International journal of IoT (ISSN: 2692-5184)

Volume 05, Issue 01, 2025, pages 126-145
Published Date: - 07-06-2025
Doi: -https://doi.org/10.55640/ijiot-05-01-07

Serverless Java: Cold Start Mitigation in Cloud Run/Spring Boot

Sandeep Reddy Gundla
Lead Software Engineer, MACYS Inc, GA, USA

ABSTRACT

This paper focuses on the cold start latency problem in serverless Java applications, seen mainly when deploying with Google Cloud Run and Spring Boot. Auto-scaling, reduced infrastructure use, and cost savings are why serverless computing is becoming more popular. Still, Java applications generally face delays when starting up, referred to as cold starts, because the JVM must boot up, and Spring Boot comprises complex settings. Because of this latency, users' experience can become very poor, especially when using real-time APIs, chatbots, and e-commerce systems. The study explains the processes behind cold starts, their impact, and several ways to address them. Cloud Run's usage of containers and Spring Boot being a good choice for creating microservices is covered, as well as their difficulties in initializing quickly. Optimizations include adjusting container images, native compilation using GraalVM, setting "minimum instances" and concurrency in Cloud Run, using lazy initialization in Spring Boot, and routinely pinging during warm-up. These approaches have been observed to improve latency from several seconds to almost nothing whenever they are used. This research uses case analysis, benchmarking, and performance monitoring to examine the effectiveness of mitigation strategies. The paper concludes that although Java encounters some cold start difficulties, recent advancements in tools such as GraalVM, virtual machine design, and better cloud support are promising. If Java is configured correctly and sufficient time is spent planning, it can be an excellent fit for serverless environments on Google Cloud Run.

KEYWORDS

Cold Start Latency, Serverless Java, Google Cloud Run, Spring Boot, GraalVM, Native Image, Compilation, Docker Optimization, JVM Startup Performance

1. INTRODUCTION

Serverless architecture has become popular in application development because it allows developers to create and deliver applications without managing the infrastructure. In a serverless model, the cloud provider sets up and operates the resources needed for an application to run independently. This approach offers companies lower costs, the ability to handle more users, and faster market release, as developers focus solely on coding. There is no need to worry about infrastructure management because the cloud handles it. Automatic scaling with serverless computing results in more efficient and affordable cloud network use than traditional server-based models. As a result, companies that aim to streamline operations and accelerate development often choose serverless architecture. Developers can run applications built using containers on Google Cloud Run without the need for server management. Since infrastructure does not need to be managed, Cloud Run is a simple solution for running stateless web services and APIs. Based on Kubernetes and Istio, Cloud Run supports any language or framework that can run in a container. For Java developers, one of the first considerations when using Cloud Run is working with Spring Boot. Spring Boot enables developers to easily create microservices for production in Java applications.

As a result, developers can build powerful and flexible applications with minimal setup required on Cloud Run. Serverless computing has its own issues to solve, one of which is the problem of cold start latency. When something happens for the first time or after an absence, the serverless function or container is cold-started, and the platform has to initialize it before continuing. Because of this initialization process, the application's performance can decrease, and its users encounter sluggishness. Many developers in serverless environments like Google Cloud Run have to solve the problem of cold starts in order for their software to work promptly. Speeding up the initial setup is important for Java software on the cloud, as it needs to perform well and scale. Compared to Node.js or Python, Java apps with Spring Boot can use more system resources and take more time to prepare. Therefore, working on starting Java apps quickly has become even more important for developers. A slow initial response from serverless Java applications for users, caused by cold starts, can create a bad user experience, higher exits by users, and a general loss in performance. Because of this, knowing how cold starts happen and using good strategies to lessen their effect is crucial for highly efficient and quick Java applications. This paper will examine how cold start latency affects serverless applications on Google Cloud Run and Spring Boot. This guide focuses on the causes of cold starts, their effect on performance, and useful methods for reducing the time needed for cold starts in Java-based serverless applications. Responding to this problem helps developers keep their serverless apps reliable, performant, and cost-effective.

