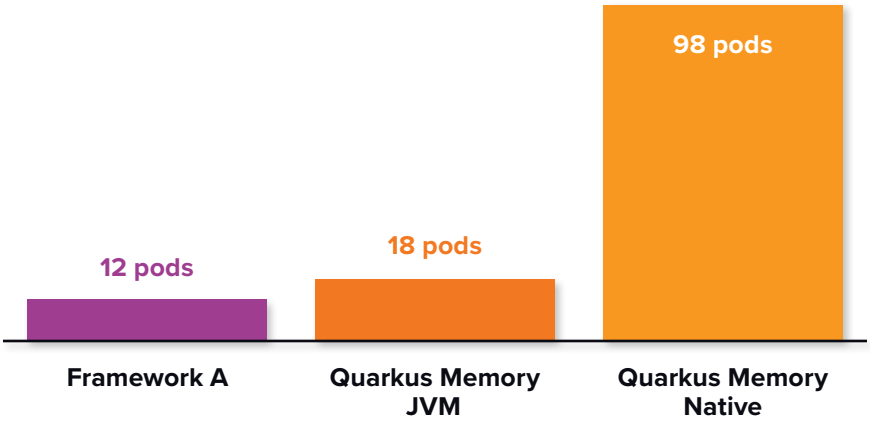
The number of pods that can be started with Kubernetes is greater in the case of Quarkus in comparison to Framework A, as follows:

- ~8x more pods in the case of Quarkus Native
- ~1.5x more pods in the case of Quarkus JVM

Number of pods that can be started with 2 GB

Application stack	Number of pods	Memory utilization
Framework A	12	1911 MB
Quarkus on JVM	18	1996 MB
Quarkus on Native	98	1967 MB

Memory usage for 10 instances



SCENARIO 1.2B – DEPLOYMENT DENSITY CONTINUED

Why It Matters

Context

- Kubernetes becomes more cost-effective as the number of applications that can be hosted on a cluster increases.
- Traditional cloud-native development stacks, based on the JVM, have large memory footprints and slow start-up times.
- Kubernetes needs direction on the amount of memory required for each pod that is scheduled to run on a node. If the node doesn't have the minimum memory needed for the application, it will die immediately.
- Reduced memory consumption of a Kubernetes application, as enabled by Quarkus, means an increased likelihood that the application will successfully run using Kubernetes.
- Quarkus increases the deployment density of container-based applications, given a specific memory allocation.

Significance

- Lower start-up memory is important because it indicates the amount of memory that an application that is not under heavy load will use in a production environment. In cases where the consumption of memory is used to calculate the cost, customers will pay less in cloud resource consumption for the same amount of memory using Quarkus than they would for Framework A.
- Quarkus Native allows customers to host roughly 8 times as many applications as Framework A, given the same amount of memory.
- Meanwhile, Quarkus JVM allows customers to host roughly 1.25 times as many applications as Framework A, given the same amount of memory.
- Quarkus optimizes resource consumption with respect to memory and subsequently enables cost savings related to memory consumption.
- Quarkus simplifies the operator experience by making it easier for customers to manage memory adequacy for container-native deployments.
- Quarkus users can do more with the same amount of resources. This means applications using Quarkus can be deployed on fewer nodes than applications that don't use Quarkus, and the reduction in the number of nodes means that organizations can leverage fewer infrastructure resources to accomplish the same goals, thereby saving money and rendering it easier to manage the resources in question.
- Because Quarkus JVM and Quarkus Native enable cost savings due to increased deployment density and optimized memory utilization, cloud-hosting costs can be reduced as a result as well.

SCENARIO 1.3

Quarkus Memory Usage Under Load

Background

This lab examined how a Quarkus application scales in relation to increasing demands from end users. The lab quantified the following:

- How application throughput varies in relation to the number of concurrent end users
- How memory consumption of a Quarkus application scales in relation to peak load, defined as the maximum number of requests per second
- Throughput per MB of memory

Deployment Environment and Relevant Code

The deployment environment was on-premises and featured containers on bare metal. See "Bare-Metal Environment B" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 1.3

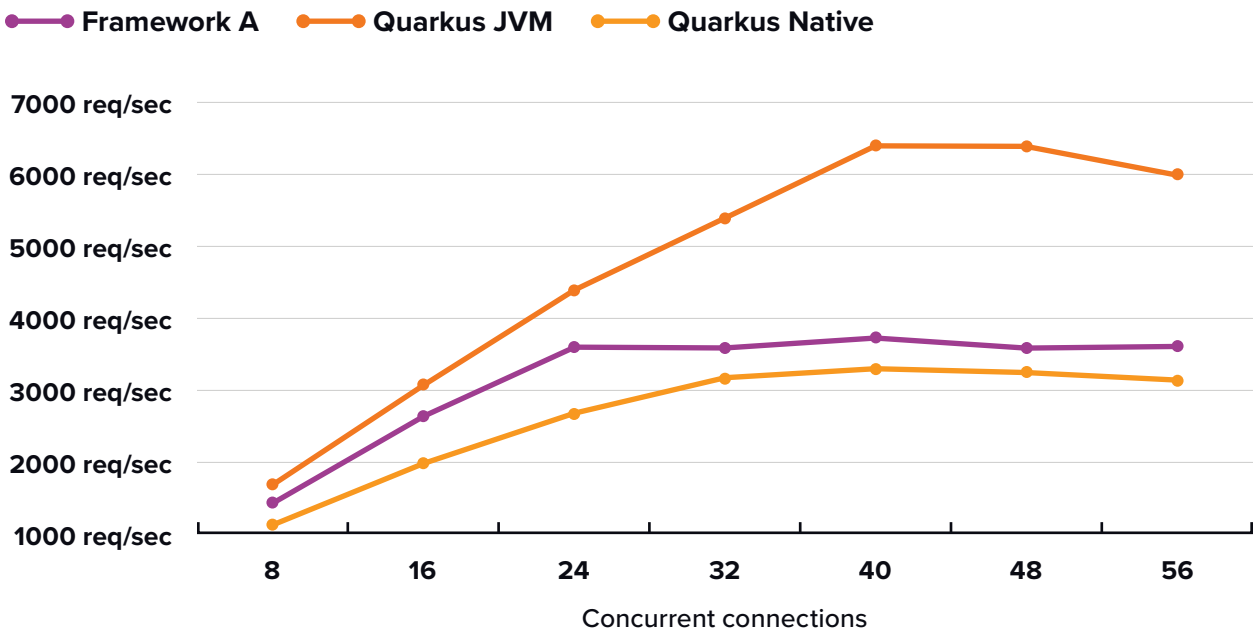
The lab measured memory usage of a Quarkus and a Framework A application as the load was steadily increased.

