Technische Hochschule Deggendorf
Faculty of Applied Computer Science
The University of South Bohemia in České Budějovice
Faculty of Science
Degree Master Artificial Intelligence and Data Science

A universal approach to access different cloud blob storage from Kubernetes pods in a cost-efficient and scalable way to enable application portability between different cloud providers.

Master's thesis to obtain the academic degree:

Master of Science (M.Sc.)

at the Technical University of Deggendorf
and the University of South Bohemia

Presented by:
Cristian Portillo
Matriculation number:
12100552

Supervisor:
Prof. Dr. Andreas Wölfl

Second Supervisor:
M Sc. Davor Klincharski

On: 17.01.2024

TECHNISCHE HOCHSCHULE DEGGENDORF

Declaration

Name of the student: Cristian Portillo

Name of the supervisor: Prof. Dr. Andreas Wölfl

Topic of the thesis:

A universal approach to access different cloud blob storage from Kubernetes pods in a cost-efficient and scalable way to enable application portability between different cloud providers.

. . . .

. . . .

. . . .

1. I hereby declare that I have written the final thesis independently in accordance with § 35 Para. 7 RaPO (examination regulations for the universities of applied sciences in Bavaria, BayRS 2210-4-1-4-1-WFK) and have not yet submitted it elsewhere for examination purposes, no other than have used the specified sources or aids and have marked literal and analogous quotations as such. I declare that I am the author of this qualification thesis and that in writing it I have used the sources and literature displayed in the list of used sources only.

   Deggendorf, 31.01.2024 [illegible]
   Date Signature of student

2. Release of the thesis:

   [✓ Thesis in full is released immediately]

   ◯ Release of the thesis in full is postponed

   ◯ Full version to be archived and shortened version to be released

   Deggendorf, 31.01.2024 [illegible]
   Date Signature of student

Annotation

P. Cristian, “A universal approach to access different cloud blob storage from kubernetes pods in a cost-efficient and scalable way to enable application portability between different cloud providers,” M.S. thesis, in English, Faculty of Applied Computer Science, Deggendorf Institute of Technology, Deggendorf, Germany and Faculty of Science, University of South Bohemia, České Budějovice, Czech republic, 2023, p. 69

Annotation:
This thesis introduces a versatile approach enabling consistent access to cloud storages from Kubernetes pods. Using containers, this method establishes a standardized interface, relieving developers from the task of dealing with provider-specific code and enabling applications to transition smoothly across various cloud platforms. Furthermore, this thesis explores the selection of a suitable Java framework, such as Spring Boot, Quarkus, or Micronaut, to implement this approach, considering factors like performance among others. This research not only provides a solution to a problem but also contributes to establishing best practices for connecting a cloud storage to Kubernetes pods.

I declare that I am the author of this qualification thesis and that in writing it I have used the sources and literature displayed in the list of used sources only.

Deggendorf, 31.01.2024 .... [illegible]
Date Signature of student

Abstract

In today's digital landscape, businesses are increasingly turning to cloud storage for cost-effective and scalable data solutions. However, connecting these storages to Kubernetes pods presents significant hurdles due to the diverse APIs and provider-specific complexities involved. This thesis introduces a versatile approach designed to tackle these challenges, enabling consistent access to cloud storages from Kubernetes pods. Through the use of containers, this method establishes a standardized interface, relieving developers from the task of dealing with provider-specific code, and enabling applications to transition smoothly across various cloud platforms.

Furthermore, this thesis explores the selection of a suitable Java framework, such as Spring Boot, Quarkus, or Micronaut, to implement this approach, considering factors like performance among others.

This research not only provides a solution to a problem but also contributes to establishing best practices for connecting a cloud storage to Kubernetes pods. Additionally, the knowledge gained from this study empowers organizations to make informed decisions, optimize their operations, and achieve greater efficiency in managing their data across different cloud environments.

Contents

1 Introduction

In the era of cloud computing, organizations are increasingly relying on cloud blob storages for scalable and cost-effective data storage solutions. However, accessing different cloud blob storages from Kubernetes pods presents significant challenges due to a variety of APIs, and provider-specific complexities. This thesis presents a universal approach that aims to face these challenges and enable appropriate access to different cloud blob storages from Kubernetes pods. By utilizing a proxy orchestrator, this approach provides a unified interface, eliminating the need for developers to handle provider-specific code and enabling application portability between different cloud providers. Additionally, this thesis explores the selection of an appropriate Java framework, such as Spring Boot, Quarkus or Micronaut, to implement the proposed approach, considering factors such as performance and ease of integration with cloud blob storage APIs. This will enable us to resolve the question of which Java framework best supports the development of the universal approach, ensuring efficient integration with Kubernetes and providing the necessary tools and libraries for a smoothly interaction with various cloud storage providers.

We will additionally conduct a comprehensive comparative analysis of the file transfer outcomes achieved through the universally proposed approach in contrast to transfers executed via the cloud-native portals offered by each provider. This evaluation aims to find out the efficacy and assess the overall quality of the proposed solution. This will allow us to determine what are the performance and scalability implications of the proposed universal approach when accessing the different cloud storages and how does it compare to traditional approaches in terms of efficiency.

Through experimentation and evaluation, this thesis evaluates the effectiveness and scalability of the proposed universal approach and we will be able to determine how the implemented universal approach enhances application portability across various cloud providers, allowing organizations to easily migrate, switch, or adopt multicloud strategies without substantial codebase modifications or disruptions to the application workflow.

Through the investigation of this thesis, this work will not only provide a solution to a challenge but also contribute to the development of best practices for accessing cloud blob storages from Kubernetes pods. Furthermore, the knowledge gained from this research will enable organizations to make accurate decisions, improve their operations, and achieve greater efficiency in managing their data in multi-cloud or hybrid cloud environments.

1.1 Outline

This section outlines the organization of the chapters in this thesis. Chapter 2 discusses previous researches that relate to our work in this research. Chapter 3 provides essential background information for comprehending the thesis contents. This includes an exploration of

Kubernetes architecture and its advantages, an examination of evaluated Java frameworks (Micronaut, Spring Boot, Quarkus), an overview of cloud providers like AWS and MS Azure, and a discussion on Helm charts. Chapter 4 provides a detailed analysis to determine the best Java framework for this thesis. This includes examining various features, as well as the development tools and dependencies used. In Chapter 5, we dive into how the proposed solution was built and put into action. This section is supported by clear diagrams and explanations of the proposed architecture. Chapter 6 assesses the results of the implementation, looking at important factors like file transfer times and how easily the system can adapt to different environments. Lastly, Chapter 7 wraps up the work with conclusions and suggestions for potential future improvements. This reflects the ongoing nature of this field and the potential for ongoing progress and enhancements.

2 Related Work

In the initial phase of this thesis, the most important task was to decide the most fitting Java framework for its development. Łukasz Latusik et al. [1] conducted a comprehensive study investigating into three highly promising frameworks, Spring Boot, Micronaut, and Quarkus. The study primarily centered around the assessment of these frameworks performance in crucial areas such as computation, compilation, and deployment, all in the context of developing Microservices. It is worth noting, however, that this study did not explore further into other facets related to cloud solutions. Nevertheless, having the results from this research, a judicious decision could be made regarding the most suitable framework for implementation.

On a parallel note, Piotr Plecinski et al. [2] also embarked on a comparative exploration of the previously Java frameworks mentioned. However, their investigation ultimately focused towards projects involving sensor networks. This covered a diverse spectrum, containing applications ranging from telemedicine to the management of extensive sensor networks responsible for collecting scientific data. Their focus, distinct from our objectives, was on operating in environments characterized by constrained resources, exemplified by the use of BLE or WIFI transmitters.

Furthermore, in the research conducted by Shani du Plessis et al. [3], insightful findings emerged highlighting the strengths and weaknesses inherent in each of the considered Java Framework platforms for the development of this thesis. Their discerning analysis provided invaluable input for making informed decisions in this regard.

Lastly, Songbin Liu et al. [4] conducted an experimental exploration involving access to a cloud storage system from Tsinghua University. Their primary focus was on enhancing cache efficiency for retrieving stored files, constituting a significant departure from the core objectives of our thesis. It's important to note that the experiments in this study were conducted exclusively within a single cloud provider, and Songbin Liu did not extend the research beyond this scope.

3 Background Knowledge

In this section, we will present a brief research of the three main Java frameworks to determine the optimal choice for cloud-native development. Selecting the most suitable Java framework that aligns with the requirements of Kubernetes based cloud native development is important for implementing the proposed universal approach. The evaluation process will involve a comprehensive analysis of the Java frameworks like Spring Boot, Quarkus, and Micronaut. Factors taken into consideration will include performance, resource efficiency, community support among others.

3.1 An Overview of Kubernetes Architecture

The design of Kubernetes circles around the idea of a flexible service discovery mechanism. Similar to other distributed middleware platforms, a Kubernetes cluster consists of several compute nodes and one or more master nodes. Diagram3.1 shows a high-level representation of a Kubernetes cluster.

[5] mentions that the Kubernetes Master nodes serve as the central control hub of the cluster. They are responsible for managing the entire cluster, offering APIs for communication, and handling deployment scheduling. On the other hand, Kubernetes nodes (depicted on the right side of the diagram3.1) contain the necessary services to execute applications in units known as Pods.

Each master node comprises the following components:

- **API Server:** This component ensures synchronization and validation of information within Pods and services.
- **etcd:** It serves as a reliable and consistent storage solution for cluster data, acting as a shared memory for the "brain."
- **Controller Manager server:** This component monitors changes in the etcd service and utilizes its API to enforce the desired cluster state.
- **HAProxy:** In cases where high availability (HA) masters are configured, HAProxy can be added to evenly distribute loads among multiple master endpoints.

Francesco Marchioni[5] highlights that Kubernetes nodes, often referred as nodes, can be considered "workhorses" of a Kubernetes cluster. Each node exposes a set of resources (such as computing, networking, and storage) to your applications. The node also ships with additional components for service discovery, monitoring, logging, and optional add-ons. In terms

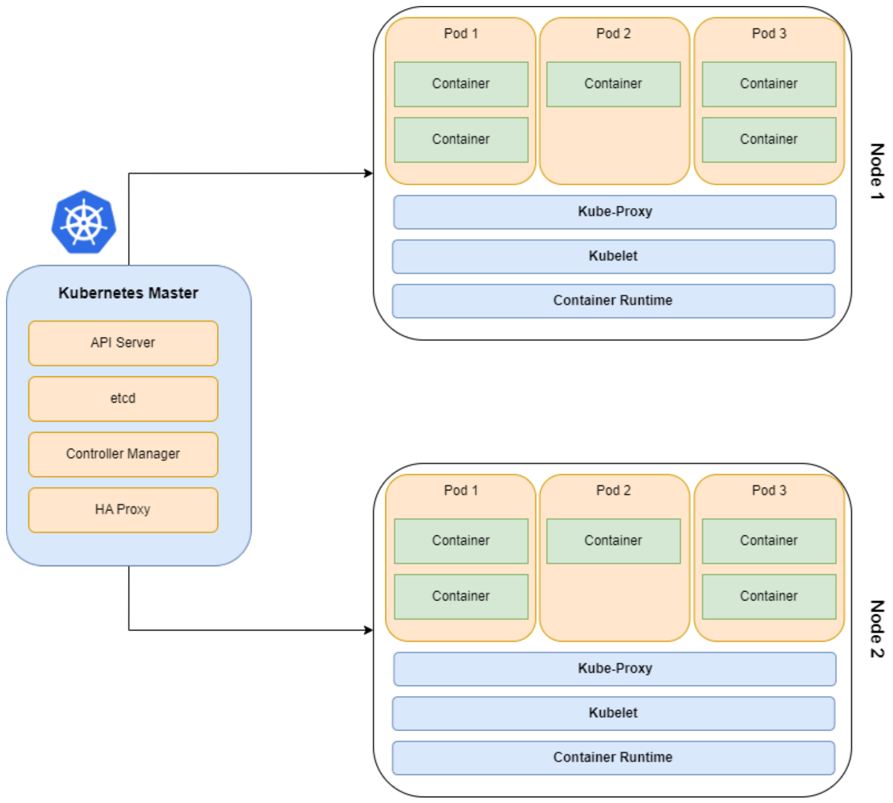


Figure 3.1: High level view of a Kubernetes cluster, diagram taken from [5]

of infrastructure, you can run a node as a virtual machine (VM) in your cloud environment or on top of bare-metal servers running in the data center.

[5] organizes each node to include the following components:

- **Pod:** It's a group that joins containers and parts of the application together. A Pod sets the limits for these containers, sharing resources and information. We can change the number of Pods while the application is running, ensuring we always have the right amount.
- **Kube-Proxy:** It's a traffic director on each node. It sets the rules for communication between Pods.
- **Kubelet:** It's a helper that runs on every node in the Kubernetes group. It makes sure the containers are running inside a Pod.
- **HAProxy:** In cases where high availability (HA) masters are configured, HAProxy can be added to evenly distribute loads among multiple master endpoints.
- **Container Runtime:** It's the software that runs the containers. Kubernetes works with different container runtimes, such as Docker, containerd, cri-o, and rktlet.

3.1.1 Benefits of using Kubernetes

[5] highlights that the advantages that Kubernetes offers inside an enterprise are:

- Kubernetes simplifies container management significantly. Instead of directly managing containers, you only need to handle Pods. Kubernetes introduces the concept of a service, which defines a logical group of Pods with their IP address. This abstraction improves fault tolerance and minimizes downtime by distributing containers across different machines.

- Kubernetes accelerates the software development process by supporting various programming languages and providing advanced deployment features. This facilitates the creation of efficient Continuous Integration/Continuous Delivery (CI/CD) pipelines.

- Kubernetes enables rapid and cost-effective horizontal scaling of Pods. As user numbers increase, the replication service can automatically launch new Pods and distribute the workload, ensuring uninterrupted service.

- Notably, Kubernetes can handle both stateless and stateful applications, offering ephemeral storage and persistent volumes. It supports various storage types, including NFS, GlusterFS, and cloud storage systems. Persistent volumes (PVs) can retain data independently of any specific Pod, allowing you to keep data as long as needed.

3.2 Quarkus

One of the primary challenges faced in a microservices architecture [1] is the potential complexity that arises from the expansion of services. Without a proper orchestration framework, managing and coordinating these services can become overwhelming. Additionally, the absence of centralized functions like authentication, data management, and API gateway can undermine the advantages offered by a microservices architecture.

Utilizing Kubernetes-based orchestration allows for efficient management and dynamic scheduling of microservices, improving resource utilization and enhancing resiliency. It enables smothly operations in response to varying demands without concerns about container failures. To fully integrate and unify all the components, a specialized framework adapted to this architecture becomes essential, and that is where Quarkus comes into the picture.

[5] considers Quarkus as a "Kubernetes native Java framework" and states that Quarkus emerges as a prominent solution for managing cloud-native enterprise applications, introducing interesting features that were previously unreachable. Quarkus can generate lightweight native code from Java classes, enabling the creation of container images that can be run on Kubernetes or OpenShift. It leverages renowned Java libraries such as RESTEasy, Hibernate, Apache Kafka, and Vert.x. Let's delve into the notable highlights of this framework.

[1] Microservices architecture is an architectural style for structuring an application as a collection of different independent services. Each service is focused on a single responsibility and only performs tasks related to this responsibility.

3.2.1 Native code execution

[5] states that native code execution has been attempted before in the past of Java, but it failed to gain significant developer adoption. For monolithic applications, the advantages of native execution were relatively minor due to advancements in Hot Spot technology, bringing Java's speed closer to native execution.

However, in a microservices scenario, the ability to quickly spin up native services becomes crucial. Even optimizing seconds or fractions of a second can make a significant difference. Similarly, if you aim to achieve high memory density, maximum request throughput, and consistent CPU performance, Quarkus native execution aligns perfectly with these requirements.

In contrast, Quarkus offers a seamless transition by utilizing plain Java bytecode[2]. This enables the development of applications with specific requirements, such as high memory density, superior CPU performance, advanced garbage collection tactics, compatibility with a wide range of libraries and monitoring tools that rely on the standard JDK, and the ability to compile once and run anywhere. Table 3.1 provides an overview of common scenarios where the choice between native applications and Java applications becomes relevant when working with Quarkus.

Quarkus Native applications	Quarkus Java applications
Highest memory density requirements	High memory density requirements
More consistent CPU performance	Best raw performance(CPU)
Fastest startup time	Fast Startup time
Simpler garbage collection	Advanced garbage collection
Highest throughput	A large set of libraries and tools that only work with JDK
No JIT spikes	Compile once, run anywhere

Table 3.1: Native applications vs java applications when developing with Quarkus

3.2.2 Quarkus Architecture

[5] states that the core element of Quarkus is responsible for the crucial task of transforming the application during the build phase, resulting in highly optimized native executable and Java-runnable applications. To achieve this, Quarkus core collaborates with several tools:

- **Jandex:** An efficient Java annotation indexer and offline reflection library that creates a compact representation of all runtime visible Java annotations and class hierarchies for a given set of classes.
- **Gizmo:** A bytecode generation library employed by Quarkus to generate Java bytecode.
- **GraalVM:** A collection of components, each with a specific role. These include a compiler, an SDK API for integrating Graal languages and configuring native images, and a runtime environment for JVM-based languages.

[2]Bytecode is the number of bytes needed to encode a program and has the ability to create a single image of a program that will execute identically (in principle) on any system equipped with a Java virtual machine.[6]

- **SubstrateVM:** A subcomponent of GraalVM that enables ahead-of-time (AOT) compilation of Java applications, transforming them into self-contained executables.

Diagram 3.2 provides an overview of the essential components in the Quarkus architecture. However, it is important to note that the list of available extensions is not exhaustive due to the need for brevity.