2. Understanding Serverless Architecture

Serverless computing has made cloud usage more accessible for developers, allowing them to focus on coding while leaving server management to the service. The core concept of serverless computing is to manage infrastructure for developers, enabling applications to scale as needed. Serverless essentially means there is no need to worry about servers, as the cloud provider handles them. This approach fosters greater operational reliability and simplifies application management.

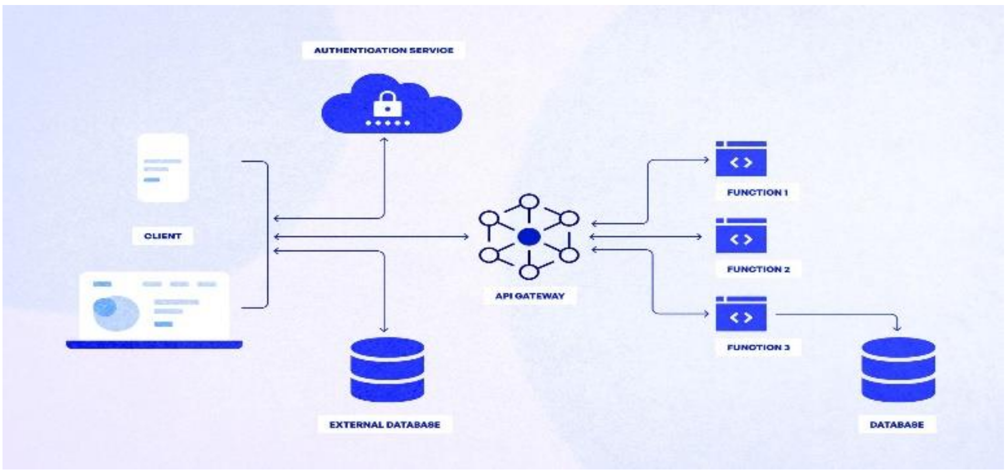


Figure 1: Serverless Architecture: A Comprehensive Guide

2.1 Definition and Principles of Serverless Computing

Using serverless computing, cloud providers set up, maintain, and run the infrastructure needed for the code. It runs using Function-as-a-Service (FaaS), meaning developers create individual functions launched when a particular event occurs. With this model, resources are distributed and changed automatically, helping applications cope easily with different traffic levels. Serverless computing acts on external events, so functions execute when HTTP requests come in, a database is updated, or a file is uploaded. Because of this architecture, developers do not have to worry about maintaining or scaling servers (Chavan, 2024).

2.2 Key Features of Serverless Platforms

What sets serverless servers apart from traditional servers is the availability of important features. Auto-scaling is the most important feature they provide. The number of requests being processed on a serverless system causes the system to scale its resources as needed. With rising traffic, extra instances of a function are set up, and resources are scaled-down when traffic lowers. Efficiency and affordability are guaranteed since people only cover the cost when the server is in use and not for unused time. The event-driven model is a main feature of serverless platforms (Pérez et al, 2019). When triggered by a request, file upload, or database update, serverless applications immediately execute a function. Because of this, systems can be designed in modules, and various components or pieces of the system can be developed and used separately. Besides, serverless platforms usually enable monitoring and logging to help developers inspect the performance of their functions as they happen.

2.3 Overview of Popular Serverless Platforms

AWS Lambda and Google Cloud Run are popular serverless computing platforms that several cloud service providers supply. AWS Lambda lets developers handle events with code without them having to manage the servers needed to run that code. AWS S3, DynamoDB, and API Gateway services can start Lambda functions. Google Cloud Run allows users to run stateless HTTP containers in an effortlessly managed environment (Khatiwada & Dhakal, 2024). When the number of people accessing the website increases, the server automatically responds and supports landing pages written in Java, Python, and Go. Both platforms are helpful in various circumstances because they are auto-scalable, handle events promptly, and connect well with other cloud services.

