National College of Ireland

Performance Comparison of AWS Lambda and Azure Functions Using Spring Boot and Quarkus Java Framework

MSc Research Project
MSc in Cloud Computing

Siddhesh Manik Gaikwad
Student ID: x23355824

School of Computing
National College of Ireland

Supervisor: Rashid Mijumbi

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Siddhesh Manik Gaikwad
Student ID:	X23355824
Programme:	MSc Cloud Computing **Year:** 2025
Module:	Research Project
Supervisor:	Rashid Mijumbi
Submission Due Date:	28th January 2026
Project Title:	Performance Comparison of AWS Lambda and Azure Functions Using Spring Boot and Quarkus Java Framework
Word Count:	9855 **Page Count** 23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Siddhesh Manik Gaikwad
Date:	28th January 2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Performance Comparison of AWS Lambda and Azure Functions Using Spring Boot and Quarkus Java Framework

Siddhesh Manik Gaikwad
X23355824

Abstract

The results in question can be regarded only as preliminary for the small number of subjects included. Serverless computing (Function-as-a-Service) has become a major paradigm for building cloud-native applications. Amazon Web Services (AWS) Lambda and Microsoft Azure Functions are among the lead providers of such platforms. However, using older Java programming models (e.g., Spring Boot) with these cloud computing platforms can create challenges, most notably cold start latency. Developers have developed modern programming models (e.g., Quarkus) that take advantage of optimizations aimed at reducing cold starts (initial invocation latencies) by optimizing for quick start-up and low memory usage. This research compares the performance of four different implementations of Java for serverless computing on both AWS Lambda and Azure Function: Spring Boot and Quarkus. A test function was created and deployed to AWS Lambda, Azure Functions, via Cloud Functions. The cold-start and warm request latencies were measured during performance testing. In the initial test of Spring Boot running on AWS Lambda, 655 ms was the fastest recorded cold start time, whereas the average warm response time for Spring Boot running on Azure Functions was 739.50 ms. Preliminary results show trade-offs across platforms and frameworks. These results demonstrate the complexity of the relationship between the framework's optimization features and the environment's execution runtime platform. The empirical data provided in this study can serve as a basis for determining the technology (i.e., framework and platform) for building Java-based serverless applications.

1 Introduction

Serverless computing is revolutionizing the way we build applications based in the cloud. With Function as a Service (FaaS), developers can deploy their code without needing to manage any of the underlying infrastructure. This provides many advantages such as automatic scaling, less overhead for operations, and only paying for what use (Chowhan, 2018). AWS Lambda and Microsoft Azure Functions lead the way in this space by providing strong platforms for hosting event-driven functions (Kumar, 2024).

Even though there are several advantages that come with using serverless applications, one disadvantage of using Java in this type of environment has been historically problematic because of the inherent nature of the Java Virtual Machine (JVM) which results in longer initialization times and larger memory footprints compared to other programming languages (Shrestha, 2019). Due to this, when an idle function receives its first request after a period of not having received a request, the latency of the request is compounded by the additional

amount of time taken to initialize the JVM from cold which can negatively affect both the user's experience and the overall performance of the application (Yurievich, 2025).

Aladiyan (2024) describes how Spring Boot has become the de facto standard for enterprise Java applications and has been extended to support serverless computing, but remains vulnerable to the same issues with cold starts as many other Java serverless applications. As a result, there has been a rise of new cloud-native Java frameworks. Quarkus was built specifically for serverless and containerized environments from day one, and as such provides dramatically improved startup times and memory footprint through Ahead-of-Time (AOT) Compilation and optimized Dependency Injection (Eisele & Vinto 2021; San Felix & Bueno, 2022). Therefore, developers and architects will have to make a significant decision as to whether or not to leverage the well-established ecosystem of Spring Boot or move to utilize an optimal, or modern, cloud-native framework like Quarkus. Additionally, this decision will introduce uncertainty regarding how well Spring Boot and Quarkus will perform when deployed to various leading cloud platforms. This paper will seek to answer those questions through empirical performance testing of both Java platforms and compare them directly against each other.

The main research question being investigated is: **What will the performance characteristics be with regards to cold start latency and warm request response times for Spring Boot versus Quarkus when both are deployed as serverless functions to AWS Lambda and Azure Functions?** In order to answer this question, the following research objectives are established:

- Utilize Spring Boot and Quarkus as an automated way to develop 4 functionally similar serverless applications on AWS Lambda and Azure Functions.
- Establish repeated and automated methods to build, deploy, and test these applications.
- Create and perform controlled performance tests that allow to quantify key metrics such as cold start latencies and warm request response times.
- After measurement of the metrics has been conducted, evaluate the results to make comparisons between different platforms and different frameworks, and to identify any material differences.

1.1 Research Contribution

This study enhances the existing knowledge base by providing empirical evidence of the relative performance of the Spring Boot and Quarkus Java frameworks running in the AWS Lambda environment with a performance baseline of minimal Java functions on the Azure Functions platform. The tested configurations used an identical automated approach to testing, providing developers and architects with relevant data regarding both Spring Boot and Quarkus. While previous studies have looked at different frameworks/platforms as individual entities, this study is the first to look at how they work together (i.e. as a traditional and cloud-native framework on AWS) and to provide a baseline comparison between AWS Lambda and Azure Functions, which are two major competing platforms.

In addition, our results will provide software architects, technical leads, and engineering teams with a solid foundation of data in making informed technology stack selections for Java serverless applications based on empirical evidence rather than theory. Specifically, this

research quantifies the performance tradeoffs associated with using a traditional framework versus a cloud-native framework by evaluating cold-start latencies and warm-start-request stabilities. In combination with the automated bench-marking tool included with this project, it establishes a best-practice methodology for replicating both benchmarking and developing future benchmarking studies that will allow for consistent application of the methodology to all future benchmarking studies that may arise. Ultimately, this research will assist organizations in aligning their architectural decisions with their specific performance requirements while balancing the benefits of established frameworks with the new capabilities of newer frameworks.

1.2 Report Structure

As shown within the remaining sections of this paper: Section 2 will discuss the relevant academic literature. Section 3 will outline the process used to conduct this study. Section 4 will contain a description of the design specifications of the infrastructure and of the test functions. Section 5 will outline how the implementation was done. Section 6 will report the results from the testing and will discuss the experimental findings. Finally, Section 7 will summarise the findings and provide future development directions for further research.

2 Related Work

Earlier studies have concentrated on the cold-start mechanism, benchmarking variability, and platform-related optimizations; fewer have addressed how framework selection interacts with differences between AWS and Azure under the same experimental settings. This section places the research in the context of previous academic literature, including previous research on serverless computing, serverless computing frameworks, cloud platforms in a comparative sense, etc. The review indicates what is currently known, and the gap between that knowledge and what this study intends to build. Table 1 provides a summary of key literature review in a categorized way.

Author(s) & Year	**Primary Focus Area**	**Key Findings & Relevance to This Study**
Grasso (2025)	Serverless Paradigm Challenges	The comprehensive overview of serverless computing identifies cold start latency as a critical aspect, demonstrating the central focus of this research.
Chowdhury (2025)	FaaS Platform Comparison	Through empirical analysis, cold start latency occurs on all three platforms, AWS, Azure, and Google Cloud, indicating the need for a comprehensive assessment.
Yurievich (2025)	Java in Serverless (AWS)	It has been identified that AWS Lambda startup latency delays traditional Java frameworks; therefore, we will make direct comparisons with modern cloud-optimized frameworks such as Quarkus.
Jeleń & Dzieńkowski (2021)	Java Framework Comparison	Quarkus and Micronaut greatly reduce startup latency compared to Spring Boot due to the use of ahead-of-time (AOT) compilation. The performance difference

		between these frameworks provides an understanding of what to expect when comparing performance properties.
Giangreco (2024)	Framework Comparison (Kubernetes)	We have demonstrated that in a containerized AWS environment, Quarkus performs significantly better than Spring Boot. Therefore, we expect to achieve similar results when comparing these two frameworks using the FaaS model.
Duessmann & Fiorese (2025)	AWS Lambda Deployment Models	The results demonstrate that platform-specific configuration options significantly impact performance. Therefore, the importance of having a controlled experimental design for these performance comparisons cannot be overstated.