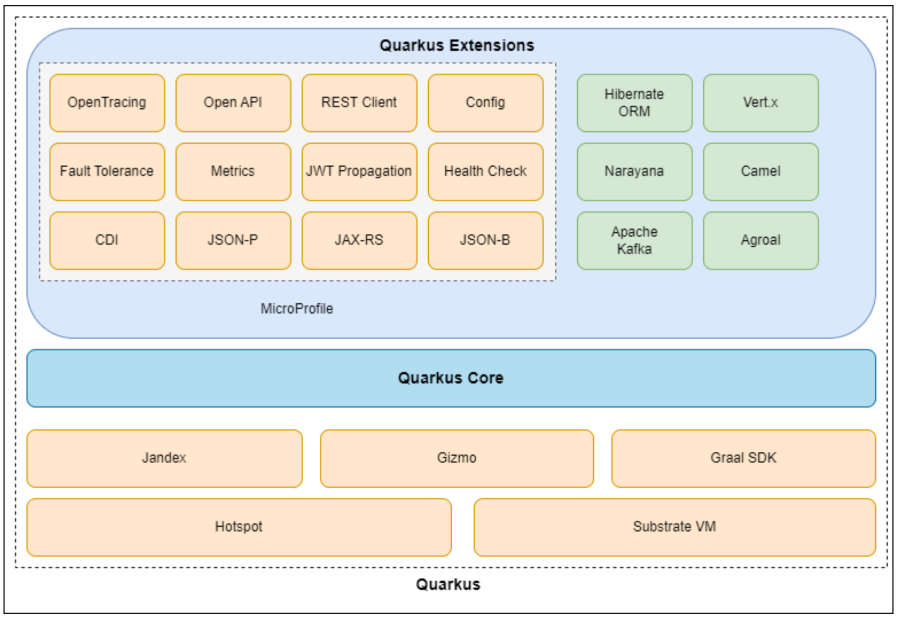


Figure 3.2: Core components of the Quarkus architecture, diagram taken from [5]

3.2.3 GraalVM

[5] describes that in order to generate native executables from Java code, an extension of the virtual machine[3] known as GraalVM is required. GraalVM serves as a versatile virtual machine that enables the compilation of bytecode from various languages, including Python, JavaScript, Ruby, and more. It also allows for the integration of multiple languages within the same project. Additionally, GraalVM offers features such as Substrate VM, a framework that facilitates ahead-of-time (AOT) compilation[4] for applications written in different languages. This enables the conversion of JVM bytecode into native executables.

GraalVM, like other JDKs available from different vendors, has support for the Java-based JVM Compiler Interface (JVMCI) and utilizes Graal as its default just-in-time (JIT) compiler. Consequently, it not only executes Java code but also supports languages like JavaScript, Python,

[3] A virtual machine, commonly shortened to just VM, are often thought of as virtual computers or software-defined computers within physical servers, existing only as code.[7]

[4] AOT, is the action to improve the performance of a Java virtual machine (JVM) by translating bytecode into C code, which is then compiled into machine code via an existing C compiler [8]

and Ruby. This capability is made possible through Truffle, a language abstract syntax tree interpreter developed by Oracle in collaboration with GraalVM.

Diagram 3.3 offers a high-level overview of the GraalVM stack.

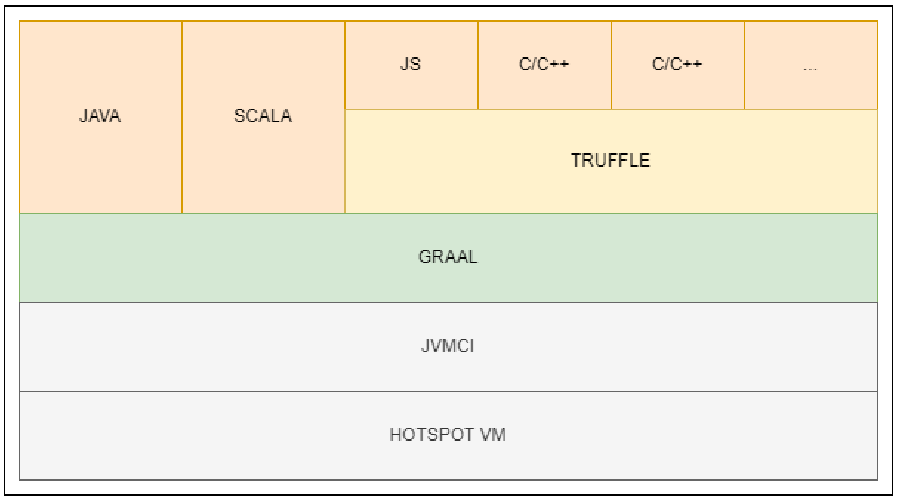


Figure 3.3: High-level view of the GraalVM stack, diagram taken from [5]

3.3 Spring Boot

Christian Posta[9] points out that when discussing Spring Boot, it's necessary to mention Spring itself. Spring is a popular and free framework that runs on the Java Virtual Machine (JVM) and is used to create high-quality applications. It was developed to address the challenges found in Java Enterprise Edition (JEE)[5]. In the beginning, the earlier versions of Spring were not very user-friendly, and developing applications with it was difficult and unpleasant. JEE solutions were complex and difficult to configure.

The goal of the developers was to make Spring accessible to everyone, so they provided default configurations from the start, which made the development process much easier. One of the key advantages of Spring is its own IoC [6] container, The IoC container takes care of the entire lifecycle of objects, starting from their creation until they are no longer needed.

Talking specifically about Spring Boot. It is a tool that makes it faster and easier to develop web applications and microservices using the Spring Framework[11].

3.3.1 Introducing Spring Boot

According to the official documentation[12] Spring Boot empowers developers to build self-contained, high quality applications based on the Spring framework, which can be executed

[5]Fundamentally, Java Enterprise Edition (Java EE), formerly referred to as J2EE, is a compilation of standardized specifications that provide prescribed solutions to commonly encountered software development obstacles.[10]

[6]IoC stands for Inversion of Control, which means that the responsibility for creating objects is transferred to the Spring container, which handles the creation, management, and configuration of bean objects.

easily. By adopting a predefined approach to the Spring platform and external libraries, Spring Boot minimizes the complexities involved in project initiation. In many instances, Spring Boot applications demand minimal Spring configuration.

Thanks to Spring Boot, developers can create Java applications that can be started using "java -jar". This flexibility enables easily execution and deployment options for Java applications developed with Spring Boot.

[12] states that the main goals of spring boot are:

- Offer an exceptionally fast and widely accessible starting point for all Spring development endeavors.
- Initially provide opinionated defaults but quickly accommodate specific requirements as they deviate from the defaults.
- Deliver a range of non-functional capabilities that are applicable to a wide range of projects, including embedded servers, security features, metrics, health checks, and externalized configuration.
- Eliminate the need for code generation (except when targeting native image) and eradicate the reliance on XML [7] configuration.

3.3.2 System requirements

To utilize Spring Boot 3.1.0, it is necessary to have Java 17 installed, and it remains compatible with Java versions up to and including Java 20. Additionally, Spring Framework 6.0.9 or a more recent version is required for compatibility[12].

Explicit build support is provided for the following build tools as shown in table 3.2

Build Tool	**Version**
Maven	3.6.3 or later
Gradle	7.x (7.5 or later) and 8.x

Table 3.2: Spring Boot System requirements

Servlet Containers

Spring Boot provides support for several embedded servlet containers, as shown in table 3.3:

In addition to the support for embedded servlet containers, Spring Boot also allows you to deploy your applications to any servlet 5.0+ compatible container. This flexibility enables you to choose from a wide range of containers based on your specific needs and preferences[12].

[7] XML stands for Extensible Markup Language and is a markup language for documents containing structured information[13]

Name	Servlet version
Tomcat 10.0	5.0
Jetty 11.0	5.1
Undertow 2.2 (Jakarta EE 9 variant)	5.0

Table 3.3: Spring Boot Servlet Containers

GraalVM Native Images

You can transform Spring Boot applications into Native Images [8] by utilizing GraalVM 22.3 or a later version.

[14] mentions that to generate these images, you have multiple options. You can utilize the native build tools such as Gradle/Maven plugins or use the native-image tool offered by GraalVM. Additionally, the native-image Paketo buildpack[9] can be utilized to create native images as well.

3.3.3 Developing with Spring Boot

This subsection provides a comprehensive exploration of the recommended practices for utilizing Spring Boot. It examines into various aspects, including build systems, auto-configuration, running applications, and essential best practices. While Spring Boot is treated as any other consumable library, adhering to these suggestions can greatly facilitate the development process, enhancing overall efficiency.

Build Systems

It is highly recommended to opt for a build system that facilitates dependency management 3.3.3 and has the capability to incorporate artifacts from the "Maven Central" repository. Spring Boot documentation [12] recommend considering Maven or Gradle as preferred choices. While it is feasible to configure Spring Boot with alternative build systems like Ant, it is important to note that they may not receive extensive support and may require additional configuration.

Dependency Management

[12] mentions that each version of Spring Boot offers a carefully selected set of dependencies that it is compatible with. In practice, you are not required to specify the version for these dependencies in your build configuration since Spring Boot takes care of managing them for you. Consequently, when you upgrade your Spring Boot version, these dependencies will also be upgraded consistently and in a synchronized manner. This streamlined approach ensures that

[8]GraalVM Native Images are standalone executables that can be generated by processing compiled Java applications ahead-of-time. Native Images generally have a smaller memory footprint and start faster than their JVM counterparts.[12]

[9]The Paketo Buildpack for Native Image is a Cloud Native Buildpack that utilizes the GraalVM Native Image builder (native-image) to compile an independent executable from an executable JAR file.

the versions of the dependencies remain compatible with each other, simplifying the process of managing and upgrading your Spring Boot applications.

Maven

[15] points out that the Spring Boot Maven Plugin facilitates Spring Boot integration within Apache Maven[10]. It offers various capabilities such as packaging executable jar or war archives, running Spring Boot applications, generating build information, and starting your Spring Boot application before executing integration tests. With this plugin, you can streamline your development workflow and leverage the features and functionality provided by Spring Boot seamlessly within your Maven-based projects.

Gradle

The Spring Boot Gradle Plugin[16] offers easily integration of Spring Boot within the Gradle build system. It facilitates tasks such as packaging executable Jar or WAR archives[11], running Spring Boot applications, and leveraging the dependency management capabilities provided by spring-boot-dependencies. The Gradle plugin for Spring Boot specifically requires Gradle version 7.x (7.5 or later) or 8.x, and it is compatible with Gradle's configuration cache feature[12]. By using this plugin, developers can efficiently manage their Spring Boot projects within the Gradle ecosystem, optimizing the build and deployment process.

Ant

Building a Spring Boot project using Apache Ant+Ivy[18] is an option. Additionally, the "AntLib" module named spring-boot-antlib is provided to assist Ant in generating executable jars.
To specify dependencies, an ivy.xml file commonly resembles the following example:

```xml
<ivy-module version="2.0">
    <info organisation="org.springframework.boot" module="spring-boot-sample-ant" />
    <configurations>
        <conf name="compile" description="everything needed to compile this module" />
        <conf name="runtime" extends="compile" description="everything needed to run this module" />
    </configurations>
    <dependencies>
        <dependency org="org.springframework.boot" name="spring-boot-starter"
```

[10] Apache Maven is a tool for project management and comprehension in software development. It operates on the principle of a project object model (POM), allowing for centralized control over a project's build process, reporting, and documentation.

[11] WAR file (Web Application Resource or Web application Archive) is a file used to distribute a collection of JAR-files, JavaServer Pages, Java Servlets, Java classes, XML files, tag libraries, static web pages (HTML and related files) and other resources that together constitute a web application.[17]

[12] The configuration cache is a feature that significantly improves build performance by caching the result of the configuration phase and reusing this for subsequent builds. Using the configuration cache, Gradle can skip the configuration phase entirely when nothing that affects the build configuration, such as build scripts, has changed.

```
            rev="${spring-boot.version}" conf="compile" />
    </dependencies>
</ivy-module>
```

An example of a standard build.xml appears as follows:

```
<project
    xmlns:ivy="antlib:org.apache.ivy.ant"
    xmlns:spring-boot="antlib:org.springframework.boot.ant"
    name="myapp" default="build">

    <property name="spring-boot.version" value="3.1.0" />

    <target name="resolve" description="--> retrieve dependencies with ivy">
        <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]"
 />
    </target>

    <target name="classpaths" depends="resolve">
        <path id="compile.classpath">
            <fileset dir="lib/compile" includes="*.jar" />
        </path>
    </target>

    <target name="init" depends="classpaths">
        <mkdir dir="build/classes" />
    </target>

    <target name="compile" depends="init" description="compile">
        <javac srcdir="src/main/java" destdir="build/classes" classpathref="
    compile.classpath" />
    </target>

    <target name="build" depends="compile">
        <spring-boot:exejar destfile="build/myapp.jar" classes="build/classes
    ">
            <spring-boot:lib>
                <fileset dir="lib/runtime" />
            </spring-boot:lib>
        </spring-boot:exejar>
    </target>
</project>
```

Starters

[12] states that starters provide a collection of convenient dependency descriptors that simplify the inclusion of necessary dependencies in the application. They serve as a comprehensive package for all the required Spring and related technologies, eliminating the need to search for sample code and manually copy-paste dependency descriptors. For instance, if you intend to utilize Spring and JPA for database access, you can easily incorporate the spring-boot-starter-data-jpa dependency in your project.

These starters include a wide range of dependencies that expedite the setup of a project, ensuring a consistent and well-supported set of managed transitive dependencies.

3.4 Micronaut

Micronaut is a modern Java framework[19] that runs on the Java Virtual Machine (JVM) and offers a comprehensive set of tools for creating modular and easily testable applications. It supports Java, Kotlin, and Groovy programming languages.

The primary goal of Micronaut is to provide a complete suite of features for JVM applications, including Dependency Injection and Inversion of Control (IoC), Aspect Oriented Programming (AOP), and sensible defaults with auto-configuration.

Many APIs in Micronaut are heavily influenced by Spring and Grails, intentionally designed to facilitate a smooth transition for developers familiar with these frameworks.

3.4.1 Micronaut and Microservices Development

According to Nirmal Singh and Zack Dawood[20], Micronaut is designed from scratch with a strong focus on addressing the specific challenges faced in microservices development as follows:

- **Dependency injection:** Micronaut achieves dependency injection by utilizing JSR-330's @Inject [13]annotation.Integrating the Java inject module into the compiler, Micronaut processes all relevant annotations during compile time. This results in the generation of bytecode for the classes based on the annotations present in their source code. Importantly, this entire process occurs during compilation, not runtime. During runtime, Micronaut can effectively instantiate the beans and retrieve their metadata directly from the generated bytecode, eliminating the need for slower reflection-based approaches.
- **Ahead-of-time compilation:** Unlike other frameworks that rely on reflection and generate annotation metadata at application startup, Micronaut performs these tasks during compile time. It utilizes annotation processors to process the metadata into bytecode using ASM (assembly)[14], which is further optimized by Java's just-in-time (JIT) compiler. This approach eliminates the need for runtime reflection and reduces the memory usage.
- **Faster boot-up time and lower memory consumption:** Unlike other frameworks that rely on reflection and perform classpath scanning at startup to generate reflection metadata, Micronaut's ahead-of-time compilation approach eliminates this overhead. By offloading the work to the compilation phase, Micronaut achieves faster boot-up times and reduces runtime memory requirements. The use of reflection metadata is minimized, resulting in more efficient resource utilization.

[13]This package specifies a means for obtaining objects in such a way as to maximize reusability, testability and maintainability compared to traditional approaches such as constructors, factories, and service locators. This process, known as dependency injection, is beneficial to most nontrivial applications.

[14]Assembly is a low-level programming language that's one step above a computer's native machine language, is commonly used for writing device drivers, emulators, and video games.

- **Serverless applications support:** Traditional frameworks with their large memory usage and slower boot-up times are not well-suited for serverless application development. Micronaut is specifically designed to address these challenges by ensuring a minimal runtime memory footprint and sub-second boot-up times. This makes Micronaut a practical choice for building serverless applications. Additionally, Micronaut provides native support for popular cloud platforms used in serverless function development

- **Language-agnostic framework:** Micronaut supports multiple programming languages, including Java, Kotlin, and Groovy. This language flexibility allows developers to choose their preferred language when considering cloud requirements. For example, Groovy may be a suitable option for IoT applications.

- **Support to GraalVM:** Many applications built with Micronaut can be compiled ahead of time into a native image compatible with GraalVM. GraalVM has the capability to execute Java applications as machine code, resulting in substantial performance improvements. When a Micronaut application is compiled into a GraalVM native image, it achieves ultra-fast startup times, typically measured in milliseconds.

3.4.2 Comparison of Startup Times: Micronaut vs Traditional Frameworks

[20] conducted a brief benchmark study to compare the application startup durations between Micronaut and a well-known conventional framework. The chart3.4 illustrates the startup times for both Micronaut and the traditional framework.

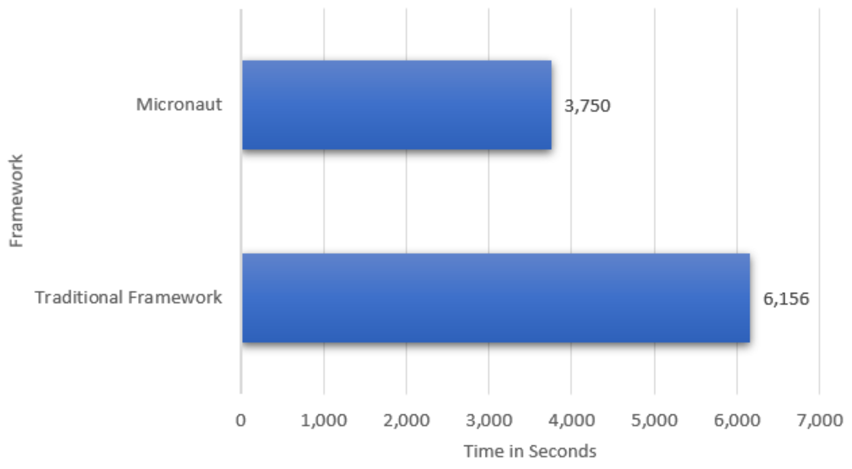


Figure 3.4: Startup times for a traditional framework versus Micronaut, graph taken from [20]

As shown in 3.4 the traditional framework took 6,156 milliseconds to boot up whereas Micronaut took only 3,750 milliseconds. This time difference in booting up the application is significant and sets Micronaut as a convenient framework for developing cloud-native and rapid microservices.