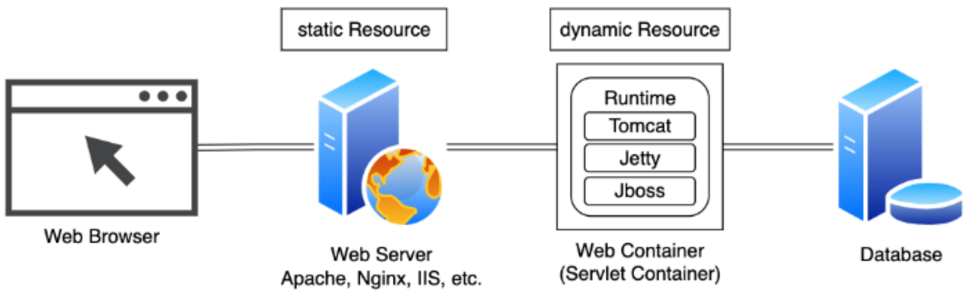


Figure 2: Web server and Web Container architecture model

2.4 Benefitss and Challenges of Serverless Computing

Serverless computing is especially valuable because it reduces costs for many users. Because serverless platforms charge only for time spent processing, they enable savings when the apps are not busy. The approach makes it simple to develop and put applications into service promptly. Because of this, developers need only write code for their role. Even so, serverless computing has a few problems. A significant disadvantage is that starting up a function after it has not been invoked for a while can be slow. Poorly timed cold starts can cause applications to be slower whenever the traffic is not steady. Serverless platforms usually scale well, but the problem of cold starts is a prominent concern developers must solve when improving their serverless applications (Golec et al, 2024).

3. Introduction to Cold Starts

When a serverless service is " asleep" and accessed for the first time, it has extra time to start up. On platforms such as Google Cloud Run and AWS Lambda, there is no physical server to handle, and users charge for how long their functions take to execute. Even so, if the function has not been executed in some time, the platform must allocate resources for initialization before it can perform the desired job. There is a wait in the process, known as the "cold start." Since serverless platforms automatically scale resources based on use, it takes time for them to process the first or next call after prolonged inactivity. Latency-sensitive applications often see their performance reduced by serverless function cold starts. Because of the waiting time during a cold start, users might experience longer delays than usual and feel the effect of a poor user experience (Crescenzi et al, 2016). Chatbots, APIs, and e-commerce websites with real-time needs face particular problems because users expect quick responses. The problem of cold start latency comes from when the environment needs to gather resources, dependencies, and runtimes for serverless functions. Most of the time, this leads to noticeably poorer performance, mainly when a process is run for the first time.