Key Findings

Throughput versus number of concurrent users (Peak value marked with bold)

Concurrent connections	Framework A	Quarkus JVM	Quarkus Native
8	1375 req/sec	1635 req/sec	1068 req/sec
16	2597 req/sec	3033 req/sec	1932 req/sec
24	3568 req/sec	4368 req/sec	2693 req/sec
32	3557 req/sec	5380 req/sec	3139 req/sec
40	**3697 req/sec**	**6396 req/sec**	**3266 req/sec**
48	3555 req/sec	6389 req/sec	3212 req/sec
56	3578 req/sec	5986 req/sec	3106 req/sec

SCENARIO 1.3 – QUARKUS MEMORY USAGE UNDER LOAD CONTINUED



Quarkus JVM vs. Framework A

- For 32 concurrent connections, the ratio of Quarkus JVM throughput to Framework A throughput was 1.5.
- Between 40 and 56 concurrent connections, the ratio of Quarkus JVM throughput to Framework A throughput ranged between 1.7 and 1.8.
- Quarkus JVM-based applications perform better than their Framework A counterparts as load increases.

Quarkus Native vs. Framework A

- For 32 concurrent connections, the ratio of Quarkus Native throughput to Framework A throughput was 0.9.
- Between 40 and 56 concurrent connections, the ratio of Quarkus Native throughput to Framework A throughput ranged between 0.85 and 0.9.
- Framework A is superior to Quarkus Native with respect to the ability to scale an application that is experiencing increased load.

Why It Matters

Context

- Throughput is a measure of the workload experienced by an application. Typically measured in requests/responses within a designated time frame, throughput is also often defined as the number of transactions per second. By measuring throughput in relation to the number

SCENARIO 1.3 – QUARKUS MEMORY USAGE UNDER LOAD CONTINUED

of concurrent connections, this lab examines how throughput scales in conjunction with demands imposed on the application by more concurrent users.

Significance

- For all three of the applications tested, throughput increased as the number of concurrent connections increased from 10 to 40.
- Quarkus JVM's ability to increase throughput in excess of that demonstrated by both Quarkus Native and Framework A illustrates its exceptional ability to continue to perform while experiencing intensified application demands.

Key Findings

Memory usage for peak load (MEM_PEAK)

Application stack	Memory utilization	Peak throughput
Framework A	264 MB	3697 req/sec
Quarkus JVM	214 MB	6396 req/sec
Quarkus Native	80 MB	3266 req/sec

Quarkus JVM vs. Framework A

- Under peak load, Quarkus JVM used 214 MB of memory in comparison to Framework A's use of 264 MB.
- Quarkus JVM optimizes memory consumption in comparison to Framework A as an application scales.

Quarkus Native vs. Framework A

- Under peak load, Quarkus Native used 80 MB of memory in comparison to Framework A's use of 264 MB.
- Quarkus Native optimizes memory consumption in comparison to Framework A as an application scales.

Why It Matters

Context

- Memory usage under peak load is a measure of the ability of an application to optimize memory consumption as the demands placed on it intensify. The smaller the memory usage under peak load, the more effectively an application can consume memory.

SCENARIO 1.3 – QUARKUS MEMORY USAGE UNDER LOAD CONTINUED

Significance

- Lower memory usage for Quarkus JVM and Quarkus Native indicate that they are more resource-efficient than Framework A.
- Moreover, lower memory consumption under peak usage means that Quarkus customers can increase the density of their applications and correspondingly maximize infrastructure-related resource consumption more generally.
- Customers using Quarkus JVM and Quarkus Native can expect to pay less for compute resources than their Framework A counterparts.

Key Findings

Result as req/sec/MB

Application stack	Req/Sec/MB	Comparison
Framework A	14 req/sec/MB	0%
Quarkus on JVM	30 req/sec/MB	113%
Quarkus on Native	41 req/sec/MB	193%

Result as req/sec/MB

- Throughput in req/sec/MB for Framework A, Quarkus JVM, and Quarkus Native was 14 req/sec/MB, 30 req/sec/MB, and 41 req/sec/MB, respectively.
- Throughput was fastest per MB for Quarkus Native.
- Both Quarkus JVM and Quarkus Native use memory more efficiently and deliver faster throughput for applications that are experiencing increased load than Framework A.