3.5 AWS- Amazon Web Services

[21] mentions that Amazon offers a comprehensive suite of IT tools that enable organizations to establish customized virtual environments, ensuring full control over their configurations. Amazon Web Services (AWS) provides to both organizational and IT development needs. While the cost-effectiveness and efficiency of migrating to the cloud are attractive to security experts, this transition introduces various security risks and compliance considerations. To approach these concerns, AWS has implemented a range of features and services, the most popular and widely utilized services among these are Amazon S3 [15] and Amazon EC2[16]. This service is promoted as offering substantial computing power, potentially involving numerous servers, at a lower cost and significantly faster pace compared to constructing a physical server infrastructure. Diagram 3.5 provides a visual overview of the AWS architecture. Here S3 denotes Simple Storage Service, enabling users to store and access a range of data through API requests.

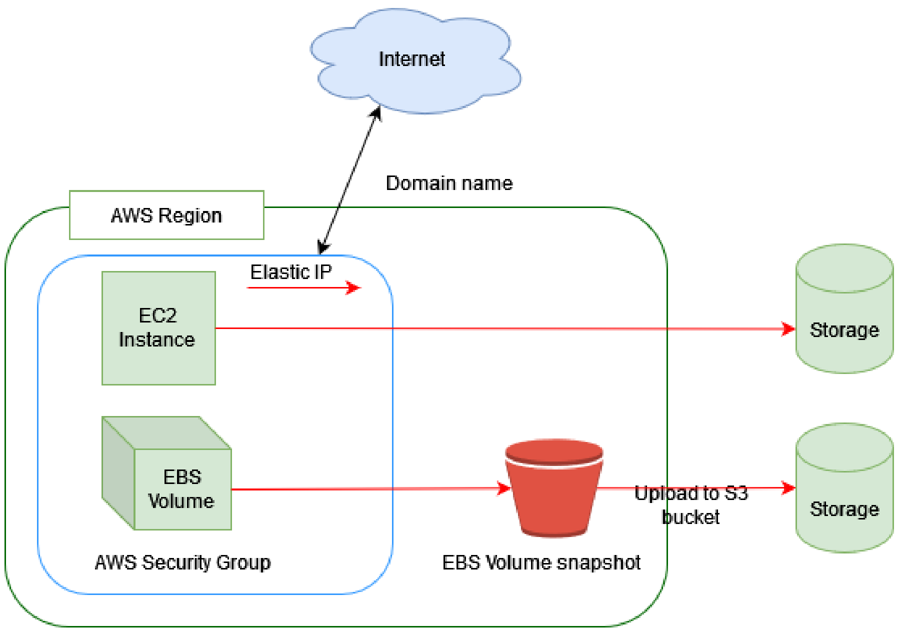


Figure 3.5: Amazon Web Services basic architecture, graph taken from [21]

Some of the AWS components that worth mentioning from the diagram 3.5 are:

AWS Region

According to [24] Amazon's cloud computing resources are distributed across various global locations, known as AWS Regions. Each of these regions constitutes an independent geographic area and includes multiple isolated sites called Availability Zones.

[15]Amazon S3 is a service for storing objects that provides scalability, data accessibility, security, and high performance.[22]

[16]Amazon Elastic Compute Cloud (Amazon EC2) is a web-based service offering secure and easily adjustable computing capacity in the cloud, ensuring a safe environment for running various applications.[23]

Domain Name

According to [25], a collaborative group of trusted internal AWS KMS entities in a specific AWS Region is known as a domain. This domain includes a collection of trusted entities, a set of regulations, and a series of confidential keys known as domain keys. These domain keys are distributed among the HSMs that belong to the domain. The name is used to identify the domain.

EBS Volume

According to [26], an Amazon EBS volume is a robust storage device at the block level that can be connected to your instances. Once you've linked a volume to an instance, you can employ it much like you would a physical hard drive.

AWS Security group

[27] mentions that a security group serves as a virtual barrier for your EC2 instances, regulating both incoming and outgoing traffic. Incoming traffic is managed by inbound rules, while outgoing traffic is governed by outbound rules.

3.6 Microsoft Azure

[28] mentions that Windows Azure, is a Microsoft's public cloud application platform, that provides diverse usage options for any application. For example, it is possible to utilize Windows Azure to develop a web application that operates and stores its data within Microsoft's datacenters. Alternatively, you may choose to employ Windows Azure solely for data storage, with the applications accessing this data running on-premises, outside the public cloud. Windows Azure also facilitates the connection of on-premises applications with one another, as well as the mapping of distinct identity information sets, among other functionalities. Given the extensive array of services offered by this platform, a wide range of capabilities, including those mentioned, are feasible.

To grasp the offerings of Windows Azure, [29] categorize its services and comprehend the functions of its components. Figure 3.6 illustrates a method for achieving this.

According to [29] the structure 3.6 includes the subsequent services and elements.

Backend systems

The diagram's right-hand portion displays the array of backend systems utilized. These may include SaaS platforms, additional Azure services, or web services offering REST or SOAP interfaces. Here, we also encounter the blob storage that is part of the Azure services which played a role in the development of this thesis for accessing the stored objects. The blob storage acts as an initial storage space for the source data before it is used.

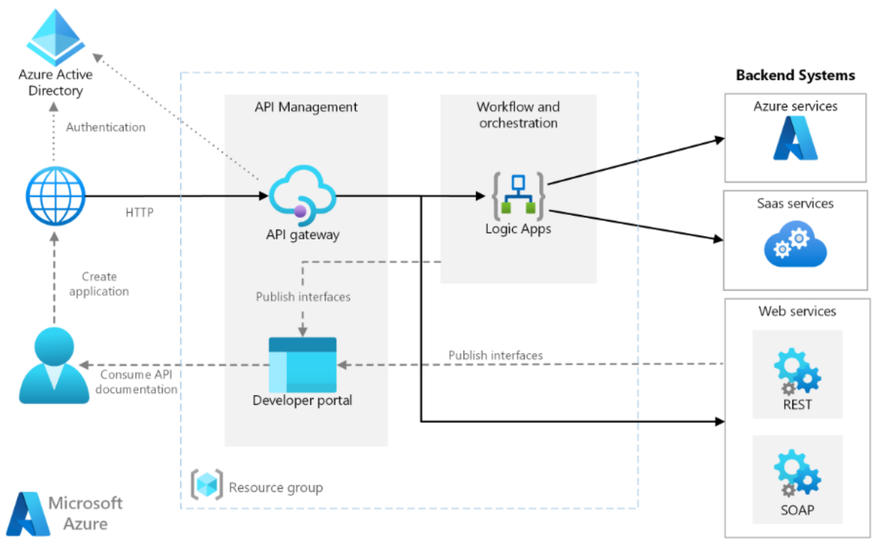


Figure 3.6: Microsoft Azure components and architecture, graph taken from [29]

Azure Logic Apps

In this setup, logic apps start when they receive a web request. You can also combine them for more complicated tasks. Logic Apps use connectors to connect with popular services. There are many pre-made connectors available, and you can also make your own.

3.6.1 Azure API Management

[29] decribes the API Management as two interconnected elements:

API gateway

The API gateway receives HTTP requests and directs them to the backend.

Developer Portal

The portal provides developers with access to documentation and examples of code for making API calls. Additionally, you can carry out API testing within the developer portal.

3.6.2 Resource Group

According to [30] resource group is like a box that contains interconnected resources for an Azure solution. It can include all the resources needed for the solution or just the ones you prefer to manage together.

3.6.3 Azure AD

Azure AD provides an organization with a cloud-based identity and access management solution, linking employees, customers, and partners to their applications, devices, and information while ensuring security[31].

3.7 Helm Charts

[32] mentions that in the Docker[17] compose format, container composition files address the creation of containers[18] but do not directly support service semantics. Containers can offer varying numbers of services. Additionally, this format does not take into account resource allocation constraints. Descriptor files in the kubernetes format address this limitation, but the abundance of deployment and service descriptors can lead to redundant values and more complex handling. Consequently, Kubernetes applications necessitate a more advanced treatment.

To tackle this limitation in Kubernetes stacks, Helm was introduced in mid-2016 as a solution to bundle sets of descriptor files, including templates and detailed metadata, into single archive files for easy deployment and removal. Helm establishes a packaging file format known as Helm Charts, along with client and server components for managing these files. On the server side, the implementation is deployed on top of Kubernetes, while on the client side, the Helm binary enables the creation, testing, deployment of charts, as well as repository searches. The format of Helm charts is outlined informally in an evolving technological documentation.

[17]Docker was designed in order to simplify the creation, deployment and execution of applications using containers. With docker is possible to deploy and expand applications across various environments, ensuring the continuous execution of the code.

[18]Containerization enables users to execute applications in a virtual setting by bundling all required elements, including files, libraries, and other crucial components. Moreover, containers are pivotal in DevOps workflows, serving as a fundamental component in automated software construction and seamless integration into continuous deployment pipelines.

4 Methodology

4.1 Analysis of the Best Java Framework for Cloud-Native Development

When evaluating existing options, multiple factors need to be taken into account, including efficiency, scalability, manageability, and reliability. The most recognized approaches at present are the monolithic architecture and microservices. This topic is currently highly popular, and numerous advancing technologies are under constant exploration and experimentation to identify the optimal solution to adopt.

Matthias Graf[33] performed an analysis of microservices technologies, focusing on their performance and ease of implementation. The performance assessment considered factors such as compile time, application startup, and peak performance. While these factors hold significance, there are numerous other considerations that should be taken into account when deciding on a particular technology.

Roman Kudryashov[34] conducted a study that was both intriguing and valuable, as it not only measured the duration of specific actions but also considered the memory usage, which holds great significance when utilizing cloud services.

Like previously mentioned numerous studies have been conducted regarding the optimal Java framework, but one particularly intriguing work stands out. [2] effectively compiles and consolidates various studies that aim to determine the most suitable Java framework for utilization.

4.1.1 Popularity

[2] points that when evaluating the popularity of the examined technologies, it is valuable to examine their code repositories on GitHub. The service offers the option to "star" a repository, indicating that a user perceives it as remarkable and is satisfied with the content it offers. According to the results4.1 Spring Boot has received approximately 57,000 ratings, Micronaut has 5,000 ratings, and Quarkus has 8,400 ratings. These findings validate the popularity of the Spring product, which is not a surprise given its longer presence in the market. In contrast, both Micronaut and Quarkus are relatively newer offerings, with Quarkus gaining more popularity than Micronaut by over 50 percent, despite being introduced slightly later.

The JAXenter website[35] carried out a interesting comparison by conducting a survey on commonly used technologies for application development. Participants were given the opportunity to express their level of interest in each technology, ranging from "not interesting at all" to "neutral" and "very interesting." In this comparison, Spring Boot emerged as the clear winner4.2. However, it is worth highlighting the remarkable result achieved by Quarkus.

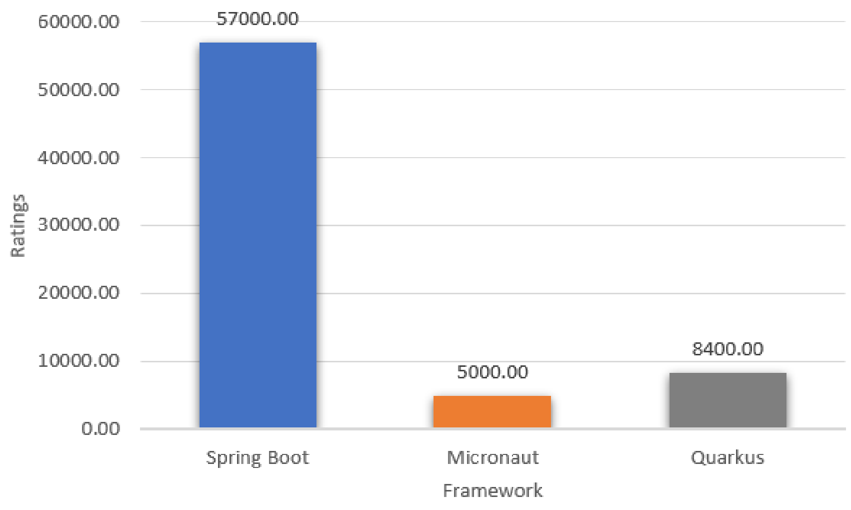


Figure 4.1: Popularity on Github, graph taken from [2]

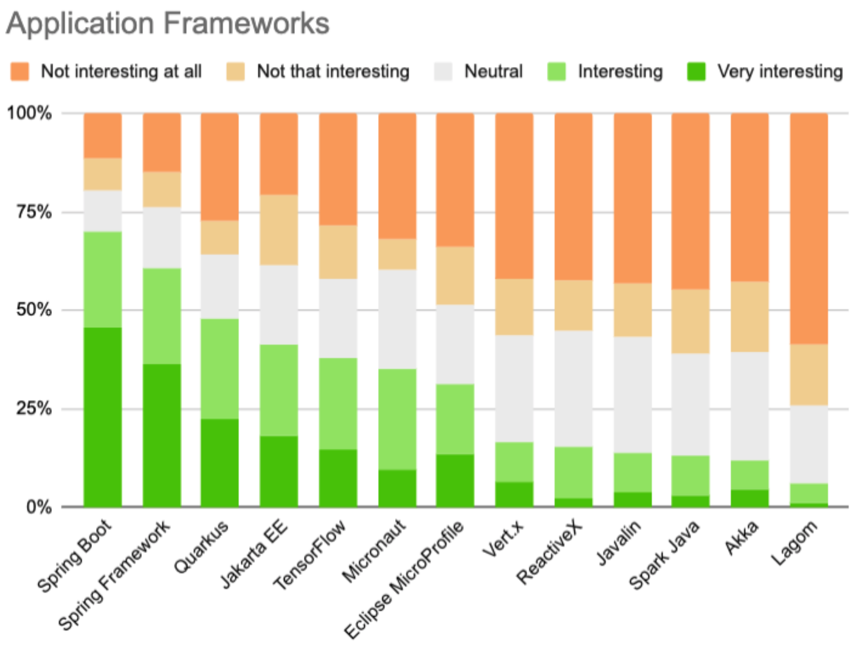


Figure 4.2: Results of a survey conducted by JAXenter [35]

4.1.2 Compile Time

[2] conducted an initial experiment that specifically targeted compile time4.3. This evaluation involved executing the "mvn clean compile" command. From the results, it is evident that Micronaut achieved the fastest time (2.2194 seconds), although its lead over Spring Boot was

marginal. Quarkus, on the other hand, exhibited a slower performance than both, lagging behind by approximately 0.2 seconds.

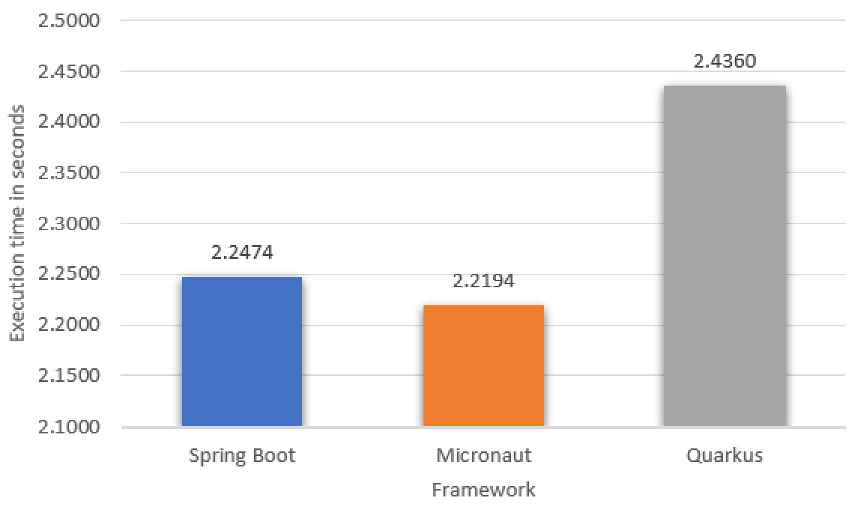


Figure 4.3: Average application compile time, graph taken from [2]

4.1.3 Test Time

The next aspect examined in [2] was the execution time of the tests 4.4. In this scenario, Micronaut demonstrated a significantly superior performance, completing the tests in 7.852 seconds. This result surpassed Quarkus by more than 2 seconds, while Quarkus, in turn, held a slight lead over Spring Boot. This discrepancy could be attributed to the simpler configuration of the class loader in Micronaut.

4.1.4 Startup of the Application

Following that, the evaluation from [2] shifted towards examining the timing of one of the crucial actions for a developer, which is launching the application4.5. Spring Boot scans annotated classes during startup to create beans, whereas the other tested technologies inject dependencies at compile time. Hence, it was anticipated that Spring Boot would be the slowest in this comparison, and indeed, that proved to be the case, with Micronaut emerging as the fastest once again. It's worth noting that this test might yield different results if the applications were executed on native images, where Quarkus could showcase its full potential.

4.1.5 Database Operations

This section explores the analysis of various database operations and their respective performances.