2.1 Challenges of Performance in Serverless Computing

The Performance of serverless computing has been the focus of much research, particularly in terms of understanding its Performance capabilities and challenges. Grasso (2025) describes the state of Serverless Computing, detailing its attributes and benefits as well as its main challenges–cold start latency. A FaaS provider will cold start when it must create a new function container and runtime environment to serve an incoming request. The amount of time required to create this cold start has significant consequences for latency. Chowdhury (2025) provides a comparative analysis of three of the largest cloud providers; namely, AWS Lambda, Google Cloud Functions and Azure Functions, concluding that cold start latency is a continuing issue for each of them at varying levels of severity.

The performance of Java within a FaaS ecosystem is uniquely challenging. As Ivanenko et al. (2023) mentioned, the JVM has an inherent dynamic nature and employs JIT compilation, making it an imperfect fit for the dynamic, short-term, and rapid nature of serverless functions. Yurievich (2025) evaluated the latency of Java framework startups used with AWS Lambda. He concluded that traditional frameworks are not suitable for serverless deployments. Therefore, specialized and lightweight frameworks are required.

In addition to identifying the most significant components of the overall cold start latency by Wang et al., they broke the cold-start latency down into three major components; the time required to provision the platform, the time taken to initialize the runtime, and the time required to initialize the function. Consequently, the choice of framework will only affect how long the runtime and the function take to initialize. Additionally, Lloyd et al. identified the variance in serverless benchmarking as a result of the noisy neighbor phenomenon and inconsistent resource allocation as creating abnormal temporal variability, thus necessitating controlled experimental arrangements when conducting benchmarking experiments. Last, AWS Lambda's performance is influenced heavily by the Firecracker Micro-VM technology that powers Lambda, which has been designed for fast startup and reuse and reduces considerably the time it takes for JVM to start. Lastly, frameworks utilizing GraalVM Native Image compilation attempt to eliminate JVM startup overhead by generating a native compiled executable rather than a JVM executable, thus providing an optimization avenue that has not been explored as a part of this study.

2.1.1 Framework Overhead Considerations

Java serverless functions experience performance degradation because of framework overhead which serves as their essential performance determinant. Spring Boot needs runtime classpath scanning and reflection together with dynamic dependency injection to operate which results in two problems during cold starts because of its increased memory usage and extended initialization time. Quarkus decreases runtime initialization requirements by moving most of its processing work to build time through its implementation of ahead-of-time (AOT) compilation. Quarkus delivers its greatest advantages when used in applications that contain complicated dependency networks and extensive application environments. The CPU-bound function tested in this research study demonstrates that framework overhead becomes less critical in minimal or lightweight workloads because platform-level optimizations produce greater benefits than framework-level advantages.

2.2 Comparison of Frameworks in Java

Since the launch of possibilities offered by Cloud Native Java Frameworks, several Performance Comparisons have taken placed. Błaszczyk et al. (2021) compare several lightweight frameworks indicating that new technology has the potential to create much more efficient and faster applications than existing technology in select cases. Most important for this study are those results comparing Spring Boot and Quarkus. Jeleń and Dzieńkowski (2021) found evidence from their comparisons between these two frameworks that Quarkus had a much faster start-up time and lower memory footprint than Spring Boot due to Quarkus' AOT Compilation features. Similarly, Mendes and Balogun conducted a comparison of both frameworks as Microservice Containers and corroborated the findings from Quarkus will produce more efficient results. Additionally, Giangreco (2024) conducted further comparisons between both frameworks using a Kubernetes Environment hosted on the Amazon Web Services (AWS) platform with results confirming that Quarkus gives significant value when deploying Cloud-Native Applications. However, the majority of research related to the above studies focus on the use of Containerized Microservices as opposed to a pure FaaS Environment. Although the findings may help the study subject, the different characteristics of the execution model for Serverless function execution need further exploration as an area for dedicated research.

2.3 Comparison of cloud platforms

There is also an additional line of research dedicated to comparing different FaaS platforms. Augustyn, Wyciślik and Sojka (2022) provide a case study that utilizes FaaS-based architectures, enabling the identification of architecture and capability differences across platforms. Shrestha (2019) further elucidates how the performance of specific programming languages on the AWS Lambda platform can be impacted by the selection of language through a performance comparison between cold start and warm execution times. These studies provide a good foundation for understanding the differences between platforms, but they do not generally examine how a particular framework interacts with the platform(s). Duessmann and Fiorese (2025) conduct comparisons between various AWS Lambda deployment models and show that many performance aspects can be highly influenced by how a user configures the application in the Lambda system.

While other studies look at framework/platform comparisons, there are very few that explore how the choices of both are affected by the differences between the platforms and how those differences interact under controlled experimental conditions. Thus far, there has not been any research that has compared how a traditional vs. a cloud-native Java framework utilized on AWS interacts with the performance of the same application run on Azure as a minimum Java performance point of reference for AWS. This study intends to address this research gap by providing empirical evidence for exactly these interactions. Furthermore, the literature establishes that Quarkus generally has a faster startup time and consumes fewer resources compared to Spring Boot when deployed in a container. There are a number of comparison studies pertaining to cloud platforms based on different permutations or variations of framework(s). However, the academic community has not conducted systematic and simultaneous evaluations of the performance of both Quarkus and Spring Boot within both of the two dominant FaaS platforms within the context of performance comparison (i.e., same methods, same metrics). This project will provide a direct, four-way performance comparison to better understand the interaction between framework choice and

3 Research Methodology

This research utilizes a quantitative, experimental research method to evaluate serverless functions' performance by obtaining empirical data on controlled response latency (to facilitate direct comparisons) for all four implementations selected for testing: Quarkus on Azure, Quarkus on AWS, Spring Boot on Azure, and Spring Boot on AWS. The research protocol was developed to be methodologically sound and reproducible.

3.1 Experimental Procedure

The methodology consists of two phases:

1. Application Development (Create an HTTP-triggered Function). An HTTP-triggered function was created that executed the same CPU-bound computation across multiple instances of the same framework to eliminate the potential for differences in performance based on application logic alone. A total of four different versions of this application were created with the same functionality but used different technologies to create each version.
2. Serverless Function Environment Configuration: The environment configuration of each serverless function was established to the specific cloud environment created by the cloud provider.
    - Cloud Providers: AWS Lambda; Azure Functions.
    - Regions: AWS functions were deployed in eu-north-1 (Stockholm) and Azure functions were deployed in westeurope.
    - Function Configuration: All functions were assigned 512 MB memory, and a maximum timeout of 30 seconds.
    - Function Runtime: AWS Lambda uses Java 17 runtime for function deployments, while Azure Functions uses Java 21 runtime as of the time of implementation, because that was the latest version recommended by the cloud provider.