Why It Matters

Context

- Throughput per MB refers to the ability of an application to handle requests/second for each designated MB of memory. Greater throughput per MB is illustrative of enhanced capabilities to scale an application as load increases given memory constraints.

Significance

- Higher throughput for Quarkus JVM and Quarkus Native illustrates how Quarkus facilitates increased throughput while keeping memory utilization constant.
- Quarkus JVM and Quarkus Native enable applications to increase load without requiring a corresponding increase in memory to support the intensification of the load placed on an application.

SCENARIO 2.1

Scale an Existing App

Background

The objective of this lab was to understand the ease with which Quarkus-based applications can be scaled up. This lab focused on the ability of an application to scale up by quantifying the time it took for a pod to register itself as ready after initialization.

Deployment Environment and Relevant Code

The deployment environment was on-premises and featured containers on bare metal. For more detail about the environment, see "Bare-Metal Environment A" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 2.1

Key Findings

Result: Time to scale (seconds)

Framework	First Run	Second Run	Third Run	Median
Framework A	49.960	49.806	50.329	49.960
Quarkus JVM	30.359	29.829	30.451	30.359
Quarkus Native	10.293	9.446	9.935	9.935

- The median Quarkus JVM time to scale up was roughly 60% of the time required for Framework A.
- Meanwhile, Quarkus Native time to scale up was roughly 20% of the time required for Framework A. In other words, Quarkus Native scaled up roughly five times faster than Framework A.

Why It Matters

Context

- An application's ability to scale up is important because a single cluster can often be used for many applications. Given that many of an application's services can be variously scaled up or down, the ability of an application to scale up is illustrative of the ease with which a cluster can be used for many applications in parallel.

Significance

- Quarkus JVM and Quarkus Native demonstrated the ability to scale up faster than Framework A. The speed with which Quarkus enables containers to scale up means that services can be

SCENARIO 2.1 – SCALE AN EXISTING APP CONTINUED

scaled down when not utilized to optimize resource consumption. Developers can rely on Quarkus's ability to rapidly scale up when designing applications characterized by a subset of their services that are scaled down. The ability of Quarkus to rapidly scale up individual services enhances the latitude developers have to develop applications with microservice architectures that distribute services amongst a multitude of individual containers that differentially scale up and down.

- Quarkus's fast scale-up contributes to seamless failovers resulting in better end-user experience and higher customer satisfaction.
- Quarkus has the ability to complete a rolling upgrade in less time, getting new application versions up and running more quickly.

SCENARIO 3

Time to First Response On-Premises or in a Serverless Environment

Details of the deployment environment and key findings for both Lab 3.1 and 3.2 are given below.

SCENARIO 3.1

Measure Time to First Response on a Docker Container on Bare Metal

Background

This lab measured the start-up time of a Quarkus application in comparison to a Framework A application. The "time to first response" methodology was used.

SCENARIO 3.1 – MEASURE TIME TO FIRST RESPONSE ON A DOCKER CONTAINER ON BARE METAL CONTINUED

Deployment Environment and Relevant Code

The deployment environment was on-premises and featured containers on bare metal. For more detail about the environment, see "Bare-Metal Environment A" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 3.1

Key Findings

Result: Time to First Response (seconds)

Framework	First Run	Second Run	Third Run	Median
Framework A	9.675	9.649	9.598	9.649
Quarkus JVM	4.539	4.455	4.398	4.455
Quarkus Native	0.862	0.755	0.830	0.830

- Start-up time for Quarkus Native was 8.6% the start-up time of Framework A. In other words, the time to first response for Quarkus Native was roughly 12 times faster than Framework A.
- Meanwhile, the start-up time for Quarkus JVM was 46.2% that of Framework A. In fact, for Quarkus JVM, time to first response was two times faster than Framework A.

SCENARIO 3.2

Measure Time to First Response on a Serverless Environment

Background

This lab measured the start-up time of a Quarkus application in comparison to a Framework A application in a serverless environment. The methodology used for measuring start-up time was to use the time to first request.

SCENARIO 3.2 – MEASURE TIME TO FIRST RESPONSE ON A SERVERLESS ENVIRONMENT CONTINUED

Deployment Environment and Relevant Code

The environment type was a serverless environment on OpenShift 4.2. For more detail about the environment, see "Container Orchestration Cluster" in the appendix.

Code and configuration details: Appendix: Code and configuration for Lab 3.2

Key Findings

Result: Time to First Response (seconds)[2]

Runtime	Actual result	Compared with ref value
Go (for reference)	11.432s, 9.534s, 9.507s; MED: 9.534s	**0.000s (REF VALUE)**
Framework A	40.557s, 41.770s, 41.941s; MED: 41.770s	**32.236s**
Quarkus JVM	15.320s, 20.281s, 19.871s; MED: 19.871s	**10.337s**
Quarkus Native	9.237s, 9.076s, 9.585s; MED: 9.585s	**0.051s**

- The median Quarkus JVM time to scale up was roughly 47.6% of the time required for Framework A. In other words, the median Quarkus JVM time to first response was about two times faster than Framework A.
- Meanwhile, Quarkus Native time to scale up was roughly 22.9% of the time required for Framework A. In fact, the median Quarkus Native time to first response was 4.4 times faster than Framework A.