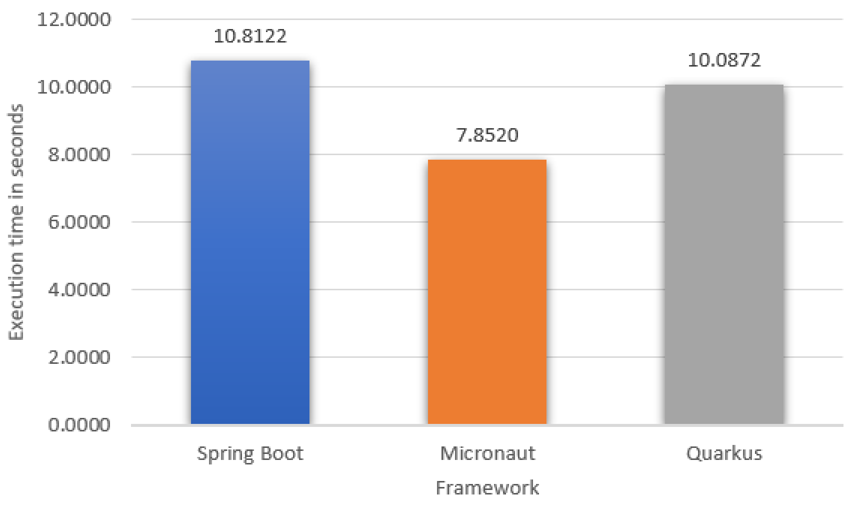


Figure 4.4: Average application test time, graph taken from [2]

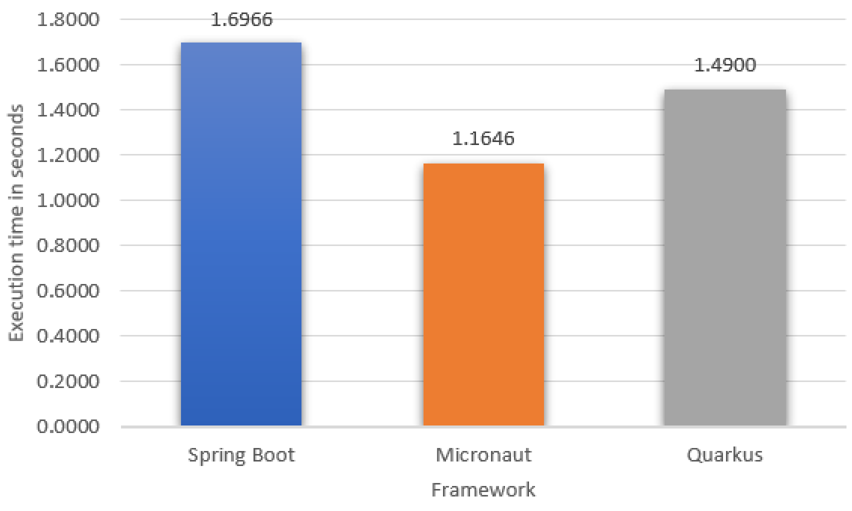


Figure 4.5: Average startup of the application time, graph taken from [2]

Save

[2] examined the process of saving 1000 books (as shown in graph 4.6), Quarkus demonstrated the highest level of efficiency. It outperformed Spring Boot by 50 percent and was approximately twice as fast as Micronaut in terms of saving speed.

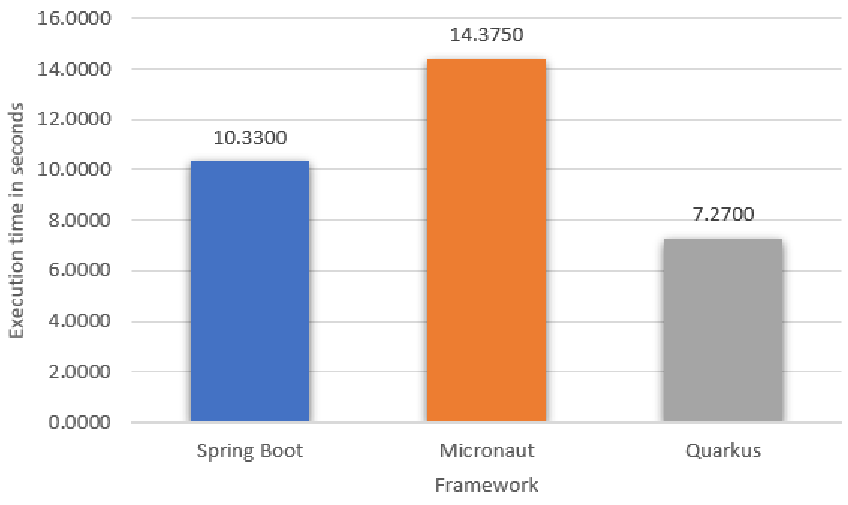


Figure 4.6: Average time of saving data to database, graph taken from [2]

Read

When it came to reading data, Quarkus emerged as the fastest[2]. However, it had a significant advantage over Spring Boot but was considerably slower than Micronaut, as indicated in Table 4.1. It's important to highlight that Quarkus utilizes PanacheRepository[1], their own implementation on top of Hibernate, for managing database data. The developers aimed to create a straightforward method of communicating with the database. The results indicate that one of the major strengths of Quarkus is its speed, making it an attractive option for native solutions.

	Spring Boot[s]	**Micronaut[s]**	**Quarkus[s]**
Write(1000 Cycles)	10.330	14.375	7.270
Read(10,0000 Cycles)	0.152333	2.665000	0.024817

Table 4.1: Average time of reading and writing data to/from database.

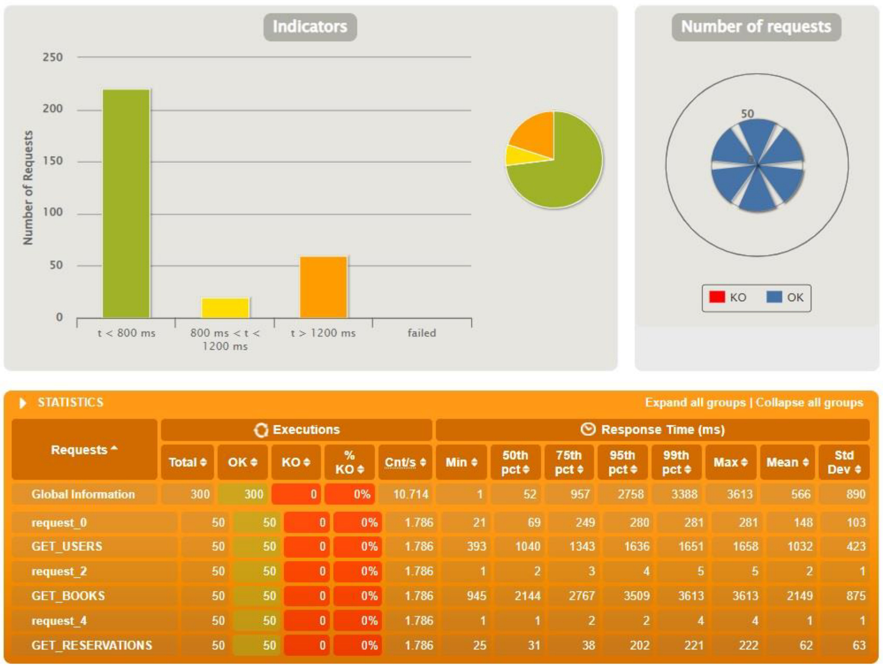
4.1.6 Stability

This subsection refers to the reliability and consistency of the framework's behavior and performance over time.

[1]Panache is a special library designed specifically for Quarkus, It eliminates the need for writing repetitive and standard code typically associated with persistence layers. One of its notable features is the provision of pre-built repositories that can be readily used and conveniently customized for entity classes. [36]

Test for Identical Data

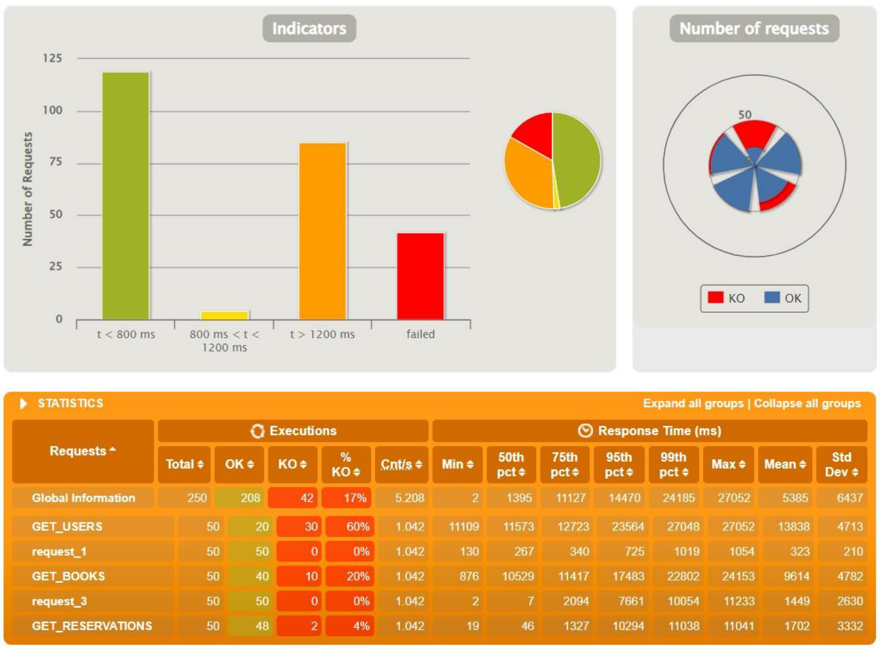
[2] shows the results of the Spring Boot test run and are illustrated in graph 4.7 , with a detailed and user-friendly presentation. Spring Boot demonstrated excellent performance during the test, encountering no significant issues. Upon examining the figure, it can be observed that it is divided into three sections. The top-left corner displays a bar chart indicating the number of queries executed within different time ranges: under 800 ms, between 800 and 1200 ms, and over 1200 ms. The fourth bar represents errors, but no errors were recorded in this case. In the top-right corner, there is a slightly modified pie chart illustrating the accuracy of different query types. In the case of Spring Boot, all queries executed successfully, with no failures. Below the graphs, more detailed statistics are provided, depicted through the aforementioned charts. Additionally, the results for Micronaut 4.8 and Quarkus4.9 are presented below.



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	300	300	0	0%	10.714	1	52	957	2758	3388	3613	566	890
request_0	50	50	0	0%	1.786	21	69	249	280	281	281	148	103
GET_USERS	50	50	0	0%	1.786	393	1040	1343	1636	1651	1658	1032	423
request_2	50	50	0	0%	1.786	1	2	3	4	5	5	2	1
GET_BOOKS	50	50	0	0%	1.786	945	2144	2767	3509	3613	3613	2149	875
request_4	50	50	0	0%	1.786	1	1	2	2	4	4	1	1
GET_RESERVATIONS	50	50	0	0%	1.786	25	31	38	202	221	222	62	63

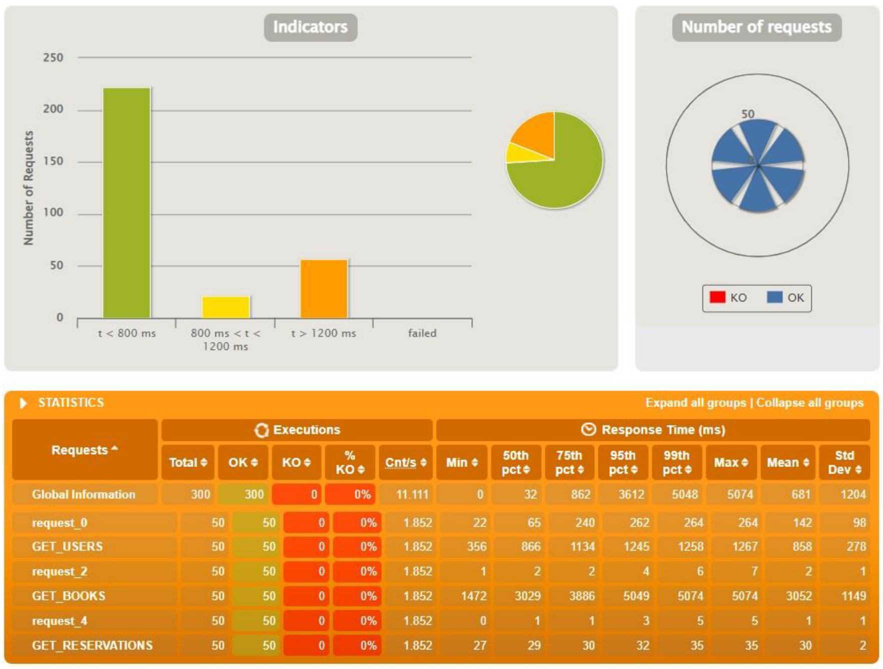
Figure 4.7: Results for test data: 500 users, 1000 reservations, 2000 books, 50 actors - Spring Boot [2]



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	250	208	42	17%	5.208	2	1395	11127	14470	24185	27052	5385	6437
GET_USERS	50	20	30	60%	1.042	11109	11573	12723	23564	27048	27052	13838	4713
request_1	50	50	0	0%	1.042	130	267	340	725	1019	1054	323	210
GET_BOOKS	50	40	10	20%	1.042	876	10529	11417	17483	22802	24153	9614	4782
request_3	50	50	0	0%	1.042	2	7	2094	7661	10054	11233	1449	2630
GET_RESERVATIONS	50	48	2	4%	1.042	19	46	1327	10294	11038	11041	1702	3332

Figure 4.8: Results for test data: 500 users, 1000 reservations, 2000 books, 50 actors - Micronaut [2]



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	300	300	0	0%	11.111	0	32	862	3612	5048	5074	681	1204
request_0	50	50	0	0%	1.852	22	65	240	262	264	264	142	98
GET_USERS	50	50	0	0%	1.852	356	866	1134	1245	1258	1267	858	278
request_2	50	50	0	0%	1.852	1	2	2	4	6	7	2	1
GET_BOOKS	50	50	0	0%	1.852	1472	3029	3886	5049	5074	5074	3052	1149
request_4	50	50	0	0%	1.852	0	1	1	3	5	5	1	1
GET_RESERVATIONS	50	50	0	0%	1.852	27	29	30	32	35	35	30	2

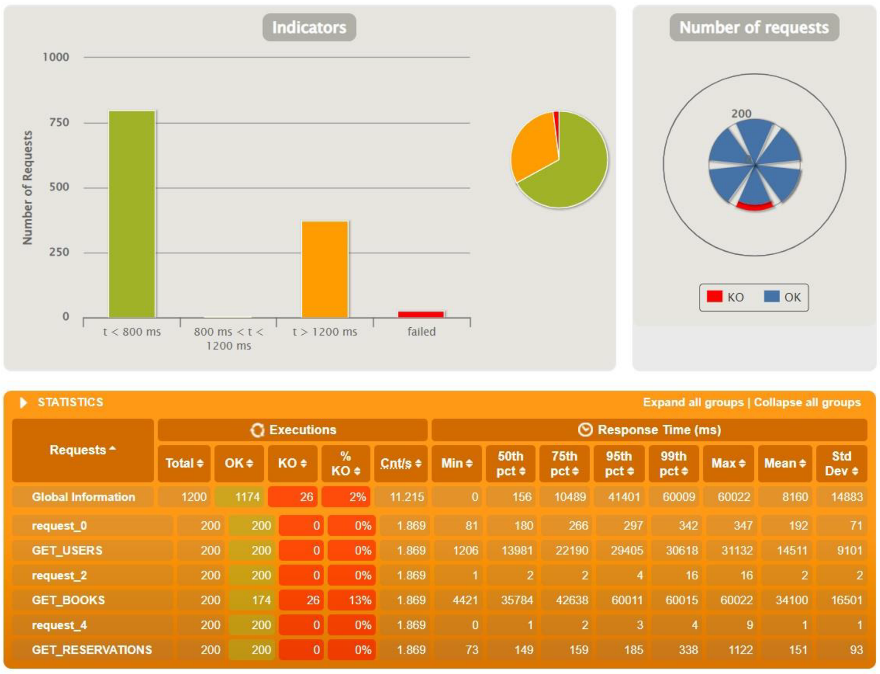
Figure 4.9: Results for test data: 500 users, 1000 reservations, 2000 books, 50 actors - Quarkus [2]

The performance of Spring Boot and Quarkus in handling this task was perfect, achieving a 100 percent efficiency rate. However, when compared to them, Micronaut's results were noticeably poorer. Specifically, 42 queries failed in Micronaut's case. By examining the logs obtained from the Gateway microservice console, [2] identified that these errors occurred due to exceeding the preset time limit.

Achieved limits

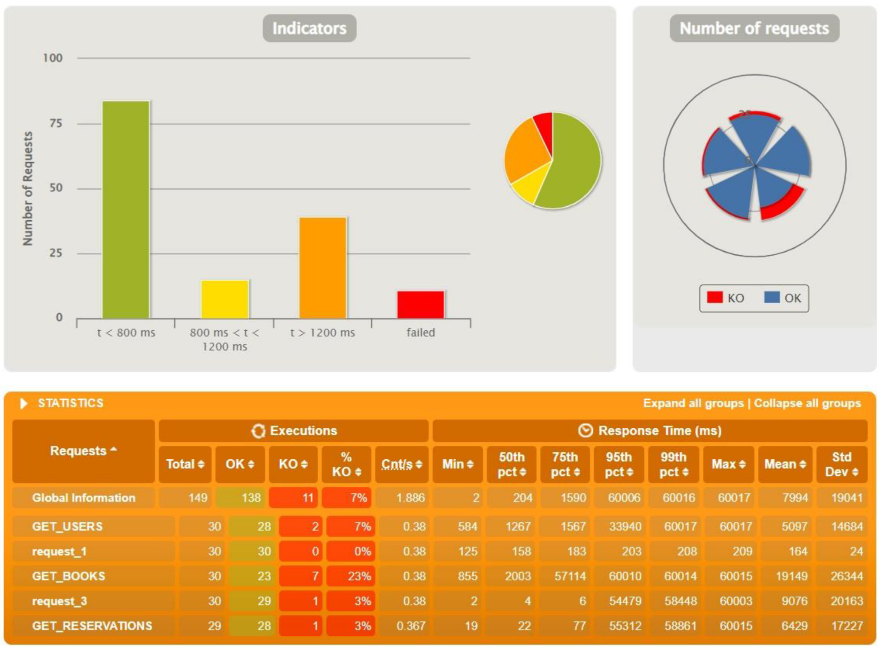
Additional tests were conducted in [2] to find the threshold of occurrence of the first errors. These tests involved increasing the resources stored in the database and the number of actors accordingly. The resulting limits are depicted in the graphs 4.10 for Spring Boot, 4.11 for Micronaut, and 4.12 for Quarkus.

In the load test 4.2, Spring Boot emerged as the clear winner. The initial issues arose when the number of actors reached 200, and the database contained 2000 users, 4000 reservations, and 8000 books. The errors were caused by the default timeout, which was set to 60,000 ms.