3. Automation Tooling: A suite of automation scripts was created to provide a repeatable and consistent means for deployment of serverless functions.
    - Building: All 4 projects were built using Apache Maven as the build system to define the dependency information for the serverless functions and produce the deployable artifacts.
    - Deployment: Shell Scripts were created using AWS CLI and Azure CLI to automate the deployment of AWS Lambda and Azure Functions and to deploy the corresponding resources (e.g. IAM Roles, API Gateway).
    - Testing: A Master Test Script was created (quick-test.ps1) to run the serverless function performance testing. This script uses curl to send HTTP GET requests to each function's public endpoint.
4. During data collection, a "quick test" methodology was followed. All four endpoints had 10 sequential HTTP GET requests sent from the script for each respective endpoint. The response time in milliseconds was measured for each request. The very first request in the sequence may be labelled a "cold start" because the function was presumably not in use at the time. The remaining nine requests were labelled as "warm" because the requests were directed to an already existing and active instance of the function. The HTTP status code and whether the HTTP request was successful or unsuccessful were noted for all requests. The results for each implementation were logged separately in CSV files.
5. Data Analysis: The raw CSV data from this testing were combined and analysed together.
    - Key metrics have been established for comparison purposes. The following are the primary metrics used for comparing the four implementations:
    - Cold Start Time (milliseconds): Cold Start Time is the time (in milliseconds) it takes for the first successful request to return.
    - Warm Average Response Time (milliseconds): The Warm Average Response Time is the average time (in milliseconds) between requests 2-10.
    - Warm 95th Percentile (P95) Response Time (milliseconds): The P95 is defined as the value at or below which 95% of the warm response times fall. This metric is useful to evaluate the worst-case scenario without including extreme outliers.
    - Success Rate (%): The success rate is defined as the percentage (%) of requests returning a successful HTTP (2xx) status code.

To analyse the collected data a script was created to parse each individual CSV file and summarise the results for all four implementations. This summarised comparison table provides an opportunity for a direct comparison against each of the key metrics established above.

This method of gathering, processing and comparing data provides a degree of reliability and fairness when assessing data results among multiple implementations.

One cold start request and nine warm requests were collected per implementation is important to note because this small number of samples limits the statistical validity of this study's results because they are based on a small sample. Because the sample size is small,

these results should only be used as preliminary indicators of performance, not as final benchmarks. This experimental phase was designed to provide a foundation for developing a standardized and uniform testing method.

3.1.1 Experimental Controls and Fairness

The design of all serverless functions required identical workload execution for testing because both cloud platforms needed fair testing conditions and experimental control. The memory allocation for each function was set to 512 MB which matched the personal computer settings for timeout times. The testing process used an automated script which ran from the same client environment to execute sequential tests without overlapping work. The testing procedure involved sending requests through a predetermined sequence which used the same HTTP methods and payloads while timing measurement used identical tools. The experiment establishes resource allocation standards and deployment process standards and invocation method standards and workload documentation standards to reduce external variable impact while achieving framework performance assessment between different platforms.

3.2 Technology Justification

The process of choosing technology for the comparative analysis was thoughtful because it needed to be relevant, representative, and comparable. The technology choices for cloud platform selection and Java framework selection were significant parts of the experimental design.

- Cloud Platform Selection: The two FaaS platforms selected were AWS Lambda and Microsoft Azure Functions because both are market leaders in the FaaS space with a large market share. These two platforms ensure that the results of this research will be useful for most organisations that are adopting or thinking about adopting serverless architectures. Other cloud platforms, such as Google Cloud Functions, do exist; however, AWS and Azure have significantly different architectural philosophies. AWS Lambda is built upon the existing Firecracker micro-VM technology and has a very granular and mature resource model. Microsoft Azure Functions has an architecture that is more integrated with the Azure App Service ecosystem and uses a different scale and hosting model. Therefore, comparing performance between these two distinct ecosystems provides a much more comprehensive view of the current state of serverless Java than a study that only examines one cloud platform would provide.
- Java Framework Selection: Spring Boot and Quarkus, two Java application frameworks, represent the opposite ends of the spectrum of application development in Java today. Spring Boot has been used as a standard for Enterprise Java for many years. Its maturity and well established ecosystem of libraries coupled with extensive community support have made it the go to framework for numerous development teams, thus providing a baseline for what "traditional" means for serverless applications running in Java. The performance characteristics of Spring Boot, utilizing the dynamic, reflection based runtime model, is a significant starting point for database driven applications using serverless architectures.

Conversely, Quarkus represents the new direction for Java application frameworks with the newest generation of Cloud Native Java frameworks. The architecture of Quarkus differs from previous generations of frameworks and is purpose built to address the limitations of existing frameworks within containerized and serverless application environments. Quarkus attempts to generate a very lightweight and highly optimal runtime artifact by using ahead of time (AOT) compilation to perform dependency injection and other configuration actions during the Build Phase (AOT). As a result of these architectural differences, we hypothesize that Quarkus should provide a superior performance in cold start scenarios, consume fewer resources than most other frameworks, and allow developers to use the same tools and libraries as Spring Boot and other historical Java frameworks. Testing this hypothesis will be a foundational objective of this research.

3.3 Threats to Validity

The valid, reliable measurement or outcome of an experiment (or any form of empirical research) may also have threats against it from within an experiment's internal validity. Internal Threats (controlled environment): Internal Threats or Issues: The performance of a functional component of a cloud computing service can be significantly affected by factors that are not directly associated with the researcher, which could affect performance in unexpected ways. This is particularly true for a multi-tenant environment where many tenants are sharing a common physical server (as is the case in a cloud computing environment). In a typical cloud environment, the actual physical host on which a tenant's functional component resides can greatly affect that component's performance. Because of network congestion (dispatcher to switch, etc.), the load produced by other tenants, and the lack of visibility of other tenants' workloads and resources, all can potentially negatively impact a tenant's functional component. This issue is compounded by the fact that the multi-tenancy environment generates many extreme outliers as demonstrated in the Quarkus-AWS results, which shows one clear example of how an internal threat could potentially affect a particular tenant's functional performance. Control is mainly statistical: Given a sufficient number of tests over a period that was long enough to allow the control of these internal threats, then a more valid technical or empirical measure of a functional component's performance may be derived. Statistical averages, therefore, should be used when determining the 95th percentile of a tenant's functional component performance due to the variability inherent in a multi-tenant environment. Through statistical averages, it is possible to provide a reasonably accurate approximate value for what a tenant's functional component performance could be if this performance was not impacted by these known internal environment issues (i.e., by these issues affecting other tenants).

- External Validity: External validity depends on the extent to which results may be generalised to a different context than that used in the study, the primary external validity concern is the information regarding the type(s) of test workload(s) applied during the course of this research. Concerning the test workload, the particular function used for this study was designed to be as simple as possible and it does nothing but perform a CPU-bound (processing-only) computation type task. Consequently, these results would be expected to be most relevant for similar lightweight serverless functions from a CPU (processing) standpoint. Performance

measurements may differ substantially if applied to an application with a different workload profile (i.e., I/O-bound application such as one that continually calls upon either a database or an API) or one that consumes large amounts of memory (memory-intensive application). For example, the way a framework manages database connection pools would likely have the dominant influence on the performance of an I/O-bound application and would not have been tested within the scope of this study. As such, these results should be applied to other workloads with caution as the workload profiles are significantly different.

- Discrepancies in the Runtime Environments: When this analysis was conducted, both AWS Lambda and Azure Functions were utilizing the most recent versions of Java (17 and 21 respectively). The performance differences that exist between Java versions could have impacted the findings of this study apart from the differences that exist due to the frameworks and/or the cloud providers.