Why It Matters

Context

- One of the key benefits of containers as compared to virtual machines is their reduced start-up time. In the case of Java applications, start-up time for container-based applications tends to be longer than other applications because of the overhead introduced by JVMs running on container infrastructure. Because container-based Java applications are notorious for lengthy start-up times, improvements to these start-up times constitute a notable performance-related benefit.

2. Framework A, Quarkus JVM and Quarkus Native experienced a start-up time-related delay of 9.0 to 9.5 seconds due to an attribute of the specific version of OpenShift on which this lab was performed. Since the execution of this lab, this delay is no longer an attribute of the OpenShift platform. We used a "Go" application as a reference value to compare the difference between frameworks we use.

SCENARIO 3.2 – MEASURE TIME TO FIRST RESPONSE ON A SERVERLESS ENVIRONMENT CONTINUED

- For serverless infrastructure, reduced start-up time is important because containers are created on demand and the time to first response can impact user experience. Reductions in start-up time for container-native, serverless applications translate to lower application latency and improved performance. In the case of Functions-as-a-Service-based applications, for example, reduced start-up time means faster execution of an application subsequent to the realization of a predefined event or trigger.

Significance

- Quarkus reduced the start-up times for on-premises, container-based applications by 53.8% in the case of Quarkus JVM and 91.4% for Quarkus Native. These reductions indicate that Quarkus was responsible for a significant reduction in start-up time given that the median start-up for the application that was tested dropped from greater than 9.6 seconds in the case of Framework A to less than 4.5 seconds for Quarkus JVM and less than a second for Quarkus Native.
- Reductions in start-up time are particularly important for microservices applications in which all services are not running at the same time given that discrete services need to be initialized on demand. Similarly, reductions in start-up time are especially important for serverless applications that are triggered by predefined events.
- The ability of Quarkus to facilitate faster start-up times for serverless applications has profound implications for the use of Java for serverless applications. Quarkus empowers developers to develop serverless, Java-based applications that exhibit accelerated start-up times in comparison to Framework A. Faster start-up times are especially significant for Java-based serverless applications given Java's history of being responsible for long start-up times for container-native applications. By reducing the start-up time for container-based applications, Quarkus promises to render Java better-suited for serverless application development. Developers can now use Java in conjunction with Quarkus to reap the benefits of serverless applications without concerning themselves about delayed start-up times for the container-based applications that undergird serverless deployments.

SCENARIO 4.1.1

Live Coding

Background

This lab examined developer productivity enabled by Quarkus in comparison to Framework A by quantifying the number of operational steps required to update an application. Specifically, the lab examined the number of operational steps associated with adding extensions to Quarkus Maven projects.

SCENARIO 4.1.1 – LIVE CODING CONTINUED

Deployment Environment

- Quarkus 1.2.1.Final
- Java SE 11
- Maven 3.6
- Extensions
  - quarkus-resteasy
  - quarkus-resteasy-jsonb
  - quarkus-hibernate-orm
  - quarkus-hibernate-orm-panache
  - quarkus-jdbc-h2
- Quarkus 1.2.1.Final

For more details about updating Quarkus applications, see:
https://quarkus.io/guides/getting-started

Key Findings

	Quarkus	Traditional Java frameworks
Developer round trip (making and testing a change in source code) - 1 measure: # of steps	**2 steps:** • Change code • Save	**> 2 steps:** • Make a project change • Stop the service • Start the service • Test the change • Run the Framework A app • Test

- Quarkus requires two steps to update applications in the form of editing an application and saving it. In contrast, frameworks such as Framework A require developers to make a change to a project, stop and start relevant services, test the change, run the application, and then test the application again.
- Quarkus radically reduces the number of operational steps required by developers to update an application, thereby enhancing developer productivity.

Why It Matters

Context

Developers routinely make changes to code that they are developing. As developers make changes to code, they need to test them to determine whether they have had the desired effect. This often involves recompiling an application, deploying the application, reviewing the updated application, and repeating the same development life cycle. In the case of most Java applications, updates made by developers to applications are enabled by reloading or rebuilding packages, or otherwise by hotswapping jar/ear files.

SCENARIO 4.1.1 – LIVE CODING CONTINUED

Quarkus, on the other hand, uses live coding to perform updates to applications, which means developers can immediately see the effect of updates to applications. Another benefit of Quarkus for developers is the ability to use live coding on any IDE or across different build tools and obtain consistent results.

Significance

The practice of live coding, on the part of Quarkus, is significant because it facilitates improvements in developer productivity related to the operational efficiency of development-related tasks. Developers have to perform fewer steps to test the efficacy of updates to an application because they no longer need to reload packages or hotswap jar files.

In addition, Quarkus reverses stack traces to display the most meaningful error first to developers while doing live coding. This leads to lower troubleshooting times and higher productivity as well.

SCENARIO 4.1.2

Live Coding a Kafka Stream

Background

This lab examined developer productivity enabled by Quarkus for applications that use Kafka-based streaming data.

Deployment Environment

- Quarkus 1.2.1.Final
- Java SE 11
- Extensions
  - quarkus-resteasy
  - quarkus-resteasy-jsonb
  - quarkus-kafka-streams
  - quarkus-smallrye-reactive-messaging
- Quarkus Tools for Visual Studio Code

For more details about using Quarkus with Kafka extensions, see:
https://quarkus.io/guides/kafka-streams

SCENARIO 4.1.2 – LIVE CODING A KAFKA STREAM CONTINUED

For more details about connecting Kafka with Quarkus, see:
https://quarkus.io/guides/kafka

Key Findings

	Quarkus	Traditional Java frameworks
Developer round trip (making and testing a change in source code that uses Kafka Streams) - 1 measure: # of steps	**2 steps:** • Change code • Save	**> 2 steps:** • Stop the Kafka server • Build the application • Restart the application • Check if the changes are working

- By using Quarkus, developers can reduce the number of operational steps required to update a Kafka-based application.
- Whereas traditional Java frameworks such as Framework A require developers to stop and restart a Kafka server to understand the efficacy of application-related updates, Quarkus users can make application changes and save them without performing additional operational steps.