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1200	1174	26	2%	11.215	0	156	10489	41401	60009	60022	8160	14883
request_0	200	200	0	0%	1.869	81	180	266	297	342	347	192	71
GET_USERS	200	200	0	0%	1.869	1206	13981	22190	29405	30618	31132	14511	9101
request_2	200	200	0	0%	1.869	1	2	2	4	16	16	2	2
GET_BOOKS	200	174	26	13%	1.869	4421	35784	42638	60011	60015	60022	34100	16501
request_4	200	200	0	0%	1.869	0	1	2	3	4	9	1	1
GET_RESERVATIONS	200	200	0	0%	1.869	73	149	159	185	338	1122	151	93

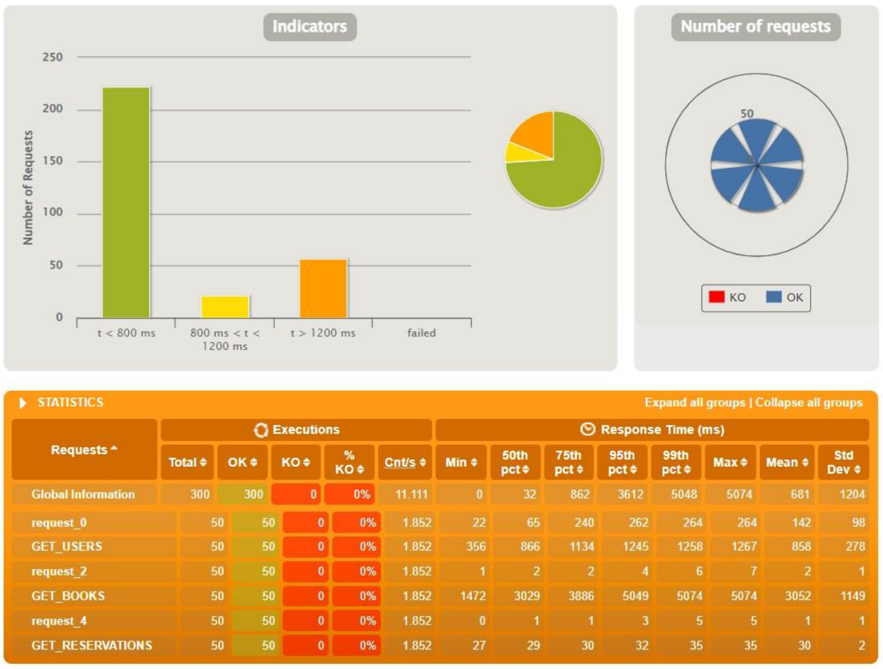
Figure 4.10: Results for test data: 2000 users, 4000 reservations, 8000 books, 200 actors - Spring Boot [2]



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	149	138	11	7%	1.886	2	204	1590	60006	60016	60017	7994	19041
GET_USERS	30	28	2	7%	0.38	584	1267	1567	33940	60017	60017	5097	14684
request_1	30	30	0	0%	0.38	125	158	183	203	208	209	164	24
GET_BOOKS	30	23	7	23%	0.38	855	2003	57114	60010	60014	60015	19149	26344
request_3	30	29	1	3%	0.38	2	4	6	54479	58448	60003	9076	20163
GET_RESERVATIONS	29	28	1	3%	0.367	19	22	77	55312	58861	60015	6429	17227

Figure 4.11: Results for test data: 2000 users, 4000 reservations, 8000 books, 200 actors - Micronaut [2]



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	300	300	0	0%	11.111	0	32	862	3612	5048	5074	681	1204
request_0	50	50	0	0%	1.852	22	65	240	262	264	264	142	98
GET_USERS	50	50	0	0%	1.852	356	866	1134	1245	1258	1267	858	278
request_2	50	50	0	0%	1.852	1	2	2	4	6	7	2	1
GET_BOOKS	50	50	0	0%	1.852	1472	3029	3886	5049	5074	5074	3052	1149
request_4	50	50	0	0%	1.852	0	1	1	3	5	5	1	1
GET_RESERVATIONS	50	50	0	0%	1.852	27	29	30	32	35	35	30	2

Figure 4.12: Results for test data: 2000 users, 4000 reservations, 8000 books, 200 actors - Quarkus [2]

	Spring Boot[s]	Micronaut[s]	Quarkus[s]
Database load	2000 users 4000 reservations 8000 items	500 users 1000 reservations 2000 items	1000 users 2000 reservations 4000 items
Actors	200	30	200
OK requests	574	30	200
KO Requests	26	11	34

Table 4.2: Load limits for tested technologies, Successful amount for queries is OK, and for invalid KO. The abbreviations have been taken from the Gatling data view.

4.1.7 CPU and Memory Usage

A CPU and memory usage test was conducted[2] , and the results are presented in Figures 4.134.144.15. Micronaut had the lowest resource consumption. Quarkus had slightly higher consumption, but its performance was significantly better when compared to Spring Boot. These findings support the notion that newer technologies are specifically designed for serverless environments. They prioritize minimizing startup time and reducing memory usage, as costs are based on the actual execution time of functions. One contributing factor to Spring's high memory consumption is its reliance on the aforementioned reflection mechanism, which is not ideal for optimization purposes.

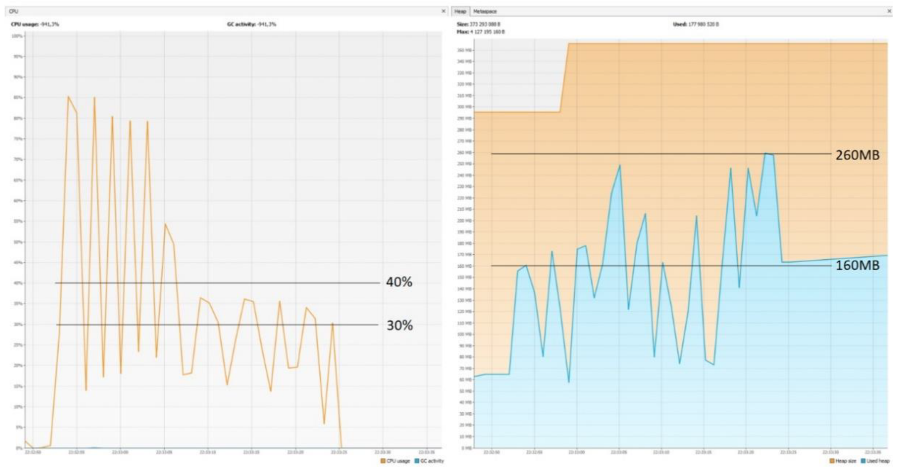


Figure 4.13: CPU usage: 30-40% ; memory usage: 160-260 MB - Spring Boot [2]

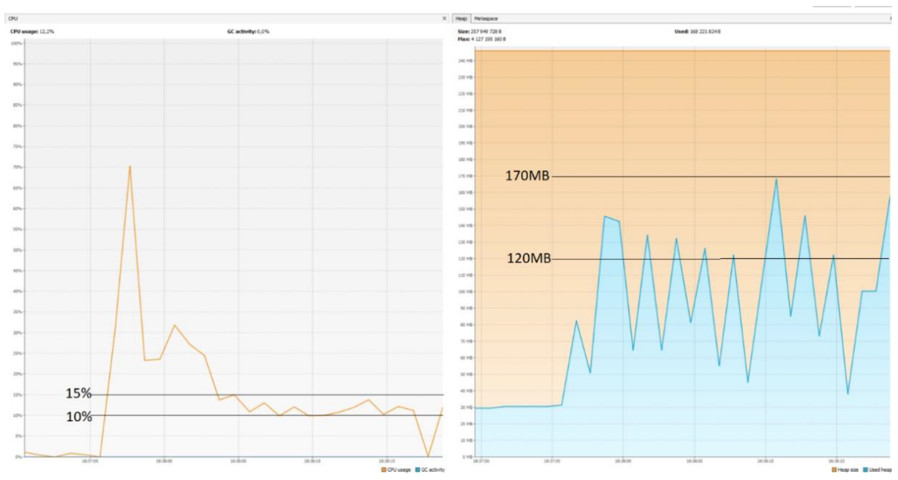


Figure 4.14: CPU usage: 10-15% ; memory usage: 120-170 MB - Micronaut [2]

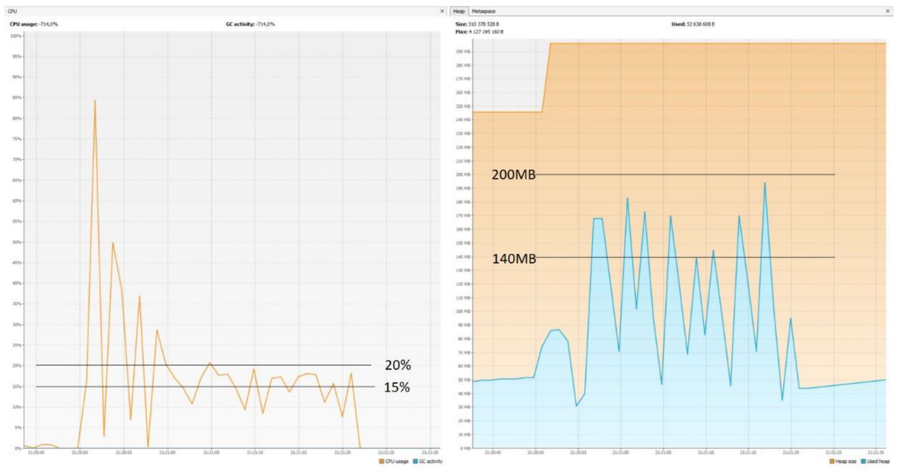


Figure 4.15: CPU usage: 15-20% ; memory usage: 140-200 MB - Quarkus [2]

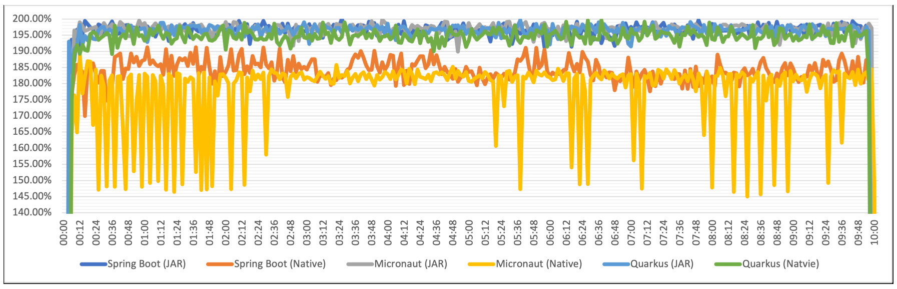
4.1.8 Stress Test Results

[1] conducted stress tests that aim to create a demanding workload on the application, pushing it to the upper limits of the virtual machine's capabilities and resulting in significant CPU usage. The objective of the conducted average load tests was to evaluate and compare the performance of the applications under typical or average workload conditions. The tests scenarios that were categorized as stress tests presented in Figures accordingly:

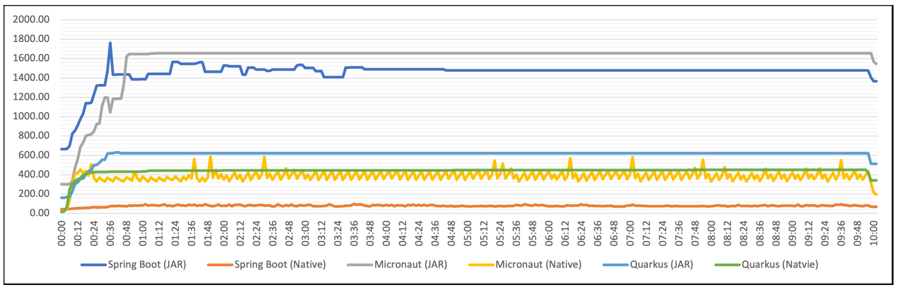
- SingleGreeting 4.16
- GreetingSSE 4.17
- CreateFetchDelete 4.18

- MediumNumberSet 4.19

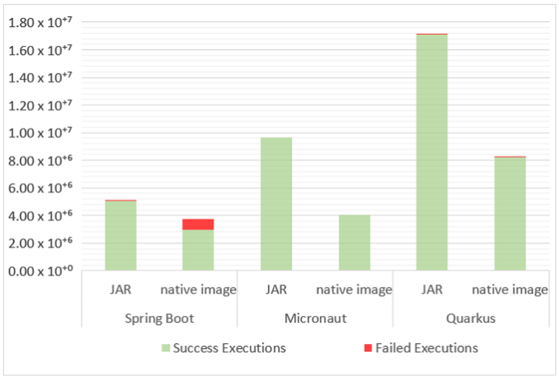
It is important to note that in the CreateFetchDelete test, the native image version of the Micronaut framework was not included in the comparison. This decision was made because there were significantly more failures compared to successful executions, as shown in Figure 4.18c. A similar situation was observed in the case of MediumNumberSet for Spring Boot (native image) and both JAR and native image versions of Micronaut, as depicted in Figure 4.19c. The same approach was applied in these cases as well.



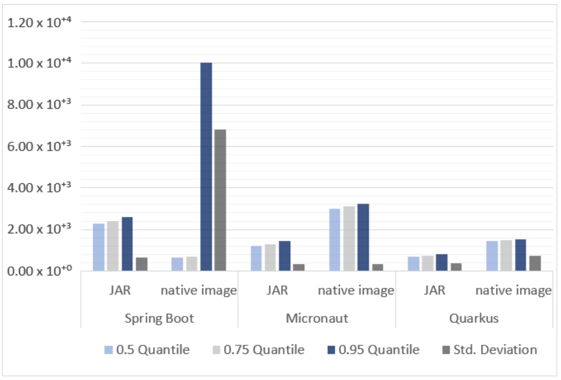
(a) Average CPU usage(%).



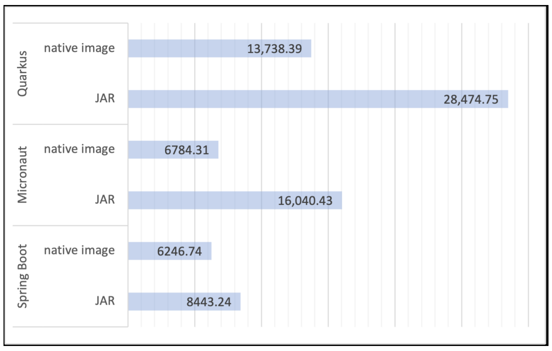
(b) Average RAM usage(MiB).



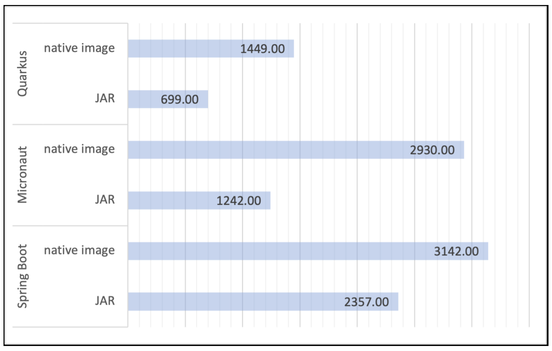
(c) Number of failed and success scenario executions.



(d) Quantile´s and std. deviation of response time(ms)

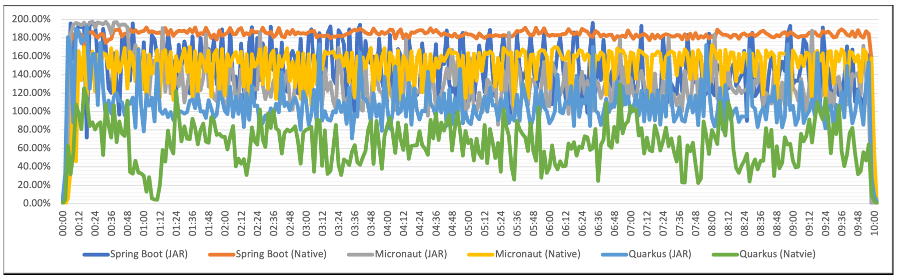


(e) Number of scenario executions(cnt/s)

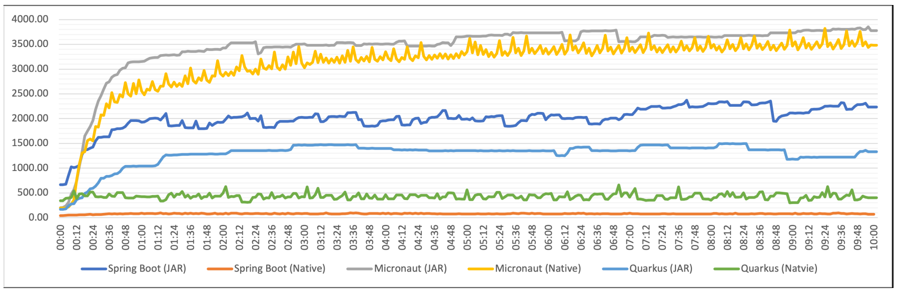


(f) Average response time(ms)

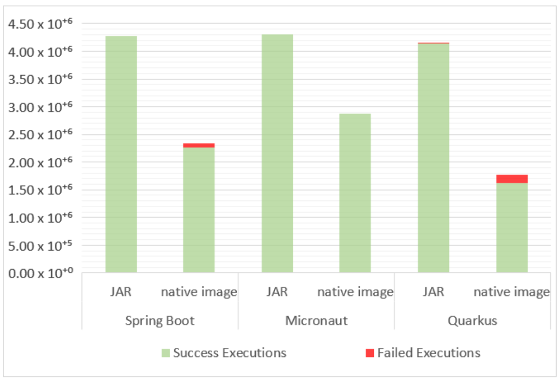
Figure 4.16: SingleGreeting test Results



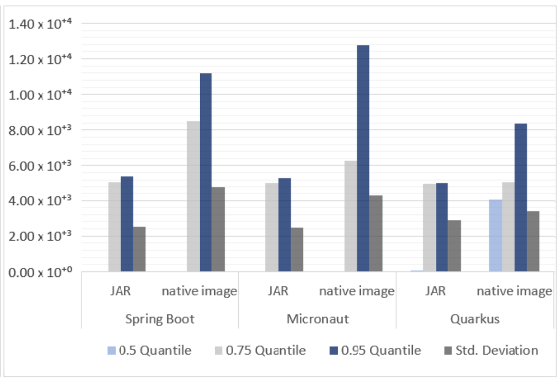
(a) Average CPU usage(%).