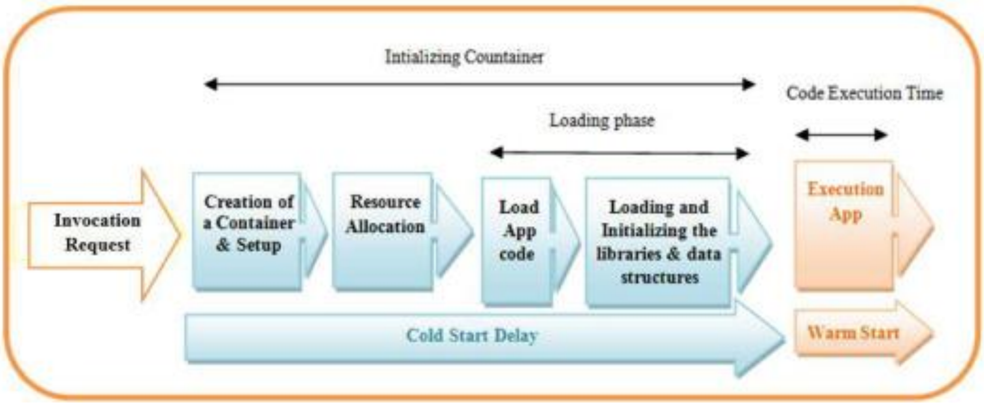


Figure 3: Cold start latency mitigation mechanisms in serverless computing

Cold-starting Java serverless applications can create serious problems with latency. Since Java must start by loading the JVM, setting up class dependencies, and preparing its environment, java programs take longer to launch. The first part of this process requires many resources and can hold up the beginning of the serverless feature. Apps created with Spring Boot and many related libraries usually take longer to start after a break (Reddy, 2017). Because the startup time with Java and Spring Boot may take a few seconds to tens of seconds, it can be challenging to ensure smooth performance in pressure situations. In the real world, having a low number of users usually results in problems for the user and the business. In an application that requires quick responses, delays in starting because of cold starts can influence things like late transactions, unhappy consumers, and missed business opportunities. Like in web apps, when serverless functions are used in a mobile app to receive live data, the slow response due to cold starts can make the app run poorly, and users tend to leave. This delay can significantly affect users who expect fast responses, such as those using games, voice assistants, or real-time web services. Identifying and solving cold-start issues is vital when moving from traditional, complete apps to independent microservices. In Chavan's (2022) view, knowing the boundaries of an application context is necessary while making architectural updates to achieve easy migration and the best results. Likewise, dealing with cold start problems in serverless systems is necessary to keep the application's environment active, steady, and scalable.

4. Cloud Run and Its Role in Serverless Java

4.1 Overview of Google Cloud Run as a serverless Compute Platform

Google Cloud Run manages containers automatically, allowing the application to run serverless. With FaaS, developers do not have to worry about managing infrastructure, which makes it suitable for serverless computing. With Cloud Run, developers can skip worrying about servers, scaling, and load balancing and instead concentrate on developing and deploying their applications. With Docker support, Cloud Run allows developers to choose any language, framework, or dependency for their applications (Mouat, 2015). Thanks to its integration with Google Cloud, Cloud Run helps developers build secure apps that grow with use while only requiring payment for what was used during processing. Because of this design, it is no longer necessary to allocate excessive resources to each user as was required with traditional server-based systems.