Why It Matters

Context

Developers routinely make changes to code that they are developing. As developers make changes to code, they need to test them to determine whether they have had the desired effect. This often involves recompiling an application, deploying the application, reviewing the updated application, and repeating the same development life cycle. In the case of most Java applications, updates made by developers to applications are enabled by reloading or rebuilding packages, or otherwise by hotswapping jar/ear files.

Quarkus, on the other hand, uses live coding to perform updates to applications, which means developers can immediately see the effect of updates to applications. Another benefit of Quarkus for developers is the ability to use live coding on any IDE or across different build tools and obtain consistent results.

Significance

The ability to make live changes to an application that leverages streaming data is significant because these applications typically require developers to stop and start streaming data to make updates to applications. By empowering developers to make live updates to streaming applications, Quarkus improves developer productivity for any application that leverages streaming data, whether an IoT or AI/ML application or a data analytics application that ingests an incoming data feed.

SCENARIO 4.1.3

Reactive and Imperative Programming Modules

Background

This lab examined the ability to program using both imperative and reactive approaches within the same source file in a project, and make live changes.

Deployment Environment

- Quarkus 1.2.1.Final
- Java SE 11
- Extensions
  - quarkus-resteasy
  - quarkus-resteasy-jsonb
- Quarkus Tools for Visual Studio Code

For more details about reactive programming using Quarkus, see:
https://quarkus.io/guides/getting-started-reactive

Key Findings

	Quarkus	Traditional Java frameworks
Developer round trip (making and testing a change in source code) - 2 measures: # of steps	**2 steps:** • Change code • Save	**> 2 steps:** • Create a new project • Change code • Save • mvn clean compile test • Run the app • Test **Maintain two different projects for two different styles of programming**

Why It Matters

Context

Imperative programming describes code that elaborates the set of steps a program must take to achieve a goal. Also known as procedural or algorithmic programming, imperative programming requires the entire set of steps that constitute an application to be executed for the application to perform.

SCENARIO 4.1.3 – REACTIVE AND IMPERATIVE PROGRAMMING MODULES CONTINUED

Reactive programming, in contrast, is a programming model that deals with asynchronous data streams and the propagation of change. A change to one variable in an application that is related to another automatically updates the other variable. The reactive model of programming means changes can be propagated asynchronously for different subsets of an application without requiring the compilation of all of its constituent steps.

While the choice of imperative and reactive programming is fundamentally stylistic, reactive programming is a good choice for applications that leverage a multitude of asynchronous data streams that deliver rapid updates to applications.

Significance

Quarkus empowers developers to create applications that feature reactive and imperative programming, thereby obviating the need to maintain discrete projects for the two different styles of programming. This centralization of the capability to develop imperatively and reactively renders it easier for developers to switch between the two programming modalities as needed within the same source file without resorting to separate development frameworks or the creation and maintenance of separate projects or sets of files.

In addition, different project teams can choose reactive or imperative styles of development while standardizing on the same runtime. This makes it easier for developers to move between projects. Teams can also leverage their investment in Java and not have to switch between different runtimes/languages like Node.js.

By bringing together the imperative and reactive models of programming, Quarkus supports non-blocking IO and avoids blocking the IO thread. By using a reactive core, Quarkus uses non-blocking code for application requests that pass through the event-loop thread. In addition, it supports imperative programming for developing REST or client-side applications. All application requests are managed by non-blocking IO threads that route the request to the code that is responsible for the request. The code that handles the request does so imperatively or reactively depending on the nature of the request. This ability to integrate imperative and reactive programming into application development makes it easier for developers to write code that executes asynchronously in conjunction with imperative code with which they are likely to be familiar.

Quarkus's enablement of imperative and reactive programming models improves developer productivity by minimizing context switching for developers and reducing the operational work required to program both reactively and imperatively. Given the relevance of reactive programming to reactive systems, the ability to code both reactively and imperatively contributes to a broader improvement of the developer experience.

Conclusion

Summary of Benefits

- Key benefits of using Quarkus include increased container deployment density, enhanced developer productivity, improved application throughput given increasing load, and decreased container start-up time.
- Taking into consideration the need for multiple environments for use cases such as development, testing, and high availability, the number of instances of services that an organization needs to run can be seven to 10 times the number of unique services. This means the total cost savings enabled by Quarkus is significantly higher than what we have represented here given that our analysis does not take into account the multitude of analogous environments that organizations need to run.
- Quarkus expands the range of use cases to which Java can be meaningfully applied, to serverless and Kubernetes-native development.
- Quarkus optimizes Java for serverless use cases because of rapid start-up times and its minimization of cold starts. Ennovative Solutions, for example, noted decreases in serverless start-up time from between six and seven seconds to less than 200 milliseconds as told to Red Hat by Ennovative Solutions in a blog post.
- Similarly, Quarkus optimizes Java for use with containers because of its increase of deployment density. Lufthansa Technik noted the ability to run deployments that were three times denser as measured by the number of microservices, leading to a threefold decrease in cloud resource costs, as told to Red Hat by Lufthansa Technik in a blog post.
- By unifying reactive and imperative programming, Quarkus gives individual developers as well as developer teams the freedom to choose imperative and reactive programming styles without switching between different runtimes, projects, and source files. This can result in the maintenance of a smaller number of projects and source files, leading to easier supportability of project assets and release processes.