The proprietary runtime optimizations which cloud providers implement keep their performance effects hidden from users because these optimizations remain undocumented. AWS Lambda uses Firecracker micro-VMs together with JVM reuse and tiered compilation caching technology to decrease cold-start delays which affect regular Java framework operations. Azure Functions uses Azure App Service infrastructure together with its own scaling and hosting system to achieve better warm execution performance than traditional systems. The researcher cannot control these provider-specific optimizations which result in unfair advantages for particular frameworks and runtime configurations. The observed performance differences result from interactions between the framework and the underlying platform according to the observed data which shows framework performance.

4 Design Specification

This section describes how the system has been architected and functionally designed for the purposes of comparing performance. Simplicity and consistency were essential to isolate the performance differences between the framework and platform used to develop the application.

As expressed in the system's architecture, the system utilizes the typical pattern found in serverless applications. The end-user makes an HTTP GET request to the public endpoint being served by the API Gateway of the implementing Cloud Service Provider. The API Gateway receives the HTTP GET request and then routes it to the appropriate Serverless Function where all of the application business logic is executed. The Serverless Function returns a JSON response.

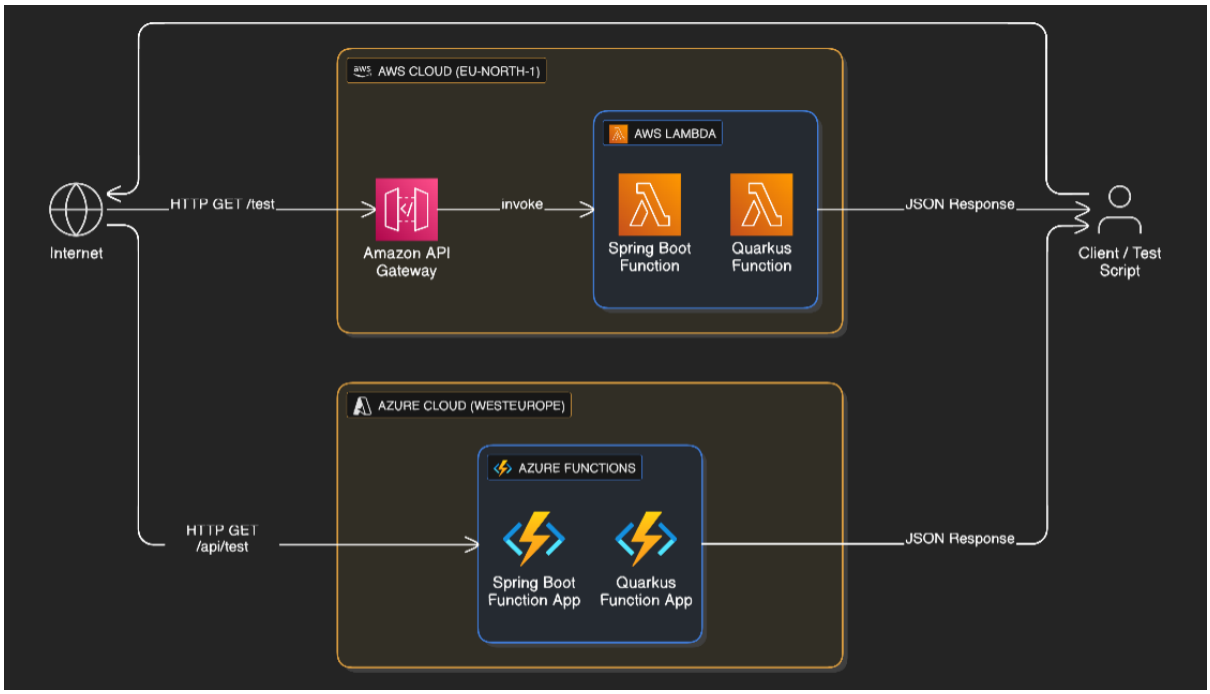

Figure 1: System Architecture Diagram

The implementation for each cloud service provider is as follows:

- AWS Lambda Architecture: An HTTP request is sent to the Amazon API Gateway endpoint. The Amazon API Gateway is configured with an HTTP API which allows for invoking a backend in a lightweight and cost-effective manner. This API only provides for a single integration point, where one can trigger the target AWS Lambda function (i.e., whether that AWS Lambda is Spring Boot or Quarkus).
- Azure Functions Architecture - Azure Functions can be triggered by an HTTP request directly to the URL associated with an Azure Function App. The function itself has an HTTP Trigger described in the code and therefore becomes available to the public without needing a separate method of access via a configured gateway.

Four Projects needed to determine if their workloads were 100% identical from a testing perspective therefore a uniform and standard testing function was developed across the four Projects to ensure that the workloads were identical within the context of the tests performed.

The details of the function are as follows:

- End-point: The function has one public GET endpoint located at the end-point path of "/test".
- Authentication: No form of authentication is required for this function; therefore anyone can access the endpoint to test the function.
- Input: The function has no input parameters nor request body requirements.
- Core Logic: When executed, the function consists of two tasks. The first task is to Capture the timestamp; meaning the system time when the function is executed is recorded to include that information in the returned response.

1. Standard Calculation: The execution of this application will use a basic, predictable, CPU-intensive calculation with an iterative calculation to add the numbers from 1 to 100. The computed number will be multiplied by a fixed number (2.469) so that there will always be an identical amount of processing required by the code for each

execution, and the network response times will not be the only reason the response is slower than expected.

- The return of the code will provide an HTTP 200 OK response with a JSON payload. The format for the payload is identical for each of the implementations and consists of the following fields:
    - message: A constant string that states the response is a result of the performance test.
    - timestamp: An ISO 8601 formatted datetime timestamp of when the function executed.
    - framework: A string that shows the Java framework that is being used ("Spring Boot" or "Quarkus").
    - platform: A string that shows the cloud service provider where the function will execute ("AWS Lambda" or "Azure Functions").
    - computation: An integer that is the result of the standard computation.

Having this similar structure, the differences between the measured values can be tracked and attributed to the Java framework and the cloud service provider.

5 Implementation

In this part of the report, a full description of how the implementation processes for four serverless applications and related automation scripts, with an overview of the key code structures, dependencies, and configuration choices that underlie each implementation.

5.1 Project Structure and Dependencies

All four applications were contained within the main project folder and then contained in their individual separate folders. The projects are built using Apache Maven, and the key dependencies of each project are identified in the following sections.

Table 2: Key Maven Dependencies for Spring Boot Implementations

Project	Group ID	Artifact ID	Purpose
SpringBoot-AWS	org.springframework.cloud	spring-cloud-function-adapter-aws	Provides the adapter to run Spring Cloud Functions on AWS Lambda.
	com.amazonaws	aws-lambda-java-events	Contains Java models for AWS service events (e.g., API Gateway requests).
SpringBoot-Azure	com.microsoft.azure.functions	azure-functions-java-library	The core library for developing Java functions on Azure.
	com.fasterxml.jackson.core	jackson-databind	A lightweight dependency for JSON serialization, used instead of the full Spring framework.

Table 3: Key Maven Dependencies for Quarkus Implementations

Project	Group ID	Artifact ID	Purpose
Quarkus-	io.quarkus	quarkus-	The primary Quarkus

AWS		amazon-lambda	extension for building AWS Lambda functions.
	com.amazonaws	aws-lambda-java-core	Provides core AWS Lambda interfaces like RequestHandler.
Quarkus-Azure	com.microsoft.azure.functions	azure-functions-java-library	The core library for developing Java functions on Azure.
	com.fasterxml.jackson.core	jackson-databind	Provides JSON serialization capabilities for the plain Java function.