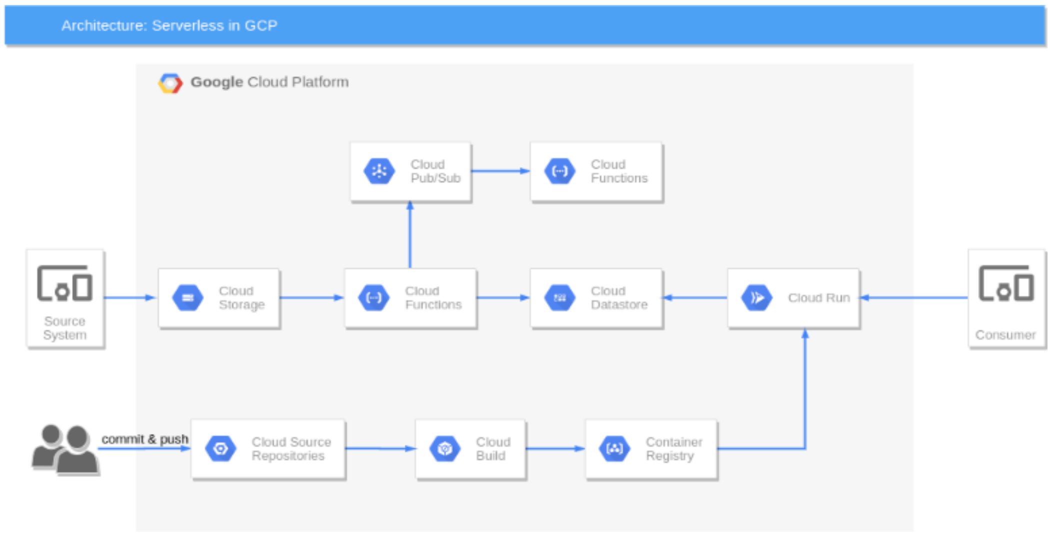


Figure 4: Serverless on Google Cloud Platform

4.2 How to Run Works with Docker Containers to Deploy Serverless Applications

The execution foundation for Cloud Run is Docker containers. A Docker container includes an application, its dependencies, configurations, and running environment, so it looks the same everywhere it is used. If a developer uses Docker to build a container for their app, it can be put onto Cloud Run, and Google Cloud will arrange for the app to be scaled, handled through load balancing, and routed. When deploying the application to Cloud Run, the serverless approach is applied. Once the application is live on Cloud Run, it scales up or down based on how much traffic is received without anyone needing to manage it. Cloud Run shrinks to nothing if nothing is served, meaning there is no need to pay for idle servers (Fox et al, 2009). When traffic picks up, Cloud Run automatically increases the application's resources so users experience no interruptions. Cloud Run's support for HTTP(S) requests enables it to help build RESTful APIs, web services, and applications that work with HTTP traffic. Java's flexibility in working with numerous web tools allows it to work well with Spring Boot frameworks when launching Java applications.

4.3 The Advantages and Disadvantages of Using Cloud Run with Java Application

Java developers will find that Cloud Run offers many valuable features. A significant benefit is its ease of deployment. Packaging Java applications into Docker ensures that they work consistently across different environments. As a result, moving an application from a local machine to a production system is simple, with no need to modify environment-specific configurations. Another important advantage of Cloud Run is its ability to scale automatically. Spring Boot-based Java apps, among others, can handle varying amounts of web traffic. Cloud Run adjusts container instances in response to incoming requests, eliminating the need for manual management. Cloud Run also automatically shuts down resources when unnecessary, saving money for unpredictable apps. However, using Cloud Run with Java presents some challenges. The primary issue is known as the cold start problem (Manner, et al, 2018). Container startup takes time, so a request arriving after a container has been inactive results in latency. This delay, referred to as a cold start, can degrade the user experience for apps that require quick responses. Although Cloud Run supports keeping containers warm with minimum instance settings to reduce cold starts, latency continues to be an issue for such applications.

4.4 The Integration of Cloud Run with Spring Boot and Its Significance

Spring Boot, a popular framework for web applications, makes it easier to develop Java web services and microservices. This feature matters because Spring Boot users can deploy their applications on Cloud Run without managing their servers. If Docker containers are used, Spring Boot applications can easily be deployed to Cloud Run. Because of this combination, developers can use Spring Boot tools like dependency injection, configuration setup, and strong security on Cloud Run. Developers can create REST APIs with Spring Boot that adjust to the number of requests automatically without having to worry much about servers. In addition, teams can benefit from Cloud Run and Spring Boot's significant flexibility and control. Developers can make their Spring Boot applications run inside containers, simplifying how they manage dependencies and maintaining consistency throughout development. Since some Java applications have specific needs and libraries, Cloud Run is useful because it allows developers to define their environments through Dockerfiles (Konneru, 2021).