Cost Savings

Cost Savings Resulting from Container Utilization

Consider an example in which a customer is using 300 discrete services in an application that correspond to 300 Kubernetes-based pods. Assume 120 pods require high memory utilization, another 120 pods require medium memory utilization, and 60 pods require low memory utilization. Based on this allocation of pods, we can compute how many nodes with AWS 16 GB of memory are needed for this application.

CONCLUSION CONTINUED

Based on the reduced memory utilization demonstrated by this lab, Quarkus JVM and Quarkus Native deliver cost savings as follows:

Application stack	Estimated Saving
Framework A	0%
Quarkus on JVM	37%
Quarkus on Native	64%

These savings in reduced cloud memory utilization are commensurate to the savings in real dollars that an organization would save by using Quarkus over Framework A. For illustration purposes, if you spend $100K USD annually in memory consumption on AWS using Framework A, you would spend $63K with Quarkus JVM and $36K with Quarkus Native.

Appendix

Bare-Metal Environment A

Operating System	Red Hat Enterprise Linux Server release 7.7 (Maipo)
CPU/Cores	2x CPU Intel(R) Xeon(R) CPU E5620 @2.40GHz, 8 cores (HT)
Total memory	157 GB
Container system	Podman
CGroups limit used	4 cores, 256 MB

Bare-Metal Environment B (Performance test)

Operating System	Red Hat Enterprise Linux release 8.0 (Ootpa)
CPU/Cores	2x Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz, 32 cores (HT)
Total memory	264 GB
Container system	Podman
CGroups limit used	4 cores, 256 MB

Container Orchestration Cluster

Kubernetes version	OpenShift 4.2
Number of worker nodes	3
Node AWS spec	m4.4xlarge • 2.3 GHz Intel Xeon® E5-2686 v4 (Broadwell) processors or 2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors • 16 vCPU (8 cores HT) • 64 GB Memory

APPENDIX CONTINUED

Code and configuration for Lab 1.1

The containers are started using the following command:

```
${container_runtime} run --ulimit memlock=-1:-1 -d --rm=true -p 5432:5432 \
       --network=${container_network_name} \
       --memory-swappiness=0 \
       --name ${container_db_name} \
       -e POSTGRES_USER=${psql_db_user} \
       -e POSTGRES_PASSWORD=${psql_db_password} \
       -e POSTGRES_DB=${psql_db_name} postgres:10.5 > /dev/null
```

See the code in GitHub here.

The **container_cpu_limit** will be set to 1 core and the **container_memory_limit** will be set to 500 MB.

The endpoint that we will hit in Quarkus code is the following:

```
@Path("/")
public class ExampleResource {

    @Inject
    @ConfigProperty(name = "app.runtime")
    String runtime;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return String.format("Hello, World from Quarkus running in %s !!!",runtime);
    }
}
```

See the code in GitHub here.

And for Framework A it's the following:

```
@RestController
@RequestMapping("/")
public class ExampleController {
  public final String runtime;

  public ExampleController(@Value("${app.runtime}") String runtime) {
    this.runtime = runtime;
  }

  @GetMapping
  public String sayHello() {
    return String.format("Hello, World from Spring on %s!!!",runtime);
  }
}
```

See the code in GitHub here.

APPENDIX CONTINUED

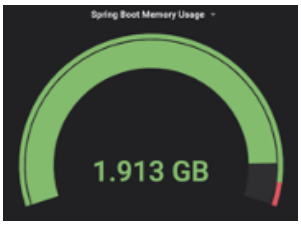
More importantly for the memory is actually what dependencies are used. The dependency will add a lot of classes to the meta space in a JVM, and depending on how good a framework is at minimizing or eliminating what is actually being, the lower the memory usage is going to be.

The following table details which dependencies are providing what functionality:

Functionality	Spring	Quarkus
REST and WEB	starter-web	quarkus-resteasy-jackson
Java Persistence API & Simplified access	starter-data-jpa, lombok	quarkus-hibernate-orm-panache
Database Drivers (PROD)	postgresql	quarkus-jdbc-postgresql
Database Driver (DEV)	h2	quarkus-jdbc-h2
Health Checks	starter-actuator	quarkus-smallrye-health

Code and configuration for Lab 1.1

Lab 1.2 uses the same code as Lab 1.1, but the setup is done in OpenShift and using Grafana to plot the memory usage and number of containers. Applications are separated with namespaces (called "project" in OpenShift), and the gauge for collecting Framework A memory looks like this:

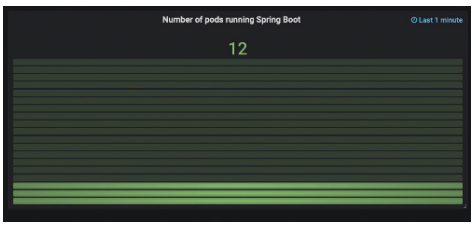


The query to collect the data looks like this:

```
sum by(namespace) (container_memory_rss{namespace=~"spring",container="",pod=~"todo-.*"})
```

Where the namespace are either "spring," "quarkus-jvm," or "quarkus"

To illustrate the number of containers, we use a "bar gauge," which looks like this:



The bar gauge is configured with a max and threshold value of 100 instances, and the query to collect the data looks like this:

```
sum by(namespace) (kube_pod_container_status_running{namespace=~"spring",pod=~"todo-.*"})
```

Where the namespace are either "spring," "quarkus-jvm," or "quarkus"

The containers are packaged together using Dockerfiles.