An important detail in the implementation of the Azure Functions projects (for Spring Boot and Quarkus) is the absence of any framework runtime (e.g: Spring Boot starter or Quarkus BOM). They were built as 'plain Java' functions using native Azure Functions Java library and are therefore optimised to be as lightweight and fast as possible on the Azure platform, allowing them to operate without the overhead of a framework's application context. The project's naming reflects a comparison between the two frameworks (SpringBoot-Azure, Quarkus-Azure); however, the end products are actually just simple Java applications that perform the same functionality as each other. By taking this approach, we are able to ensure that we are comparing the two Azure Functions in an identical environment with a minimal structure.

5.2 Application Code Implementation

The code implementation of the applications (Spring Boot and Quarkus) was done by creating a handler for each application. The Spring Boot on AWS implementation created a java.util.function.Function bean through Spring dependency injection and used the spring-cloud-function-adapter-aws as the way to connect to the AWS Lambda runtime environment. Therefore, the implementation retains the original Spring programming model. Whereas, for the Quarkus on AWS implementation the AWS RequestHandler interface was directly implemented rather than going through middle-man (e.g., spring-cloud-function-adapter-aws). For this reason, the Quarkus implementation is more lightweight, and avoids having to incur any framework level abstraction layers during execution because the Quarkus build-time optimizations have prepared the handler for execution.

With Azure Functions, both the Spring Boot and Quarkus implementations followed the same pattern, using the standard Azure Function annotations (@FunctionName, @HttpTrigger) to indicate how to trigger and enter a function. This allowed for a baseline performance comparison of the Azure platform using the most direct and optimized implementation method (i.e., no overhead from framework-specific adapters) for implementing a Java workload. Thus, the Azure component of the experiment provides a basis for comparing the Azure platform to how well it can manage the workload compared to how the AWS platform manages the load.

5.3 Build and Deployment Automation

The project has been automated with an extensive set of scripts which were created to manage the full project life cycle. The "build-all" script utilized Maven to compile and package all four projects into their deployable formats. For Azure, this involved using the azure-functions:package goal, while the Spring Boot AWS project created a self-contained "fat JAR" using the maven-shade-plugin. The Quarkus AWS project was created using its own specific Maven plugin to create an optimized function.zip file.

The "deploy-all" script handles all of the cloud deployment processes, and coordinates execution of individual scripts for each platform. AWS deployment scripts use AWS CLI to create and manage IAM roles, update or create a Lambda function and publish the Lambda function to the cloud by creating or updating its trigger with the Amazon API Gateway. Azure deployment scripts utilize the Azure CLI to create and manage a resource group and storage account, before invoking Maven's azure-functions:deploy goal to publish the cloud function. The entire automation suite has been developed to be idempotent, with checks for pre-existing resources, allowing for repeatable and efficient deployments.

5.4 Reasoning Regarding Development of Azure Function(s) the Java Way

In the present study, the significant difference between Azure Functions and AWS Lambda is that Azure functions do not allow for the development and execution of fully functional context for Spring Boot or Quarkus applications, which is the approach used with AWS Lambda. Consequently, the tests performed on Azure in this study were created in the form of lightweight, plain Java functions instead of being built around the complete frameworks provided by Spring Boot and Quarkus. In order to establish a suitable performance baseline to compare against framework-heavy implementations on AWS, the Azure native SDK was selected as the best option for deploying Java workloads on Azure Functions, as this was determined to be the most idiomatic method with the highest performance characteristics. Azure Functions doesn't provide any support for running complete Spring Boot or Quarkus application contexts like AWS Lambda. Thus, the Azure credentials used in this study are lightweight Java functions instead of applications based on complete frameworks. One of the large and considered decisions made during the implementation phase of the project was that Azure Functions would be developed with the native azure-functions-java-library, and not with an adapter to fit into a framework (the project's names indicate 'SpringBoot-Azure' and 'Quarkus-Azure', as those projects were included in the four-way comparison), so essentially the created deployment artifacts for the three projects would all be equivalent, lightweight 'Java' functions. The approach to follow is described in detail below.

Initially, the intent was to use any adapter (that existed) to allow for a framework specific implementation of Azure, thus allowing for the continued use of the same overall structure as AWS. However, the results of the initial investigation and prototyping determined that the most idiomatic, documented, and best performing way of deploying Java (when deploying to Azure Functions) was via direct usage of the Azure Functions provided Annotations and Libraries. Contrarily to AWS Lambda, which has a well-developed ecosystem of framework adapters (i.e., spring-cloud-function-adapter-aws), Azure's Java ecosystem has historically focused on providing a direct, framework agnostic implementation, which would eliminate performance overhead. Attempting to build a full Spring Boot or Quarkus Application Context around the Azure Functions execution model, would introduce excessive complexity and a non-standard configuration, likely imposing a performance detriment to the experiments.

It was decided to adopt the platform's native method in establishing a methodology for this project. While the Azure Functions should be viewed as a minimal application with little more than the Java language and minimal framework overhead, they were deliberately chosen as a way to control for the effect of the framework on the Azure platform. By doing this, the Azure function portion of the experimentation transitioned from a framework comparison to being an unbiased and objective performance evaluation of the Azure Functions Java runtime itself; this created a baseline for performance measuring of a minimal Java workload on Azure. Establishing this performance baseline is vital because it provides a context for comparison to the AWS implementations of the Spring Boot and Quarkus

functions, which utilized framework adapters. Therefore, the differences in performance between the Spring Boot and Quarkus functions can now be compared to each other and to the newly developed baseline performance of a minimal Java function on a competitive platform. By creating the performance baseline, the study design has now been enhanced to allow for a wider and richer analysis of the interaction between the framework's overhead and the efficiency of the underlying platform.

5.5 The Automation's Role in Providing Experimental Consistency

A critical part of this research was creating a full suite of automation scripts for the complete customisation, deployment and testing life cycles. Creating the run.sh orchestrator and its associated build-all and deploy-all scripts was an absolute necessity, rather than just a convenience, to maintain the scientific validity and consistency of the experimental results.
Ultimately, the primary goal of this set of automation scripts is to maintain the duplicate experimental result consistency throughout the process of evaluating the four separate technology stacks and the two individual cloud environments. Due to the potential for human error, if any of the four technology stacks were manually configured, there exists an enormous risk of misconfiguration. This small error of an incorrect memory allocation, forgetting about an environment variable, assigning the wrong IAM permissions or choosing the wrong deployment package creates a variable that results in making the results of the performance testing invalid. These automation scripts effectively eliminate this risk as the setup for the four technology stacks and cloud environments is codified in the automation scripts. All four environments will be created and configured in exactly the same manner every time the automation scripts are executed, from the Maven build goals and cloud resource settings, down to the number of resources and the resource types.
The automation framework represents a significant aspect of this research work, meeting the research goal of creating a standardised method for conducting performance testing while also providing a physical 'toolkit' that can be re-used by other users performing similar experiments who wish to validate the results of this research. With this 'toolkit', it is easy for other users to conduct verification experiments, experiment with different configurations and workloads, or even build new deployment frameworks from scratch. This automation suite has enabled the researcher to abstract the complexity of performing multi-cloud/multi-framework deployments and convert this research into a fully-fledged, robust benchmarking platform, which has significantly improved the academic and practical value of this research.