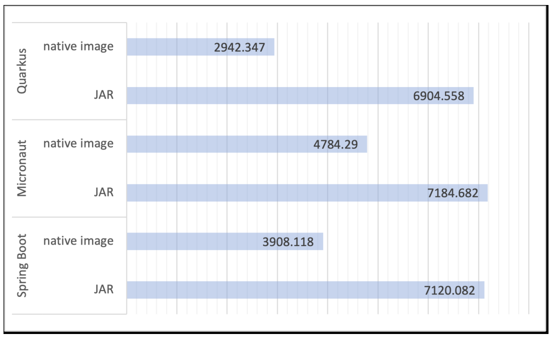
(b) Average RAM usage(MiB).



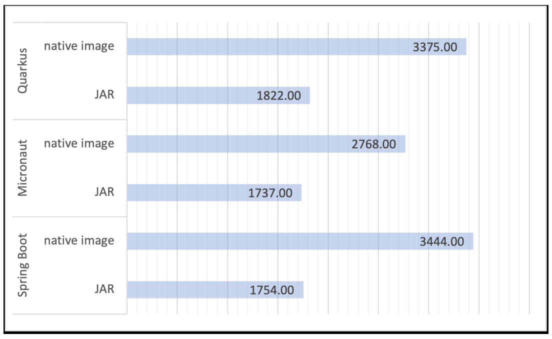
(c) Number of failed and success scenario executions.



(d) Quantile´s and std. deviation of response time(ms)

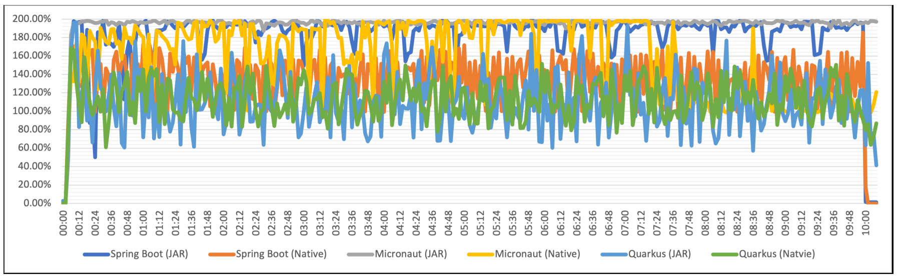


(e) Number of scenario executions(cnt/s)

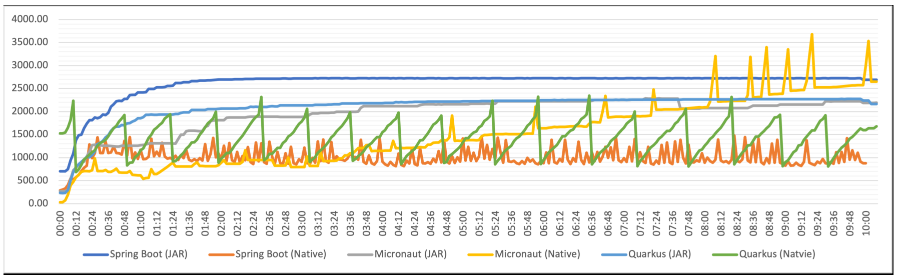


(f) Average response time(ms)

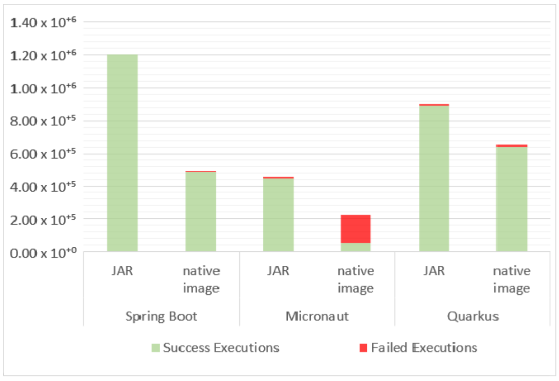
Figure 4.17: GreetingSSE test results

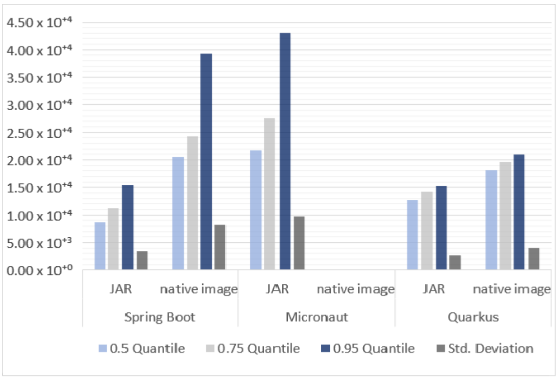


(a) Average CPU usage(%).

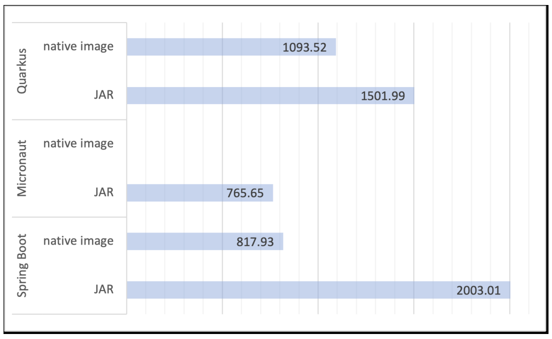


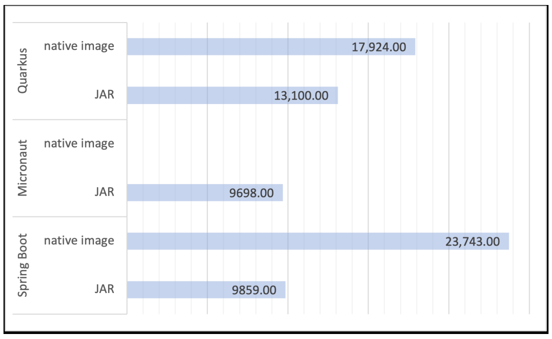
(b) Average RAM usage(MiB).





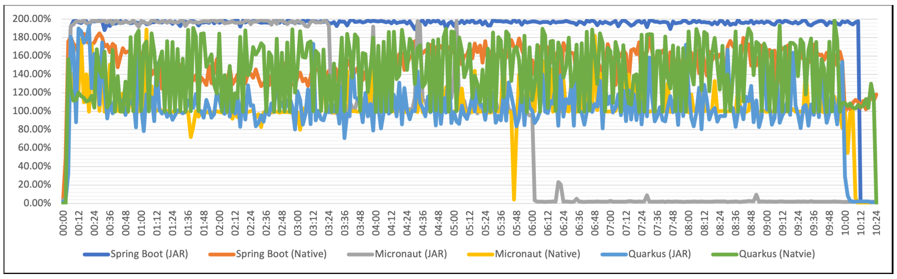
(c) Number of failed and success scenario executions. (d) Quantile´s and std. deviation of response time(ms)



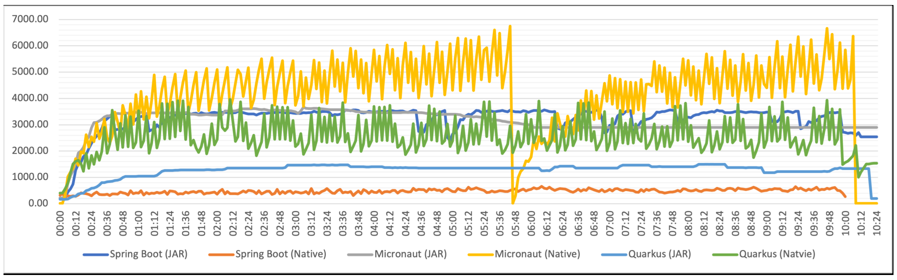


(e) Number of scenario executions(cnt/s) (f) Average response time(ms)

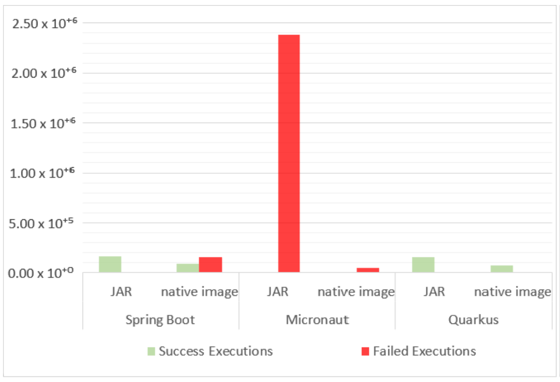
Figure 4.18: CreateFetchDelete test results



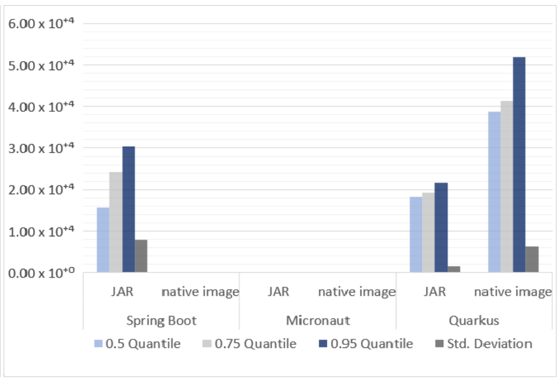
(a) Average CPU usage(%).



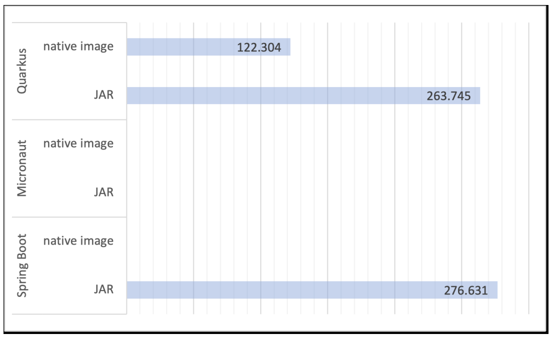
(b) Average RAM usage(MiB).



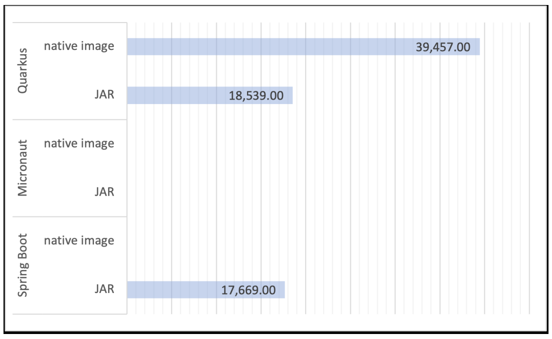
(c) Number of failed and success scenario executions.



(d) Quantile´s and std. deviation of response time(ms)



(e) Number of scenario executions(cnt/s)



(f) Average response time(ms)

Figure 4.19: MediumNumberSet test results

4.2 Development tools and dependencies

In the development of the microservice and the orchestrator, IntelliJ IDE was used along with the following dependencies:

1. **spring-boot-starter-web**: This dependency facilitated the construction of RESTful applications using Spring Boot.

2. **S3**: Employed for referencing objects stored in S3 buckets.

3. **spring-boot-maven-plugin**: This Maven plugin extends support for Spring Boot in Apache Maven. It enables the packaging of executable JAR or WAR archives, running Spring Boot applications, generating build information, and initiating the Spring Boot application before executing integration tests.

4. **com.microsoft.azure**: This package contains authentication connectors to Active Directory for the JDK.

Figure 4.20 illustrates the dependencies employed in the development of the applications.

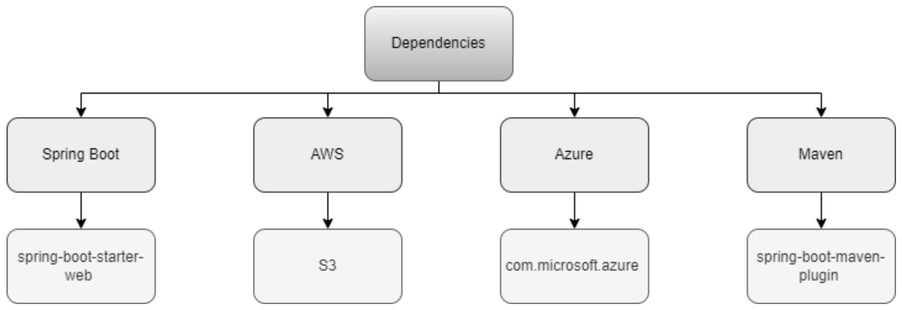


Figure 4.20: Dependencies diagram

5 Implementation

The diagram 5.1 illustrates the architecture of the implemented solution for approaching the problem outlined in this thesis.

From bottom to top, we observe that the input is a URL, which is then passed to the Proxy Orchestrator (Sidecar). The Proxy Orchestrator communicates with the microservice through ports 8080 and 8081. The microservice, in turn, is responsible for making the appropriate API call. Both the container and the microservice are Docker images and have a Helm Chart defining environment variables, connection strings, region (in the case of AWS). In order to establish communication between the Proxy Orchestrator (Sidecar container), a REST template[1] was implemented.

When the microservice receives a request from the proxy orchestrator, it locates the corresponding API, which could involve a download, a listing, or an upload operation, and determines the cloud provider from which to query the information, which can be either AWS or Azure.

In the case of Azure, it accesses the Azure Blob Storage. For this to occur, there must be an Azure container associated with a resource group, which in turn is linked to an Azure subscription.

For AWS, the process differs slightly. Instead of Azure Blob Storage, it interacts with the defined bucket and locates the objects within the previously specified region, which, for the purposes of this thesis, is set to eu-central-1, for future modifications, simply adjusting this parameter in the Helm chart should suffice to implement the desired changes.

Finally, the diagram provides the outcome of the executed query.

5.1 Structure

The solution was methodically structured into distinct elements. On one side, there's the Microservice, including the APIs. On the other side, we have the Proxy Orchestrator, tasked with receiving requests and subsequently transmitting the information to the microservice to appropriately route the request to the cloud storage.

[1]A RestTemplate is a synchronous tool used for making HTTP requests. This operates at a higher level as it utilizes an HTTP client library such as JDK HttpURLConnection or Apache HttpClient to execute these requests. The underlying HTTP client library handles the intricate aspects of communication over HTTP, while RestTemplate extends its functionality by enabling the conversion of request and response data in JSON or XML formats to Java objects.[37]

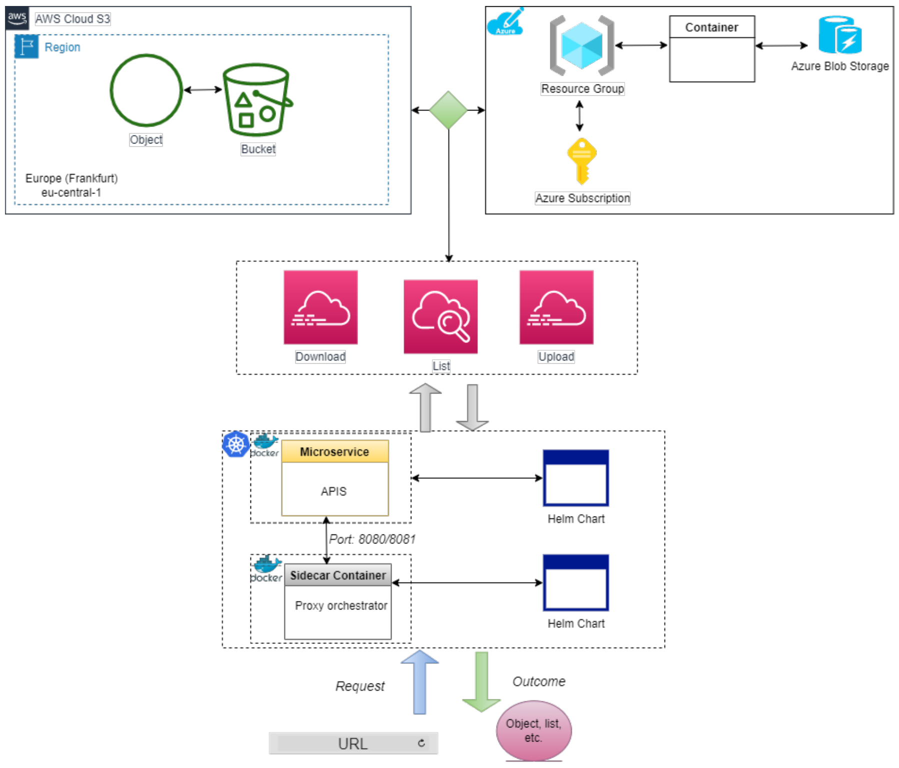


Figure 5.1: Solution diagram architecture

5.1.1 Microservice

The primary motivation for implementing a microservice was to develop a software composed of small, self-contained services, each handling a specific task. The goal was to ensure that these services collaborate together to form a unified and complete application.

One of the advantages that we can achieve with the implementation of our microservice are:

Modularity and Scalability

Our microservice aims to decompose the application into smaller, more manageable components. This modular approach enables the individual scaling of services according to their specific resource needs. As depicted in Figure 5.2, we delineated the divisions between the utilized regions, the developed APIs, and the properties employed for accessing cloud storages.

Easy Scaling

Our microservice enables the independent scaling of each component based on its particular usage. For example, if you need to expand the solution to more AWS regions, configuring this property can be done easily in the region service without affecting the rest of the application.

Easier Maintenance and Updates

With the implemented microservice, you can update or replace one service without having to redeploy the entire application. This makes maintenance and updates less risky and more manageable. For instance if a new API needs to be added, you only need to developed new function in the controller.

Scalable Teams

Organizing teams around individual microservices promotes specialization and independence. This approach can result in more optimized development and maintenance. For example, if a complex task involving a specific cloud provider, Azure, requires expertise, it can be assigned to a skilled team member, ensuring focused and efficient execution.

Better Technology Fit

Different services can use different technologies based on their requirements. For example, a service that handles real-time communication might use Node.js, while a service dealing with big data applications and server-side technologies might use Java, like is the case of this thesis.

Faster Development and Deployment

The microservice developed in this thesis is a component of a larger project, where several other microservices are also implemented. Using microservices in this solution enabled us to focus on the benefits of smaller teams working on individual services, resulting in quicker development cycles. Furthermore, because each service operates independently, they can be deployed and updated without impacting the entire final application.