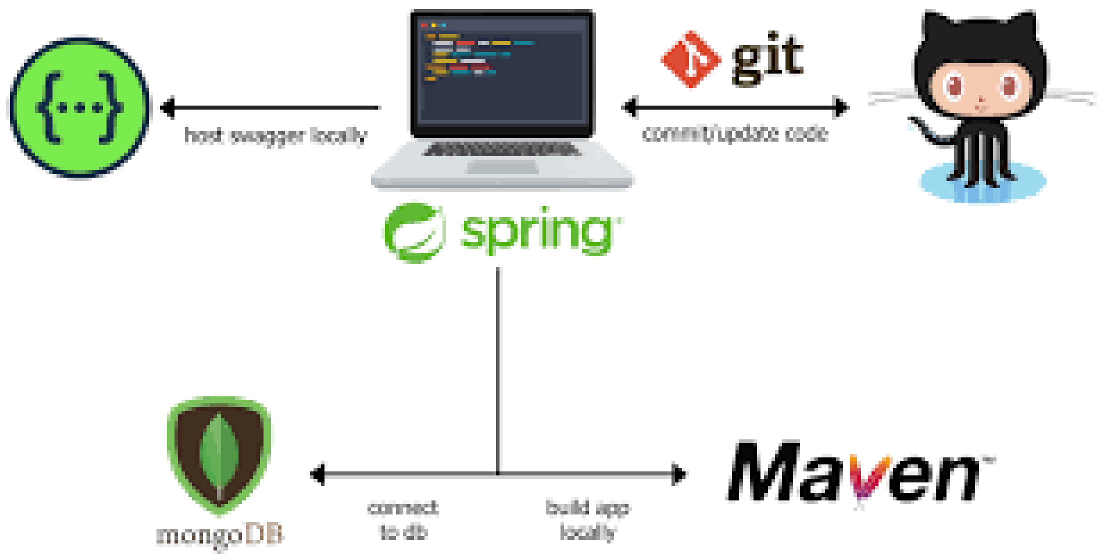


Figure 5: How to Design a RESTful Spring Boot API

5. Understanding Spring Boot in Serverless Environments

5.1 Overview of Using Spring Boot to Create Java Microservices

Thanks to its tools', Spring Boot is ideal for Java developers who want to build microservices easily. It performs well by enabling simple deployment, easy configuration, and smooth scaling of production code. Java apps have been considered heavy on resources. However, Spring Boot reduces this by offering prepared templates and letting developers concentrate on what their software does rather than how it runs. Since Spring Boot relies on conventions, manual configuration is unnecessary (Sardana, 2022). Applications created with Spring Boot are easy to structure as microservices since each service can be deployed separately. Every microservice works in isolation and can interact with others using the network, which supports cloud-native solutions. The two go hand in hand because serverless environments are becoming widely used for their quick ability to scale.

5.2 Spring Boot's Integrating with Serverless Frameworks

Because serverless computing manages the infrastructure, programmers can focus only on the code. Easy serverless environments for applications are available through AWS Lambda, Google Cloud Functions, and Azure Functions cloud platforms. They are built so that activities (functions or services) occur whenever something important happens, such as an HTTP request, a file being uploaded, or data changes in the database. Thanks to Spring Boot, it is simple for developers to build microservices that run smoothly on Google Cloud Run and other serverless platforms. With Cloud Run, developers can put containerized applications into the cloud and allow them to scale automatically when needed. With Spring Boot and Cloud Run, Spring Boot applications are packaged as Docker images and deployed on Cloud Run (Larsson, 2023). This approach makes use of the powerful tools from Spring Boot and also uses serverless services, which charge by use and can scale up or down as needed.