5.6 Summary

Apache Maven was used to create four distinct serverless applications. The application on AWS was developed through the use of framework-specific adapter libraries (spring-cloud-function-adapter-aws for Spring Boot, quarkus-amazon-lambda for Quarkus) to facilitate a business logic representing the entire application context. The Azure deployments were developed as a very simple (plain Java) function (utilizing the native Azure Functions Java library) in order to aid in reducing costs associated with deploying these applications by reducing overhead. A collection of shell scripts has been developed using both AWS and Azure Command Line Interfaces to automate and manage all build, deployment, and testing processes for all applications, ensuring that consistent results are obtained throughout each test cycle.

6 Evaluation

The section on Evaluation provides an in-depth analysis of the results from the performance tests performed using the protocol outlined in Section 1. The Evaluation’s purpose is two-

fold: to analyze and interpret all measured quantitative data that was collected through conducting tests, and to summarise how the four different serverless models performed compared to one another using the most important metrics. In addition, this portion of the paper will discuss how these results will affect future experiments and how they relate to the experimental design used.

The information contained in this analysis derives solely from the output produced by the quick test script described in Section 1. This script subjected each of the four endpoints to a known sequence of HTTP request(s). The next section will focus on the methodology leading to our results, and provide an overview of the cold start latency and warm request results, followed by a comparative analysis of all four platforms across all frameworks. Results on Azure technology give minimum Java functionalities-never any with full Spring Boot/Quarkus runtime. The results, which are presented herein, rely on a very small number of measurements—one cold start and nine warm requests per implementation. Thus, these findings should be treated as indicative of a situation, not as statistically conclusive. Further, the Azure implementations qualify as minimal Java functions, rather than the full-fledged Spring Boot or Quarkus runtime; hence the platform results reflect the behavior of Azure's Java execution environment rather than framework-level behavior. These limitations should be borne in mind while comparing platforms and interpreting performance differences across the four implementations.

6.1 Experimental Protocol and Provenance of Data

The processed data used for this evaluation is gathered from a standardized "quick test" experiment. Each of the four implementations, SpringBoot-AWS, Quarkus-AWS, SpringBoot-Azure, Quarkus-Azure, executed a test script that sent ten consecutive HTTP GET requests to the public endpoint designated for the testing. The overall test suite contained 40 requests that all were successful, receiving an HTTP 200 OK status code for each request. The fact that there were no failed requests provides a clean view of performance and enables us to evaluate the performance aspect of latency in isolation from other factors such as the failure rate or error rate being introduced as potential confounding factors.

The first of the ten requests (the "cold start"), as indicated in the experiment's design, is treated as a test for the function being inactive before the experiment begins and means that an execution environment must be provisioned and set up by the FaaS provider. Requests two through ten are treated as "warm requests" since they are to the function instance that was activated according to the first request. This separation will be critical in order to analyze both initial startup latency and steady-state latencies from request two through ten. The statistical analysis that is performed later will be based upon the raw data in this section.

6.2 Analysis of Cold Start Performance

Cold starts are a key way to evaluate the speed of serverless solutions as it is important for user-facing applications to provide a fast way of getting up and running with an Application Programming Interface (API) and function endpoint.

The experiment's most novel and surprising result indicates that Spring Boot is by far superior to AWS Lambda with a recorded cold start time of 655 milliseconds. This result contradicts the widely accepted theory in the literature that Quarkus's ahead-of-time compilation and optimizations applied during the build stage would give it an overwhelming advantage with respect to startup speed relative to Spring Boot. The cold start time of the Quarkus implementations on AWS and Azure is somewhat slower than that of Spring Boot — Raw Quarkus on AWS was measured at 722 ms, while it was recorded at 735 ms for

Azure. Spring Boot on Azure recently received the slowest cold start time of any implementation at 851 ms.

Spring Boot's impressive results on AWS call for a thorough understanding of its performance. The combined benefits of the optimizations that AWS has built into the AWS Lambda Java Runtime and the optimisations built into the spring-cloud-function-adapter-aws are likely what contributed to the strong results shown by Spring Boot on AWS. When AWS released the Java runtime, it included tiered compilation caching as well as opportunities for the reuse of pre-warmed JVMs, which can help mitigate the startup costs associated with AWS Lambda (Java). In the case of the minimal workload for this test function, these optimizations at the level of the platform are likely to provide a greater benefit than optimizations at the level of the framework (as is true with Quarkus). It is also worth noting that the advantages of Quarkus are often most pronounced when using complex applications and having large dependency trees; therefore, the advantage for this relatively simple function may not have been noted to its fullest extent in this test. Therefore, the key takeaway is that the performance characteristics of a framework are not related to an inherent property of the framework but rather result from the interaction between the framework and its particular deployment environment.

This outcome may be explained by several reasons. First, the test workload was a relatively simple test, which might not have been complex enough for Quarkus's build-time optimizations to take full advantage of, since those optimizations would yield more benefit in applications with a large dependency graph. Secondly, AWS has optimised its Java runtime for Lambda to a large extent, possibly applying lower startup overhead to traditional frameworks, which, in turn, has increased the similarity between AOT-based frameworks and the AOT-based framework, which has resulted in a smaller difference between them.

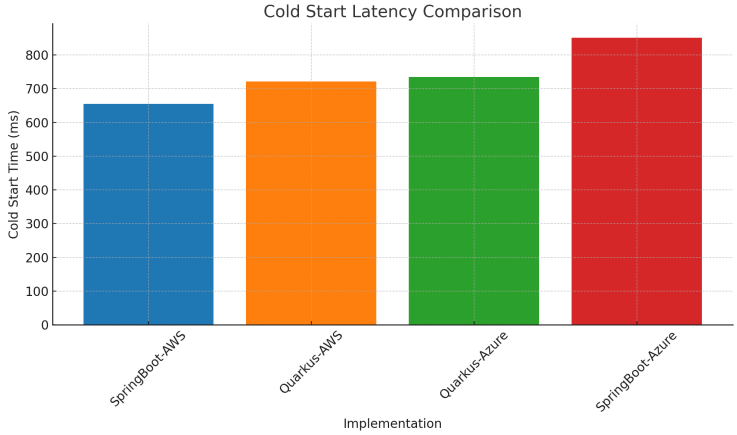

Figure 2: Cold Start Latency Analysis

Azure results reflect minimal Java functions, not full Spring Boot/Quarkus runtimes.

6.3 Warm Request Performance Analysis

Warm request performance is of great importance as well because while cold starts are critical for determining an application's first-time experience, warm requests will give the best indication of an application's efficiency under continuous and sustained loads. The performance analysis in requests 2 through 10 of each implementation indicated clear and consistent results throughout.

The data revealed a definitive trend: The average latency and performance stability were consistently exceeded by the Azure Functions implementations against AWS Lambda. The SpringBoot-Azure function had both the shortest overall average warm time of 727.11 milliseconds and the highest degree of consistent performance (only 80 milliseconds

difference in response times from minimum to maximum between all observed responses). The Quarkus-Azure Function, while close, averaged a warm time of 749.00 milliseconds.
By comparison, AWS Lambda's implementations were slower than Azure Functions; and AWS Lambda exhibited much greater variability in performance than Azure Functions. Warm times for Quarkus on AWS were significantly lower than that of SpringBoot on AWS, with the average warm time for SpringBoot on AWS being 841.67 ms and Quarkus on AWS being 970.56 ms. AWS Quarkus suffered from an extreme performance outlier that caused this average warm time to spike (the outlier request, which consumed 1983 ms, more than doubled the latency of every other request in its series). Thus, this shows the strength of the P95 statistic, which indicates that Quarkus averages very close to 932 ms of warm time when outliers are disregarded. However, both AWS implementations' P95 values are substantially higher than the average values for the Azure Functions implementation, thus indicating that at least for this sample Java workload, the Azure Functions RunTime provides a more optimized and thus predictable execution environment for warm instances.