Components

The microservice is structured as shown in Figure 5.2, The microservice consists of three key components:

1. The controller, designated with the purpose of managing the APIs.
2. The application properties file, containing important specifications such as the Azure connection string, Azure container name, AWS access key ID, AWS secret Access Key, AWS region, and the designated server port for communication.
3. The AWS service, which is responsible for handling the region defined in the application properties variable.

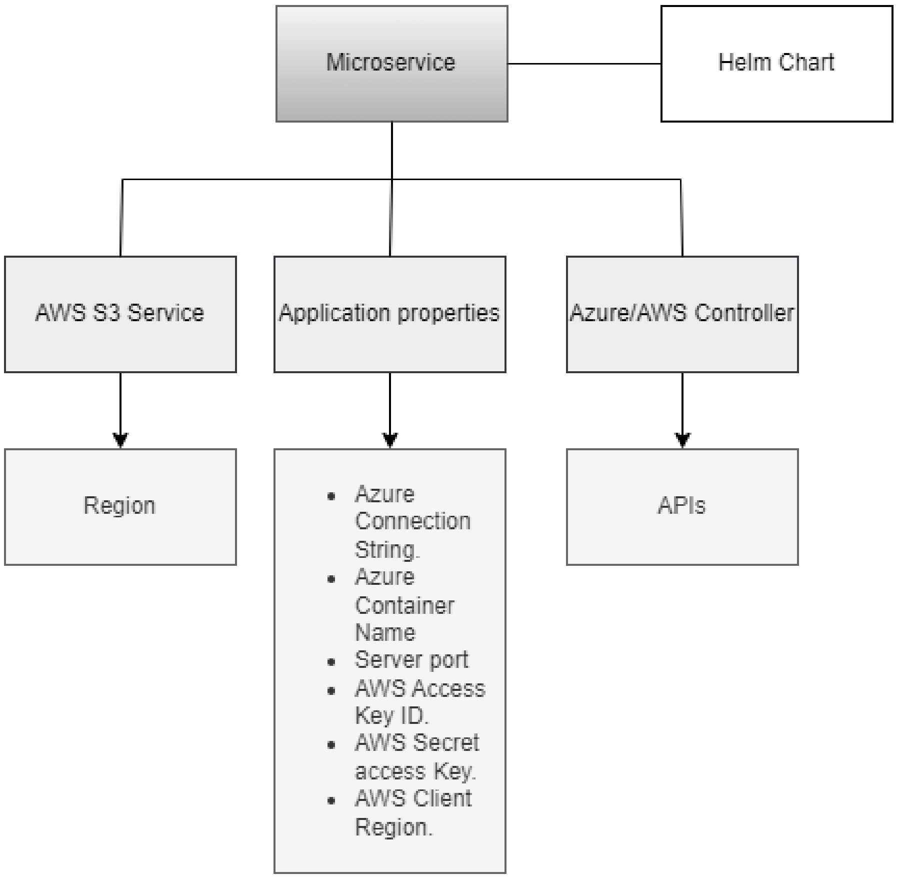


Figure 5.2: Microservice Composition

5.1.2 Proxy Orchestrator

While the development of the proxy orchestrator component was not within the scope of this thesis, it is essential to acknowledge its important role as a tangible application service in the development of the project, effectively complementing the microservice.

The purpose of applying a proxy orchestrator to the project was to have an additional container that could run alongside the microservice container within the same pod in a container orchestration platform, for the purpose of this thesis, using Kubernetes. One of the main advantages of applying this solution was to enhance and extend the functionality of the main container, hosting the microservice, without directly modifying it.

It's also important to note that by incorporating the proxy orchestrator, we could achieve the following:

Separation of Concerns

It allows us the separation of functionalities such as the reception of the request and the setup of environment variables into two distinct containers, the microservice and the proxy orchestrator, making it easier to manage and update each component independently.

Modular and Scalable

Allow us to enable a modular framework, permitting the addition or removal of various components without impacting the core application container, for instance if in the future a security proxy must be implemented in order to handle tasks like authentication, encryption, and load balancing it can be added without affecting the main container by modifying the proxy orchestrator controller.

Resource Sharing

The proxy orchestrator has also the ability to share resources with the microservice, such as network namespaces, storage volumes and, for the case of this thesis, the environment variables.

Dynamic Configuration

Another advantage of employing the proxy orchestrator is its ability to dynamically update configurations without disrupting the microservice, enabling real-time adjustments. For example, this includes the ability to modify the communication port wihtout disruption.

Proxy orchestrator components

The Proxy Orchestrator, as illustrated in Figure 5.3, consists of a Proxy Controller responsible for receiving the URL and routing it to the microservice, enabling it to locate the appropriate API for execution. Additionally, the orchestrator includes a Helm Chart, which contains three main components outlined below:

1. **Values**: This object grants access to the parameters passed into the chart.
2. **Deployment**: In this context, it is responsible for deploying the environment variables defined in the service.
3. **Service**: This is where the environment variables are specified.

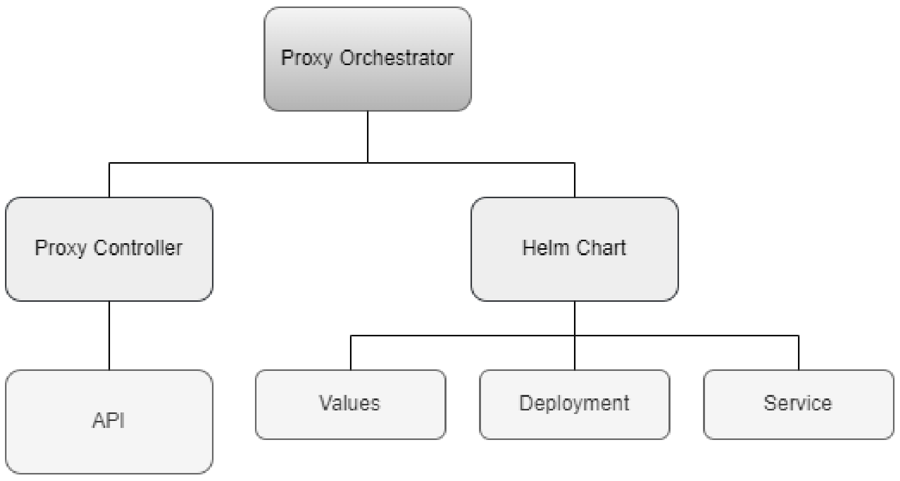


Figure 5.3: Proxy orchestrator components

6 Evaluation

This chapter will analyze the most appropriate Java framework for use, as well as evaluate the transfer and download speeds of the developed microservice in comparison to the web interfaces of each of the cloud providers. Additionally, we will examine code portability.

6.1 Java Framework

As depicted in Table 6.1 and according to [2] the newer contenders to Spring Boot, Micronaut and Quarkus, outperform it in various crucial aspects like application launch time and resource usage. This advantage derives from the fact that dependencies are integrated during the compilation phase, leading to enhanced efficiency. Nonetheless, when subjected to stress tests for handling excessive loads, Spring Boot demonstrated greater stability compared to its counterparts. Regarding performance studied in [1], it's worth noting that both the Quarkus framework, along with its older counterpart Spring Boot, delivered strong performance. Micronaut also managed to attain competitive results, but it faced challenges during stress tests, causing it to fall behind its rivals.

Results	**Excellent**	**Good**	**Deficient**
Compile time	Micronaut	Spring Boot	Quarkus
Test time	Micronaut	Quarkus	Spring boot
Startup of the application	Micronaut	Quarkus	Spring boot
Database Operations(Save)	Quarkus	Spring Boot	Micronaut
Database Operations(Read)	Quarkus	Spring Boot	Micronaut
Stability (Test for identical data)	Spring Boot	Quarkus	Micronaut
Achieved Limits	Spring Boot	Quarkus	Micronaut
Request per second	Micronaut	Quarkus	Spring boot
CPU and Memory Usage	Micronaut	Quarkus	Spring boot
Compilation time for JAR Files	Spring Boot	Quarkus	Micronaut
Compilation time for native image	Micronaut	Quarkus	Spring boot
Startup Time	Micronaut	Quarkus	Spring boot
Docker Image and Executable File Size	Micronaut	Quarkus	Spring boot
Stress results	Quarkus	Spring boot	Micronaut

Table 6.1: Evaluating JVM Frameworks for Building Microservices: A Comparative Analysis

Although Micronaut demonstrated good performance results in [2], [1] conducted a comprehensive stress test, which revealed that Micronaut did not perform as well as Quarkus and

Spring Boot. Based on these findings, [1] did not recommend Micronaut. Moreover, factors like comprehensive documentation and robust support, as outlined in [3], suggest that Spring Boot may offer greater convenience for the development of this thesis. Nevertheless, it's important to acknowledge that there is a potential for Quarkus to enhance its performance and support aspects in the future. This potential evolution could potentially make Quarkus a more competitive option in these areas.

6.2 Upload time

In table 6.2, a comparison is presented between the upload times using the Azure API and Azure Portal[1]. Table 6.3 illustrates a similar comparison between the upload times using the AWS API and AWS Management Console[2]. This evaluation was conducted under the conditions of a 5mb/s internet upload speed, utilizing Postman [3] as the upload tool. The configuration included setting the Content-Type key to "multipart/form-data" in the header option. In the body, the "form data" option was selected, with "file" specified as the key and corresponding to the file uploaded to each of the storage platforms.

6.2.1 Azure

As shown in table 6.2 and figure 6.1, the Azure portal demonstrates significantly shorter times compared to the custom Azure API developed. The difference, however, is not particularly pronounced, as illustrated in the graph, where the upload times to Azure Blob Storage appear quite similar.

File Size (MB)	Azure API (hh:mm:ss)	Azure Portal (hh:mm:ss)
60	00:00:52	00:00:46
150	00:02:49	00:02:24
300	00:04:17	00:04:14
550	00:08:34	00:08:29
1000	00:14:53	00:14:36
5000	01:14:00	01:13:31

Table 6.2: Evaluating upload time between Azure API and Azure Portal

[1]The Azure portal is a unified web-based console that offers an alternative to using command-line tools.s[38]
[2]The AWS Management Console is a web-based application that contains a wide range of service consoles used for the administration of AWS. resources[39]
[3]Postman serves as an API platform designed to facilitate the creation and utilization of APIs.[40]

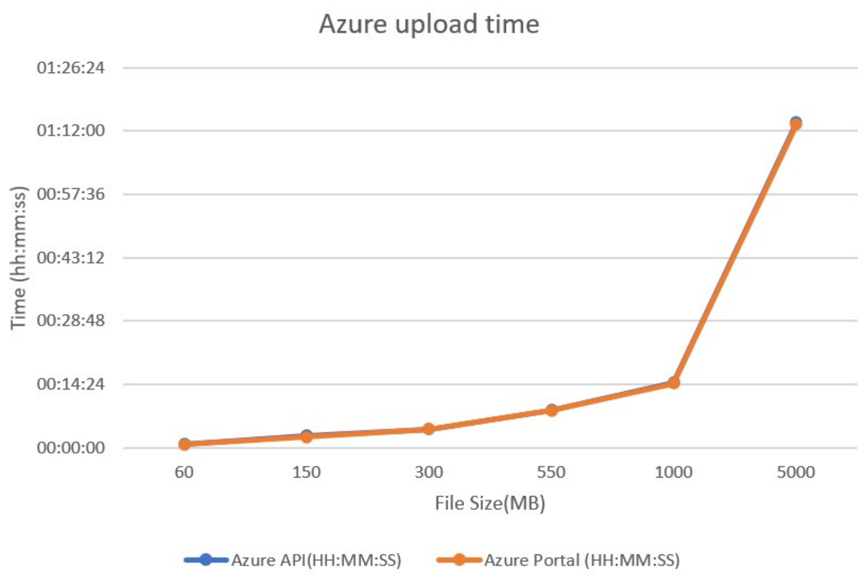


Figure 6.1: Upload time between Azure API and Azure Portal

6.2.2 AWS

In the case of AWS, as indicated in Table 6.3 and Figure 6.2, the AWS Management Console exhibits notably quicker times in contrast to the custom AWS API that was created. Nevertheless, the distinction is not remarkably prominent, as depicted in the graph, where the upload times to S3 Storage seem quite comparable.

File Size (MB)	AWS API (hh:mm:ss)	AWS Management Console (hh:mm:ss)
60	00:00:56	00:00:51
150	00:02:45	00:02:06
300	00:04:30	00:04:20
550	00:08:22	00:08:15
1000	00:16:02	00:15:22
5000	01:19:13	01:17:12

Table 6.3: Evaluating upload time between AWS API vs AWS Console

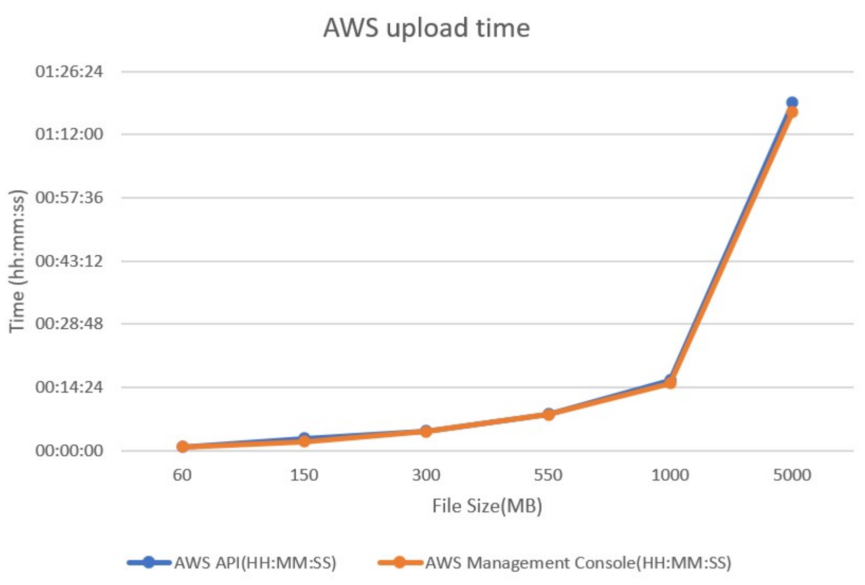


Figure 6.2: Upload time between AWS API and AWS Management Console

6.3 Download time

In table 6.4, a comparison is presented between the download times using the Azure API and the Azure Portal, table 6.5 illustrates a similar comparison between the download times using the AWS API and AWS Management Console. This evaluation was conducted under the conditions of a 50mb/s internet download speed.

6.3.1 Azure

As indicated in table 6.4 and depicted in graph 6.3, the Azure portal exhibits notably quicker downloads in contrast to the Azure API that was created. However, the distinction is not remarkably visible, as demonstrated in the graph, where the download times from the Azure Blob Storage seem quite comparable.

File Size (MB)	Azure API (hh:mm:ss)	Azure Portal (hh:mm:ss)
60	00:00:10	00:00:09
150	00:00:27	00:00:26
300	00:00:54	00:00:52
550	00:01:42	00:01:39
1000	00:03:01	00:02:57
5000	00:15:11	00:14:53

Table 6.4: Evaluating download time between Azure API and Azure Portal

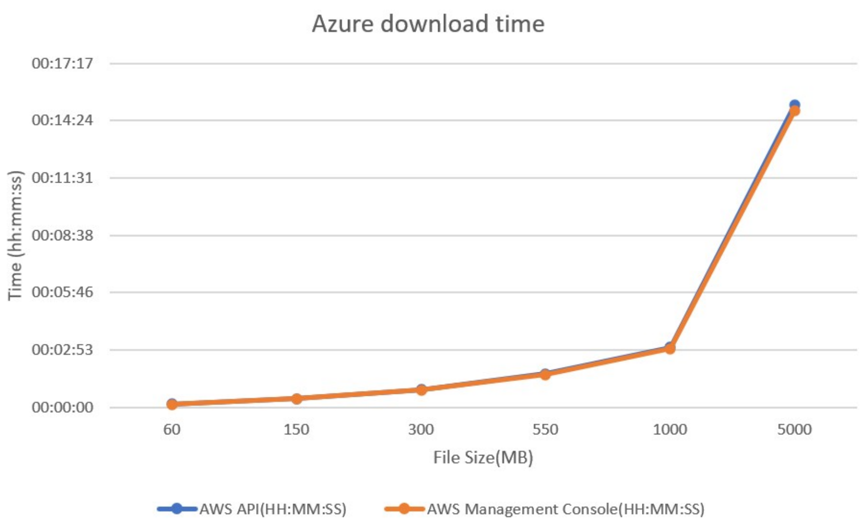


Figure 6.3: Download time between Azure API and Azure portal

6.3.2 AWS

Regarding AWS, as shown in Table 6.5 and graph 6.4, The AWS API exhibits noticeably faster download speeds when compared to the AWS Management Console, which was quite unexpected. Nonetheless, the disparity becomes more noticeable once we reach the 5GB threshold, as depicted in the accompanying graph. Apart from this slight difference, the download times from S3 Storage seem relatively consistent and comparable.

File Size (MB)	AWS API (hh:mm:ss)	AWS Management Console (hh:mm:ss)
60	00:00:15	00:00:15
150	00:00:29	00:00:43
300	00:01:10	00:01:22
550	00:03:17	00:03:08
1000	00:04:59	00:05:40
5000	00:26:30	00:29:05

Table 6.5: Evaluating download time between AWS API and AWS management console

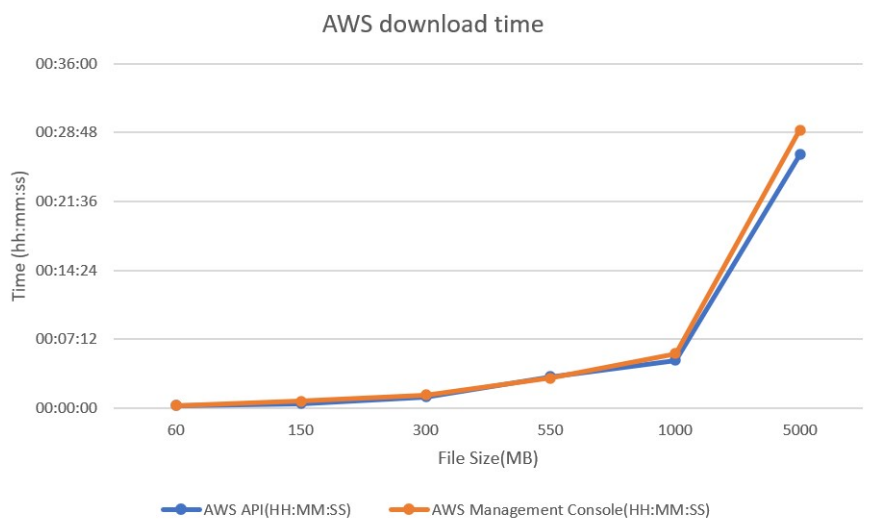


Figure 6.4: Download time between AWS API and AWS Management Console

6.4 Code Portability

When discussing code portability, the feasibility of employing the same application across diverse environments is considered. This involves ensuring that the code can function continuously across various platforms, enhancing its versatility and adaptability to different settings or systems. This characteristic is crucial for optimizing development processes and ensuring consistent performance across a range of deployment scenarios.