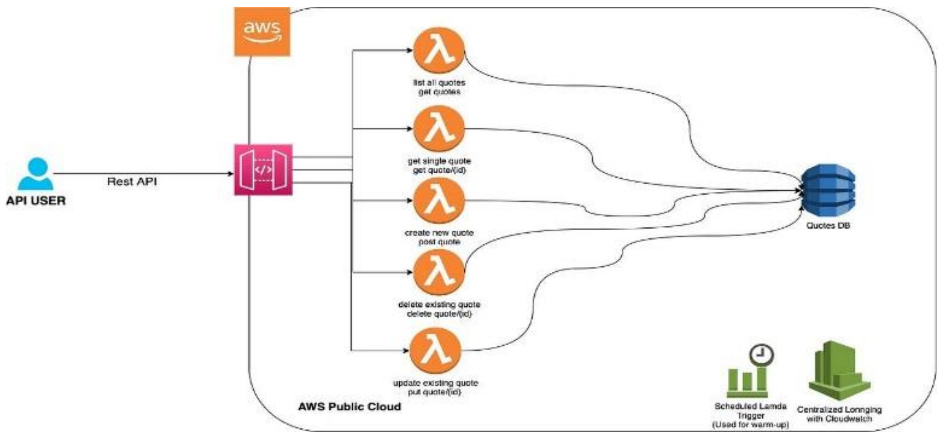


Figure 6: Serverless Framework with Spring Boot & AWS Lambda

5.3 Advantages of Using Spring Boot Is Helpful for Cloud Applications

Using Spring Boot makes building cloud-native applications easy, with a special advantage in serverless environments. First, Spring Boot sets up a ready environment that has embedded servers, health checkpoints, and metrics by default, helping designers work smoothly with cloud tools. Because of this approach, developers can easily develop and deploy their applications without concerning themselves with complex configurations. Secondly, Spring Boot is suitable for working with microservices structures, which is necessary for constructing applications designed for the cloud. Thanks to domain-driven design, developers can create distinct services that are not connected, which makes it easier to build, test, and scale applications. This aligns closely with the ideas behind serverless computing, where services grow independently, depending on demand. In addition, Spring Boot has access to many tools that support working with databases, queuing messages, handling authentication, and other advanced cloud-focused features, helping make app development more successful (Jandl, 2020).

5.4 Points to Keep in Mind When Using Spring Boot on Cloud Run

While it's beneficial to run Spring Boot applications with Cloud Run, developers must keep certain things in mind. A main issue with serverless platforms is a cold start. Cold start happens when a serverless function takes time to load after being inactive for some period. As a result, the cloud provider must prepare the application first, which contributes to a longer response time. In these serverless setups, keeping cold start latency low is essential, specifically in applications that need rapid, near-real-time responses. Because Spring Boot applications start up more slowly than many lightweight frameworks, this can be a challenge for them (Jani, 2020). However, specific methods can be used to overcome challenges with a new business. One instance of this is using GraalVM to compile Spring Boot applications into native images, which decreases the time needed to start and makes it easier to run them in serverless settings. Moreover, an application's setup can affect how fast it runs when it starts for the first time. For example, streamlining a Spring Boot project and removing extra components can take up to 50 percent less time to start. Because Cloud Run can turn down to zero resources with no traffic, resource consumption is managed well, but the additional scaling up may make starting requests slower.

6. Cold Start Latency Serverless Java Applications

Using serverless computing, developers do not need to worry about servers because the system handles these tasks (Wen et al, 2021). Cloud providers offer serverless tools like Google Cloud Run and automatically adjust their resources according to demand, but one issue they present is cold start latency. A cold start happens when a serverless function is called for the first time since being inactive or turned off, so it has to be activated anew. Setting up the environment and loading resources takes time, so response times suffer. This section examines the science of cold starts within Java applications, compares their speed to other languages, explores the underlying causes of these issues, and analyzes how Cloud Run addresses them for Java applications.

6.1 The Science Behind Cold Starts in Serverless Environments

A function in serverless environments is defined as experiencing a cold start if the environment needs to be initialized when the function is called. In serverless environments, devices such as virtual machines or containers generate a response to an incoming demand. After no activity, the cloud provider has to start the function's environment, add its code, and initialize its dependencies before processing the request. How fast an operation takes to carry out from scratch is known as cold start latency. Cold Start Timings Comparing Java to Other Programming Languages. Many people mention that Java runs slowly because it takes longer for the JVM to start. Because of its cold start times, Java is slower than interpreted languages like Python or Node.js. The problem is that it takes the JVM a considerable time to start up, which is particularly important in serverless situations where quick scaling is necessary. A typical cold start for Node.js or Python occurs within a timeframe of 200–500 milliseconds, whereas for Java, several seconds of waiting are required. The key reason is that the JVM must take care of loading classes, starting garbage collection, and preparing the runtime environment (Singh, 2022). While interval-based hot start works well, it can cause latency increases in cases where low response is expected.