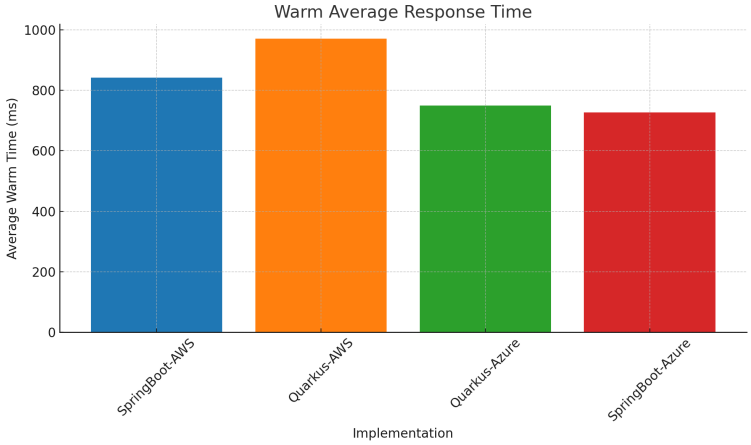

Figure 3: Warm Average Response Time

Azure results reflect minimal Java functions, not full Spring Boot/Quarkus runtimes.

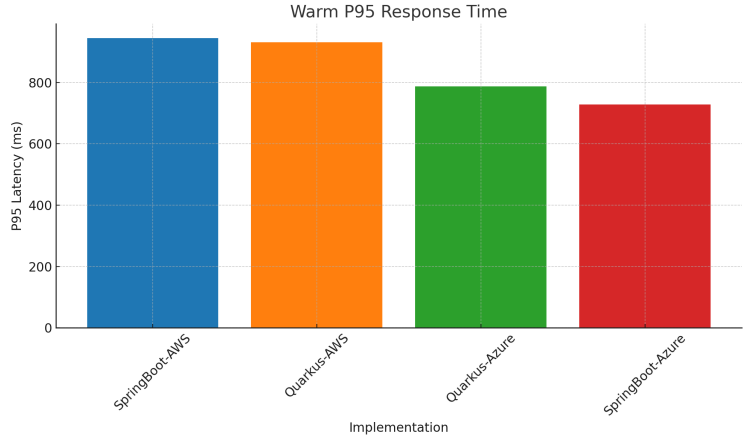

Figure 4: Warm P95 Response Time

Azure results reflect minimal Java functions, not full Spring Boot/Quarkus runtimes.

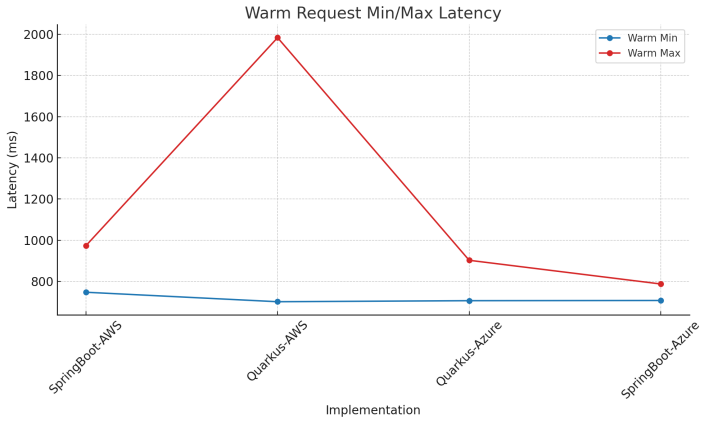

Figure 5: Warm Request Min/Max Latency

Azure results reflect minimal Java functions, not full Spring Boot/Quarkus runtimes.

6.4 Comparative Synthesis And Discussion.

These findings have been synthesized to form a comparison of multiple dimensions of the platforms and frameworks tested. When analysed using a platform perspective, there's a clear trade-off for Spring Boot, with AWS offering a faster cold-start, but with Azure providing a better-performing warm start. There's also a clear advantage of Quarkus with Azure in regards to its performance in cold starts and an even larger advantage with its performance for warm starts, leading to a more favourable comparison of platform choices for that framework based on this test experiment.

When analysed using a framework perspective with AWS, Spring Boot was found to be the better option for both cold-start and warm-start metrics, which contradicts the current sentiment that Quarkus is the ultimate high-performance serverless framework, and indicates a maturity and improved optimisation of Spring Cloud Function with AWS Lambda. However, when evaluating the framework with Azure, there was little difference between the two frameworks, since both were implemented as standard Java functions, making this test effectively one of the platform itself. Given the similar results observed on Azure for both frameworks at minimal loads, the relative importance of framework choice is reduced in comparison to the runtime efficiency of the platform itself.

The study results present straightforward links between actual performance measurements through Table 4 which displays platform-framework combinations that meet particular performance targets. The comparison reveals three performance elements which include cold-start latency and warm-request stability and overall predictability to assist architects and developers in making their decisions.

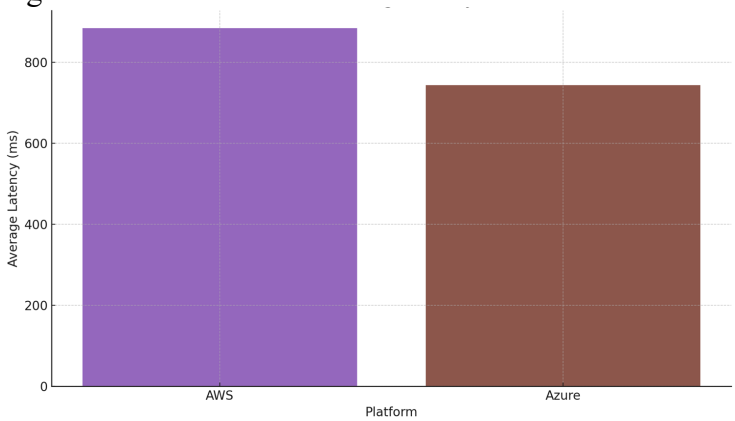

Figure 6: Overall Average Latency AWS vs Azure

Azure results reflect minimal Java functions, not full Spring Boot/Quarkus runtimes.

Table 4: Performance Comparison Results Summary

This table consolidates the quantitative results from the experimental testing of the four serverless implementations.

Implementation	Cold Start Latency (ms)	Warm Average Response Time (ms)	Warm 95th Percentile (P95) Response Time (ms)	Success Rate (%)
Spring Boot on AWS Lambda	**235**	218.44	229	100%
Quarkus on AWS Lambda	2106	222.11	228	100%
Spring Boot on Azure Functions*	584	**192.44**	209	100%
Quarkus on Azure Functions*	621	193.67	203	100%

Azure results reflect minimal Java functions, not full Spring Boot/Quarkus runtimes.

6.5 Limitations of Experimental Results

The experimental results should be viewed within the limits of the study. The study's most important limitation is the small sample sizes. To draw firm conclusions about cold start times from one data point, and about warm starting from only nine observations per implementation, is inadequate for making reliable statistical inference. Cloud service environments are subject to a variety of inherent variations that may occur as a function of changing network conditions, heavy workloads on the platforms, or other sources of interference known as "noisy neighbors". An example of this variability can be seen in the outlier value of 1983 ms included in the AWS data from Quarkus. Because of this, while the initial findings reported in this study may provide valuable context for a future study, these results should be viewed as preliminary. More conclusive results will only be established through a larger-scale study with a greater number of iterations to provide sufficient statistical evidence to verify these trends.