APPENDIX CONTINUED

Spring Dockerfile:

```
FROM fabric8/java-alpine-openjdk8-jre
ENV JAVA_OPTIONS="-Xmx64m -Djava.security.egd=file:/dev/./urandom"
ENV AB_OFF=true
COPY target/todo.jar /deployments/app.jar
EXPOSE 8080

# run with user 1001 and be prepared for be running in OpenShift too
RUN adduser -G root --no-create-home --disabled-password 1001 \
 && chown -R 1001 /deployments \
 && chmod -R "g+rwX" /deployments \
 && chown -R 1001:root /deployments
USER 1001

ENTRYPOINT [ "/deployments/run-java.sh" ]
```

Quarkus JVM Dockerfile:

```
FROM fabric8/java-alpine-openjdk8-jre
ENV JAVA_OPTIONS="-Xmx64m -Dquarkus.http.host=0.0.0.0 -Dquarkus.profile=java
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
ENV AB_OFF=true
COPY target/lib/* /deployments/lib/
COPY target/*-runner.jar /deployments/app.jar
EXPOSE 8080

# run with user 1001 and be prepared for be running in OpenShift too

RUN adduser -G root --no-create-home --disabled-password 1001 \
 && chown -R 1001 /deployments \
 && chmod -R "g+rwX" /deployments \
 && chown -R 1001:root /deployments
USER 1001
```

Quarkus Native Dockerfile:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY target/*-runner /work/application
#RUN microdnf update && microdnf install procps
RUN chmod 775 /work
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0","-Xms32m","-Xmx64m","-Xmn32m"]
```

The Dockerfiles has on purpose been constructed to use similar memory settings for Framework A, Quarkus JVM, and Quarkus Native.

Code and configuration for Lab 1.1

The code in Lab 1.3 is the same as for Lab 1.1 and Lab 1.2; however, to make sure we also make use of the functionality included in the application Lab 1.3, it will use a different endpoint than Lab 1.1 and Lab 1.2.

APPENDIX CONTINUED

The following endpoint will be used to load test the application for Spring:

```
@GetMapping
   public Iterable<Todo> findAll() {
       return todoRepository.findAll();
   }
```

See the code in GitHub here.

The following endpoint will be used to load test the application for Quarkus JVM and Quarkus Native:

```
@GET
   public List<Todo> getAll() {
       return Todo.listAll();
   }
```

See the code in GitHub here.

Both endpoints will query the PostgreSQL database for Todo's and transform the response to JSON.

Content returned from the Spring endpoint:


```
[
 {
   "id": 6,
   "title": "Introduction to Quarkus",
   "completed": true,
   "order": 0,
  "url": null,
   "user": {
     "id": 4,
     "surname": "Qvarnstrom",
     "firstname": "Thomas",
     "email": "no-reply@redhat.com"
   },
   "categories": [
     {
       "id": 1,
       "name": "Work"
     }
   ]
 },
 {
  "id": 7,
   "title": "Write Evaluation Plan",
   "completed": true,
   "order": 1,
   "url": null,
   "user": {
     "id": 4,
     "surname": "Qvarnstrom",
     "firstname": "Thomas",
```

APPENDIX CONTINUED


```
      "email": "no-reply@redhat.com"
    },
    "categories": [
      {
       "id": 1,
        "name": "Work"
      }
    ]
  },
  {
    "id": 8,
    "title": "Run Lab 1.1 - Startup memory",
    "completed": false,
    "order": 2,
    "url": null,
    "user": {
      "id": 4,
      "surname": "Qvarnstrom",
      "firstname": "Thomas",
      "email": "no-reply@redhat.com"
    },
    "categories": [
      {
        "id": 1,
        "name": "Work"
      }
    ]
  },
  {
    "id": 9,
    "title": "Run Lab 1.2 - Container density",
    "completed": false,
    "order": 3,
    "url": null,
    "user": {
      "id": 4,
      "surname": "Qvarnstrom",
      "firstname": "Thomas",
      "email": "no-reply@redhat.com"
    },
"categories": [
      {
        "id": 1,
        "name": "Work"
      }
    ]
  },
  {
    "id": 10,
    "title": "Run Lab 1.3 - Memory usage under load",
    "completed": false,
    "order": 3,
    "url": null,
```

APPENDIX CONTINUED

```
  "user" {
    "id": 5,
    "surname": "OHara",
    "firstname": "John",
    "email": "no-reply@redhat.com",
    "persistent": true
  },
  "categories": [
    {
      "id": 1,
      "name": "Work",
      "persistent": true
    }
  ],
  "persistent": true
 }
]
```

The performance test is run by increasing the number of connections (referred to as DRIVER_THREADS). Since the purpose of the test is to measure maximum throughput, the tests are executed without any wait time between calls. Number of connections is increased step wise to find a point where each runtime (spring-boot, quarkus-jvm, and quarkus-native) is found. Throughput (req/sec) and memory are recorded in each iteration.

Before the actual test is executed, there is a warm-up period to allow for the JVM's just-in-time optimization. Note that this is not needed for quarkus-native since quarkus-native has been optimized at build time instead of runtime and can therefore immediately deliver max throughput.

The actual test environment is using a Jenkins-based environment to schedule the job:

```
name: todo_{RUNTIME}_{DRIVER_THREADS}
http:
 host: http://{HOST}:{PORT}
 sharedConnections: "{SHARED_CONNECTIONS}"
phases:
 - rampUp:
     always:
       users: "{DRIVER_THREADS}"
       duration: "{RAMP_UP_DURATION}"
      scenario:
         - testSequence: &testSequence
             - httpRequest:
                 GET: "/api"
 - steadyState:
     always:
       users: "{DRIVER_THREADS}"
       startAfter:
         phase: rampUp
       duration: "{STEADY_STATE_DURATION}"
       scenario:
         - testSequence: *testSequence
```

View the file on GitHub.

APPENDIX CONTINUED

Results from the test are recorded as raw data and then provided to a graph service that will create a nice-looking graphical representation of the results.

Code and configuration for Lab 2.1

The applications in Lab 2.1 are the same applications that were used in Lab 1, and the code is available here. To deploy the application, a setup script is triggered using the following "make" command: `make oc-setup`. The setup script can be found here.

To run the lab as documented in the operational procedure, an HTTP command line client called HTTPie was used. To verify that the application was up and running, the Framework A application uses the Actuator and the Quarkus applications use SmallRye Health Checks (MicroProfile Health check). Both Actuator and SmallRye Health Checks are designed to respond with response code 200 OK when the application is ready to respond to requests.

To measure the time before an application is responding, the procedure details a command that will probe the application's health endpoint every 0.5 second, and when successful prints a time stamp. In between measurements, we are scaling up and scaling down the number of pods from 0 to 1 on the Kubernetes cluster.

Code and configuration for Lab 3.1

The applications in Lab 3.1 are the same applications that were used in Lab 1, and the code is available here. This lab requires a container infrastructure like Docker or Podman. The operational procedure uses a script called `src/main/scripts/run-lab2.sh` to start the dependent database container as well as the application containers (e.g., spring, quarkus-jvm, or quarkus-native). The script full source is available here. All three applications are started using the same container limits. etc.

The configuration for the script looks like this:

```
############
## Settings
############

container_runtime=podman
container_network_name=host
container_stats_extra_settings="--no-reset"

container_db_name=postgresql

container_spring_name=spring-boot
container_spring_port=8080
container_spring_image=spring/todo

container_quarkus_jvm_name=quarkus-jvm
container_quarkus_jvm_port=8081
container_quarkus_jvm_image=quarkus-jvm/todo
```

APPENDIX CONTINUED

```
container_quarkus_native_name=quarkus-native
container_quarkus_native_port=8082
container_quarkus_native_image=quarkus-native/todo

container_cpu_limit=4
container_memory_limit=512M

psql_db_host=localhost
psql_db_name=todo-db
psql_db_user=todo
psql_db_password=todo
```

To run the lab as documented in the operational procedure, an HTTP command line client called HTTPie was used. To verify that the application was up and running, the Framework A application uses the Actuator, and the Quarkus applications use SmallRye Health Checks (MicroProfile Health check). Both Actuator and SmallRye Health Checks are designed to respond with response code 200 OK when the application is ready to respond to requests.

To measure the time before an application is responding, the procedure details a command that will probe the application's health endpoint every 0.1 second, and when successful prints a time stamp. In between measurements, we are scaling up and scaling down the number of pods from 0 to 1 on the Kubernetes cluster.

Code and configuration for Lab 3.2

This lab requires a configured OpenShift environment with Serverless (Knative) installed.

At the time of this lab evaluation, OpenShift Knative was still not fully released and a Tech Preview version of OpenShift Knative was used. That version of OpenShift Serverless used a non-optimized scheduler,[3] which introduces an overhead of approximately 10 seconds to start the container. Since this report targets differences in performance, memory, and start-up time between different applications, a Go application was introduced as a reference application to use for comparison. After three measurements, a median value was determined for how long it takes to start a container running a Go application. This value was then deducted from the values for https://httpie.org/, Quarkus JVM, and Quarkus Native.

The applications used for Lab 3.2 are the same applications used in Lab 1.1. The script used in the operational procedure for Lab 3.2 to deploy all four applications to OpenShift can be found here.

3. OpenShift Knative has, between the lab execution and the publishing of this report, improved the scheduling time, and actual start-up times are now much lower.

About IDC

International Data Corporation (IDC) is the premier global provider of market intelligence, advisory services, and events for the information technology, telecommunications, and consumer technology markets. IDC helps IT professionals, business executives, and the investment community make fact-based decisions on technology purchases and business strategy. More than 1,100 IDC analysts provide global, regional, and local expertise on technology and industry opportunities and trends in over 110 countries worldwide. For 50 years, IDC has provided strategic insights to help our clients achieve their key business objectives. IDC is a subsidiary of IDG, the world's leading technology media, research, and events company.

IDC Custom Solutions

This publication was produced by IDC Custom Solutions. The opinion, analysis, and research results presented herein are drawn from more detailed research and analysis independently conducted and published by IDC, unless specific vendor sponsorship is noted. IDC Custom Solutions makes IDC content available in a wide range of formats for distribution by various companies. A license to distribute IDC content does not imply endorsement of or opinion about the licensee.

IDC Research, Inc.
5 Speen Street
Framingham, MA 01701
USA
508.872.8200

idc.com @idc

Doc. #US46109520