6.4.1 Orchestrator

To ensure the portability of the orchestrator, a Helm chart was employed. Within this chart, two pivotal YAML files were designed: ’values’ and ’deployment’, as depicted in Figure 6.5.

In the ’values’ file, several crucial properties are established to facilitate future adjustments:

1. **Image**: This file holds the latest Docker image built for the orchestrator.
2. **Repository**: This relates to the project application.
3. **Container Port**: This denotes the internal service port that receives incoming requests.
4. **Configuration**: It includes all environment variables.

As for the ’deployment’ file, it references the environmental variables specified in the ’values’ file. For instance, with regard to the orchestrator, it incorporates the Container Port and configuration variables. This interconnection ensures the continuous functioning of the orchestrator across diverse environments.

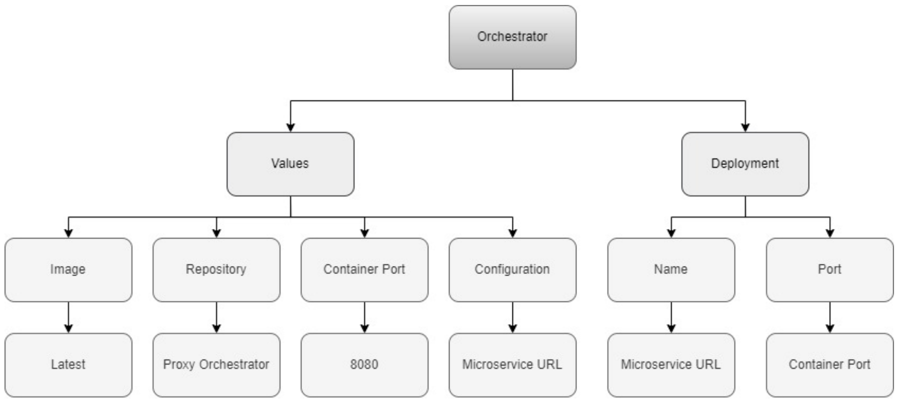


Figure 6.5: Orchestrator's Code Portability

6.5 Microservice

When it comes to the microservice, a comparable approach to that of the orchestrator was taken as show in figure 6.6. A Helm chart was employed, including the 'values' and 'deployment' YAML files. The distinction lies in the 'values' file, where the container port was modified to 8081, allowing it to process incoming requests.

In the 'deployment' file, the environmental variables now include the connection string, AWS region, and Azure name. This signifies that for any forthcoming adjustments, simply modifying the configuration in the deployment file is enough to adapt the application to the new setup, ensuring a consistent transition.

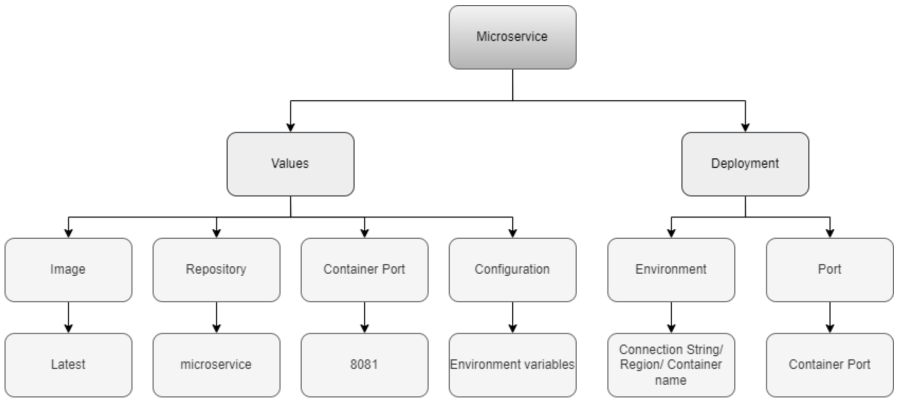


Figure 6.6: Microservice Code Portability

6.6 Evaluating the significance of how the solution performs with and without its existence.

Enabling continuous access to diverse cloud blob storage solutions from Kubernetes pods represents an important advancement in modern cloud computing infrastructure. This approach not only ensures cost efficiency but also improves scalability, a critical feature for contemporary applications built on microservices architecture. By adopting this methodology, organizations gain the invaluable capability of achieving true application portability across various cloud providers. Without this integration, applications can become contrained to specific cloud environments, limiting flexibility and impeding the agility required for dynamic, multi-cloud operations.

In practical terms, applying the proposed solution can lead to substantial time and resource savings as we can see on table 6.6. Traditionally, configuring individual connections to distinct cloud storage platforms can be a time-consuming task, often involving complex setups and custom configurations. With a unified access mechanism through Kubernetes, the complexity decreases significantly, simplifying the deployment process. Additionally, the ability to continuously transition between different cloud providers translates to potentially saving significant resources that would otherwise be allocated to reconfiguration and testing. In quantifiable terms, organizations could potentially reduce deployment and migration times by up to 30 percent and save resources equivalent to several person-hours, depending on the scale and complexity of the application ecosystem. This level of efficiency not only optimizes operational costs but also enhances the overall agility and adaptability of the organization in an evolving cloud landscape.

Aspect	Traditional Approach	Proposed Approach
Configuration Time	High	Low
Deployment Flexibility	Limited	High
Application Portability	Limited	Extensive
Scalability	Limited	High
Cost Efficiency	Moderate	High

Table 6.6: Comparison between the traditional and the proposed approach

From the context provided above:

Configuration Time

- **High**: Setting up individual connections to various cloud storage platforms in the traditional approach can be time-consuming due to the need for manual configuration and potentially complex authentication processes.

- **Low**: With the proposed approach using Kubernetes, setting up connections to different cloud storage platforms is efficient and requires less manual configuration, leading to faster setup times. Existing dedicated segments within the microservice can be utilized for accelerated API deployment.

Deployment flexibility

- **Limited**: The traditional approach may limit deployment options, making it more challenging to deploy applications in different cloud environments or adapt to changing infrastructure needs.
- **High**: The proposed approach provides greater flexibility, allowing for seamless deployment across various cloud providers and adapting to dynamic infrastructure requirements. Simply incorporate the access and secret keys as environmental variables within the helm chart, and proceed to configure the corresponding methods within the controller.

Application Portability

- **Limited**: Applications in the traditional approach may be tightly applied to a specific cloud provider's services, limiting their portability to other environments.
- **Extensive**: The proposed approach can run on different cloud providers with minimal modifications. Simply include the additional configuration in both the microservice and the proxy orchestrator.

Scalability

- **Limited**: The traditional approach may have limitations in scaling applications to meet increased demand, potentially leading to performance issues.
- **High**: The integrated approach provides better scalability options, allowing applications to easily scale to accommodate changing workloads.

Cost Efficiency

- **Moderate**: The traditional approach may have moderate cost efficiency, depending on the specific configurations and resource usage.
- **High**: The integrated approach is designed to optimize resource usage and reduce costs, making it more cost-efficient. You don't have to rebuild the entire microservice from scratch. Just incorporate the appropriate APIs, reducing the amount of person-hours needed for implementation.

7 Conclusion and future work

In the course of this thesis, we carefully designed a universal approach that contains the creation of both a microservice and a proxy orchestrator, designed for access to various cloud storage solutions, including AWS and Azure. The solution focuses around the integration of API configurations within the microservice. Concurrently, within the Helm chart, a key step involves the addition of the string connection environment variable. This dynamic process is crucial in ensuring the system remains in an uninterrupted state of adaptation to any evolving cloud environment. We also observed that the age of the Spring Boot framework does not necessarily imply a decrease in API performance. As demonstrated by our results, the performance of file transfers is both acceptable and satisfactory. The adoption of a microservice architecture for accessing both of our cloud storages significantly improves scalability. It allows for the incorporation of additional API configurations, enabling a broader range of functions to be performed. It also improved the flexibility, important in development teams that allow them to work on different components concurrently, enabling faster development and deployment cycles. This aspect is especially beneficial when integrating with dynamic cloud storage environments that may evolve over time. The microservice also improved the fault isolation by preventing issues in one component from affecting the entire system, enhancing overall system stability. The resource efficiency is other aspect to mention due to the fact that our microservice can be deployed on smaller and more specialized instances, reducing resource wastage. This is particularly advantageous when considering the pay-as-you-go model of many cloud providers, as it can lead to cost savings. Furthermore, implementing a microservice architecture when accessing cloud storage not only improves scalability and performance but also provides a foundation for flexibility, resilience, and cost-effectiveness in a dynamic cloud environment.

Looking ahead, it would be advantageous to contemplate the integration of additional cloud providers into the architecture. This could mean the inclusion of notable platforms like Google Cloud, Alibaba Cloud, IBM Cloud, Oracle Cloud, Red Hat Cloud, DigitalOcean Cloud, Rackspace, among others. Maximizing the comprehensive approach outlined in this thesis, the implementation of these additional providers should be relatively simple, granting favorable outcomes aligned with specific business requirements.

Moreover, exploring into an analysis of how the microservice's performance is impacted by adopting alternative Java frameworks, such as Quarkus or Micronaut, holds the potential to produce invaluable insights. As previously observed, these two Java frameworks have exhibited notable enhancements over time, particularly in terms of performance. It would not be surprising if, in the future, these frameworks surpass the transfer speeds achievable on the web portals of individual cloud providers.

Bibliography

[1] Łukasz Wyciślik, Łukasz Latusik and Anna Małgorzata Kamińska, “A comparative assessment of jvm frameworks to develop microservices.” [Online]. Available: https://www.mdpi.com/2076-3417/13/3/1343

[2] Piotr Plecinski, Nataliia Bokla, Tamara Klymkovych, Mykhailo Melnyk and Wojciech Zabierowski, “Comparison of representative microservices technologies in terms of performance for use for projects based on sensor networks,” 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/20/7759

[3] Shani du Plessis, Bruno Mendes, Noélia Correia, “A comparative study of microservices frameworks in iot deployments.” [Online]. Available: https://ieeexplore.ieee.org/document/9505049

[4] Songbin Liu†, Xiaomeng Huang , Haohuan Fu , Guangwen Yang†, “Understanding data characteristics and access patterns in a cloud storage system,” 2013. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6546109

[5] Francesco Marchioni, “Hands-on cloud-native applications with java and quarkus,” 2019. [Online]. Available: https://www.packtpub.com/product/hands-on-cloud-native-applications-with-java-and-quarkus/9781838821470

[6] Harlan McGhan, Mike O’Connor, “Picojava: A direct execution engine for java bytecode,” 1998. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=722273

[7] Microsoft, “What is a virtual machine (vm)?” Accessed on June 2023. [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine

[8] Dong-Heon Jung, Jong Kuk Park, Sung-Hwan Bae, Jaemok Lee, Soo-Mook Moon, “Efficient exception handling in java bytecode-to-c ahead-of-time compiler for smbedded systems,” 2006. [Online]. Available: https://dl.acm.org/doi/10.1145/1176887.1176915

[9] Christian Posta, “Microservices for java developers. in microservices for java developers pp. 6–7,” 2016. [Online]. Available: https://www.oreilly.com/content/microservices-for-java-developers/

[10] Luqman Saeed, “What is java ee? in: Introducing jakarta ee cdi. apress, berkeley, ca.” 2020. [Online]. Available: https://doi.org/10.1007/978-1-4842-5642-8_1

[11] IBM, "What is java spring boot?" Accessed on June 2023. [Online]. Available: https://www.ibm.com/topics/java-spring-boot

[12] Phillip Webb et al., "Spring boot reference documentation," 2023. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started

[13] Norman Walsh, "What is xml?" Accessed on June 2023. [Online]. Available: https://www.ce.unipr.it//people/bianchi/Teaching/IntelligenzaArtificiale/rdf_pl/XML-RDF/xmlguide1.html

[14] Cédric Champeau et al., "Graalvm," 2023. [Online]. Available: https://github.com/graalvm/native-build-tools/commits?author=dnestoro

[15] The Apache Software Foundation, "Spring boot maven plugin documentation," Accessed on June 2023. [Online]. Available: https://docs.spring.io/spring-boot/docs/3.1.0/maven-plugin/reference/pdf/spring-boot-maven-plugin-reference.pdf

[16] Andy Wilkinson, Scott Frederick, "Spring boot gradle plugin reference guide," Accessed on June 2023. [Online]. Available: https://docs.spring.io/spring-boot/docs/3.1.0/gradle-plugin/reference/pdf/spring-boot-gradle-plugin-reference.pdf

[17] Jason Hunter, "What's new in java servlet api 2.2?" Accessed on June 2023. [Online]. Available: https://www.infoworld.com/article/2076518/what-s-new-in-java-servlet-api-2-2-.html

[18] Spring, "Spring boot reference documentation," Accessed on June 2023. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using

[19] Micronaut, "Micronaut documentation," Accessed on June 2023. [Online]. Available: https://docs.micronaut.io/latest/guide/#introduction

[20] Nirmal singh and Zack Dawood, "Building microservices with micronaut," 2021. [Online]. Available: https://books.google.de/books?id=7qQ_EAAAQBAJ&printsec=frontcover#v=onepage&q&f=false

[21] Mr. Arabolu Chandra Sekhar, Dr. R. Praveen Sam, "A walk through of aws(amazon web services)," 2015. [Online]. Available: https://www.irjet.net/archives/V2/i3/Irjet-v2i332.pdf

[22] AWS Official Documentation, "What is amazon s3?" Accesed on September 2023. [Online]. Available: https://aws.amazon.com/pm/serv-s3/?trk=518a7bef-5b4f-4462-ad55-80e5c177f12b&sc_channel=ps&ef_id=CjwKCAjwkNOpBhBEEiwAb3MvvV_SVjJVa1hTrHEoXzSa0NT4OBPzq_Qp6cX7HUErePYUiqJIFY3QthoCddAQAvD_BwE:G:s&s_kwcid=AL!4422!3!645186213484!e!!g!!amazon%20s3!19579892800!143689755565

[23] AWS website official Documentation, “What is amazon ec2?” Accesed on September 2023. [Online]. Available: https://aws.amazon.com/pm/ec2/?trk=b59ef3d1-61fa-4eea-9a0b-96fbd6584e69&sc_channel=ps&ef_id=CjwKCAjwkNOpBhBEEiwAb3MvvRd4nobVjBH1WRIUxUIhCNiC26wgLwiFK0lME8NEUgUIZYHM3evtBhoCyxkQA
BwE:G:s&s_kwcid=AL!4422!3!645133569747!e!!g!!amazon%20ec2!19579892353!148838337561

[24] AWS Official Documentation, “Regions, availability zones, and local zones,” Accesed on September 2023. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html

[25] AWS website official Documentation, “Domains and domain state,” Accesed on September 2023. [Online]. Available: https://docs.aws.amazon.com/kms/latest/cryptographic-details/domains-and-domain-state.html

[26] AWS Official Documentation, “Amazon ebs volumes,” Accesed on October 2023. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volumes.html

[27] AWS website official Documentation, “Amazon ec2 security groups,” Accesed on October 2023. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-security-groups.html

[28] Prof Vaibhav A Gandhi, Dr C K Kumbharana, “Comparative study of amazon ec2 and microsoft azure cloud architecture,” 2018. [Online]. Available: https://www.researchgate.net/publication/327537294_Comparative_study_of_Amazon_EC2_and_Microsoft_Azure_cloud_architecture

[29] Microsoft Documentation, “Basic enterprise integration on azure,” Accessed on September 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/enterprise-integration/basic-enterprise-integration

[30] Azure Official Documentation, “What is a resource group,” Accesed on October 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal

[31] Azure website official Documentation, “Azure active directory is now microsoft entra id,” Accesed on October 2023. [Online]. Available: https://www.microsoft.com/en-us/security/business/identity-access/microsoft-entra-id

[32] Josef Spillner, “Quality assessment and improvement of helm charts for kubernetes-based cloud applications,” 2019. [Online]. Available: https://arxiv.org/pdf/1901.00644.pdf

[33] Matthias Graf, “Which java microservice framework should you choose in 2020?” 2020. [Online]. Available: https://betterprogramming.pub/which-java-microservice-framework-should-you-choose-in-2020-4e306a478e58

[34] Roman Kudryashov, “Review of microservices frameworks: A look at spring boot alternatives,” 2021. [Online]. Available: https://dzone.com/articles/not-only-spring-boot-a-review-of-alternatives

[35] Hartmut Schlosser, “Java trends: Top 10 frameworks in 2020,” 2020. [Online]. Available: https://devm.io/java/java-trends-top-10-frameworks-2020-168867

[36] Thorben Janssen, “Panache repository pattern,” Accessed on June 2023. [Online]. Available: https://thorben-janssen.com/panache-repository-pattern/

[37] Pratik Das, “Complete guide to spring resttemplate,” 2021. [Online]. Available: https://reflectoring.io/spring-resttemplate/

[38] Azure Official Documentation, “Azure portal documentation,” Accesed on September 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-portal/

[39] AWS Official Documentation, “What is the aws management console?” Accesed on September 2023. [Online]. Available: https://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/learn-whats-new.html

[40] Postman Official Documentation, “Postman api platform,” Accesed on September 2023. [Online]. Available: https://www.postman.com/