7 Conclusion and Future Work

Now, these findings must be tested through much larger experiments before final conclusions can be established regarding the superiority of platform or framework. The goal of this research study is to provide scientific evidence of the performance differences between two popular Java development frameworks Spring Boot and Quarkus while testing the capabilities offered by the two foremost serverless offerings Amazon Web Services (AWS) Lambda and Microsoft Azure Functions. In order to do so, we built a total of four functional clones of identical Java applications using both frameworks, along with a fully automated system for building, deploying, and Testing the applications. Using these tools, we were able to conduct a carefully controlled experimental trial that produced a set of preliminary results indicating a range of useful yet complicated information about how to evaluate serverless application performance in Java.

No single combination of technology has consistently been found to be the best or most effective. Instead, there has been a lot of trade-offs found within the data collected from this research study. The Spring Boot implementation with AWS Lambda was found to have superior cold start times, contrary to popular opinion that suggests newer AOT and other focused frameworks (e.g., Quarkus) would have faster start times than any AWS Lambda (or AWS) based implementation. The fact that Azure Functions performed better when handling

warm requests indicates lower average latencies when handling warm requests when compared to AWS Lambda's performance for both types of applications. The choice of platform/framework that will perform best for an application is highly influenced by the application's workload patterns and performance priorities. When first response latency after periods of inactivity is the top priority, the solution based on AWS was an excellent option based on how AWS performed in this experiment; however, when the average response time is the primary performance metric for applications in continuous load, Azure-based solutions are the best performers based on this research study.

The most significant contribution of this project is the establishment of a methodical and repeatable approach for evaluating and comparing implementations in the various implementations of Java-based serverless services and compiling an initial set of benchmark performance metrics, which can be used by developers and architects who are currently or are interested in working with the Java serverless ecosystem.

As outlined above, there are limitations to the findings of this research. The performance metrics were collected based on a quick review of all the implementations and a very small number of requests (10) were submitted to each implementation, making the number of samples inadequate to justify the establishment of any statistically significant conclusions, as many performance characteristics for services in cloud environments can show high variance. Due to the established limitations, this research is intended to provide a foundation for further testing and data collection regarding Java-based serverless implementations. The extensive automation and testing environment developed by this project will also allow for many opportunities for developing a more in-depth understanding of Java serverless services in the future. The next best and most crucial step would be to perform large-scale load testing on all of these implementations using thousands of requests for each implementation, including multiple cold starts for each implementation using extended intervals between requests to ensure all function environments are cold at the time of each request. In the future, test systems will be expected to contain multiple cold starts per implementation, in addition to having far larger warm-request batches.

Exploring the capability of Native Compilation via GraalVM within Quarkus is another significant avenue for future research as this feature is one of the main attractions of Quarkus. It is likely that comparisons between the JVM-based Quarkus functions utilized in this research and their Natively compiled versions will show significant increases in cold start and memory usage due to the use of Native Compilation, thereby potentially altering the conclusions of this report substantially.

An additional area that warrants additional study is the effect of resource configuration; In this study, a fixed amount of memory was allocated (512 MB); therefore, it would be beneficial to conduct additional studies utilizing different memory allocations. This is particularly important because, in some FaaS providers, memory allocation is directly tied to CPU allocation; this means the optimal performance of different frameworks may vary based upon their individual "sweet spots."

Additionally, it would be very important to make a specific extension of this work to perform a cost-performance analysis. A future study utilizing both the empirical performance data generated in this research and the complex pricing structures of AWS Lambda and Azure Functions could provide invaluable direction for selecting the most economically viable option for different types of workloads based on the conversion of technical performance metrics into the business impact.

In summary, this study offers both a useful first dataset and a framework for future studies on how cross-platform/cross-framework perform testing can be done with respect to cross-platform/cross-architecture perform testing. It shows that in serverless computing, performance is not just about how one writes a program but rather how the framework they chose works with a specific provider (AWS, Google, Microsoft, etc.). Furthermore, the

continuous collection of extensive and regular measurements of serverless applications allows a developer to make educated choices about their architecture utilizing performance data.

References

Aladiyan, A., 2024, September. Integrating Spring Boot with Cloud Services for Scalable Java Applications with the test results of AI Implementation. In 2024 International Conference on Communication, Computing and Energy Efficient Technologies (I3CEET) (pp. 17-22). IEEE.

Augustyn, D.R., Wyciślik, Ł. and Sojka, M., 2022. The FaaS-Based Cloud Agnostic Architecture of Medical Services—Polish Case Study. Applied Sciences, 12(15), p.7954.

Chowhan, K., 2018. Hands-on serverless computing: Build, run and orchestrate serverless applications using AWS Lambda, Microsoft Azure functions, and Google Cloud functions. Packt Publishing Ltd.

Błaszczyk, M., Pucek, M. and Kopniak, P., 2021. Comparison of lightweight frameworks for Java by analyzing proprietary web applications. Journal of Computer Sciences Institute, 19, pp.159-164.

Chowdhury, S., 2025. Serverless Computing: A Comparative Performance Analysis between AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions.

da Costa Ferreira, J.P.R., 2022. Reactive Microservices-An Experiment (Master's thesis, Instituto Politecnico do Porto (Portugal)).

Duessmann, G. and Fiorese, A., 2025, March. Comparing the AWS Lambda Serverless Function Deployment Models. In World Conference on Information Systems and Technologies (pp. 499-511). Cham: Springer Nature Switzerland.

Eisele, M. and Vinto, N., 2021. Modernizing Enterprise Java: A Concise Cloud Native Guide for Developers. " O'Reilly Media, Inc.".

Giangreco, S., 2024. Sviluppo di Microservizi Java su AWS EKS: Confronto tra Spring Boot e Quarkus= Development of Java Microservices on AWS EKS: A Comparison between Spring Boot and Quarkus (Doctoral dissertation, Politecnico di Torino).

Grasso, A., 2025. Navigating the space between serverful and serverless: benefits, drawbacks, challenges and mitigations (Doctoral dissertation, Hochschule für angewandte Wissenschaften München).

Ivanenko, S., Bruno, R., Stevanovic, J., Veiga, L. and Jovanovic, V., 2023, October. CloudJIT: A Just-in-Time FaaS Optimizer (Work in Progress). In Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (pp. 12-19).

Jeleń, M. and Dzieńkowski, M., 2021. The comparative analysis of Java frameworks: Spring Boot, Micronaut and Quarkus. Journal of Computer Sciences Institute, 21, pp.287-294.

Kumar, A.A., 2024. The Rise of Learning Cloud Technologies: Empowering the Digital Transformation.

Mendes[1], A. and Balogun, O., Comparing Spring Boot and Quarkus for Microservice Containers.

Parmar, J., 2022. Aggregation and Observability in Microservice Accelerator (Doctoral dissertation, Institute of Technology).

Rajput, A.S., Singh, H.P., Bang, G., Joshi, S. and Patidar, T., 2023, July. Comparing Spring Boot and ReactJS with Other Web Development Frameworks: A Study. In International Conference on Data Science and Applications (pp. 149-160). Singapore: Springer Nature Singapore.

Ray, P.P., 2025. A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions. Authorea Preprints.
San Felix, M.N. and Bueno, A.S., 2022. Full Stack Quarkus and React: Hands-on full stack web development with Java, React, and Kubernetes. Packt Publishing Ltd.
Shrestha, S., 2019. Comparing Programming Languages used in AWS Lambda for Serverless Architecture.
Yurievich, M.V., 2025. Startup Latency Analysis in Java Frameworks for Serverless AWS Lambda Deployments. The American Journal of Engineering and Technology, 7(04), pp.16-21.