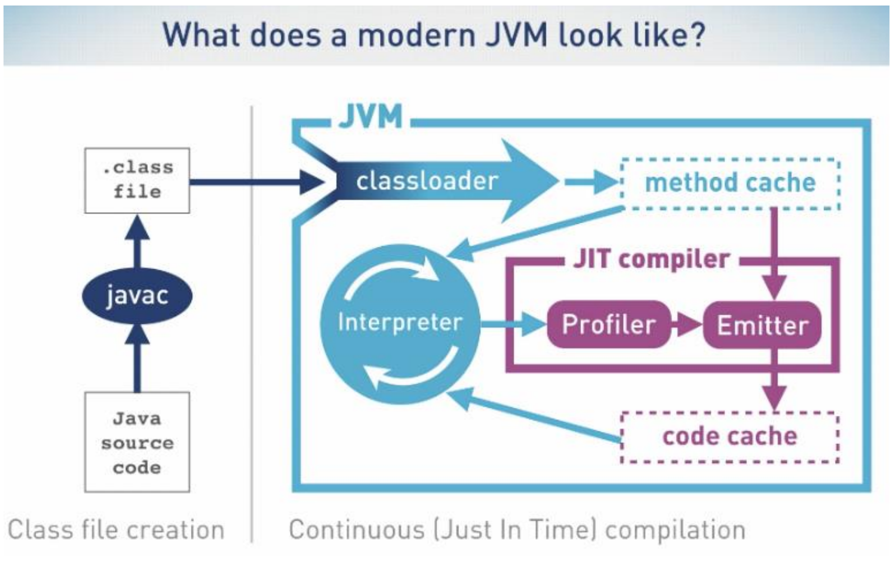


Figure 7: Recap the Java(JVM) : after reading Optimizing Java

6.2 Factors Contributing to Cold Start Problem

There are many reasons why starting Java serverless applications takes longer than expected. The Java Virtual Machine must first deal with many libraries and prepare several components before it starts. The lengthy startup process for the JVM is one reason cold start latency is experienced. Besides, Java applications require much memory, so more resources are needed for their setup. When a Java application starts, it needs to initialize its dependencies, which usually include third-party libraries or frameworks such as Spring Boot. Depending on these services, the first startup takes extra time. Java employs class loading and reflection extensively, which means it checks and loads classes during execution. Since classes are delivered to the runtime dynamically, it can take a bit more time for the program to execute. The JVM has built-in background garbage collection (GC), so there is no need to manage it. However, the processes involved in building GC and memory management tend to increase the size of the cold start overhead (Wu et al, 2017).

6.3 What Happens with Cold Starts in Java Apps on Cloud Run

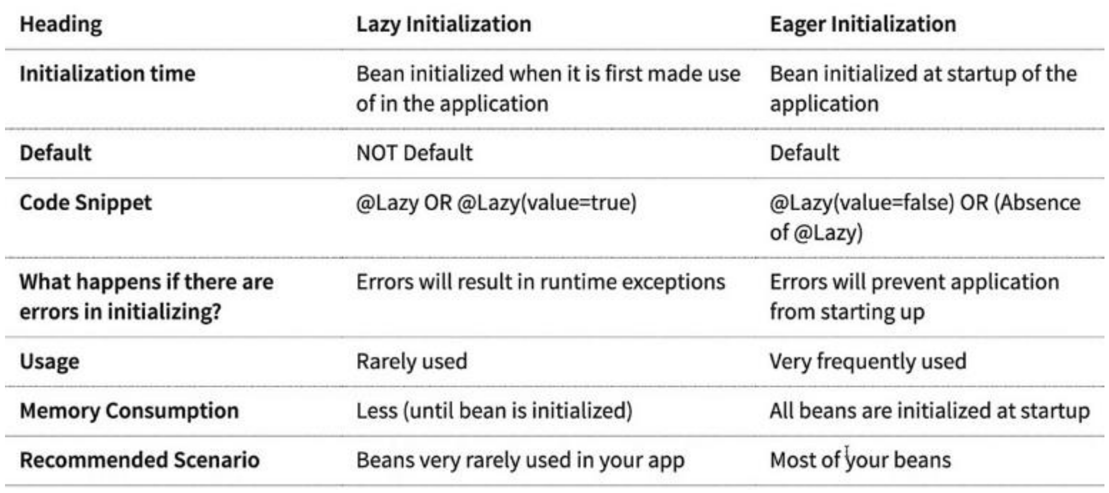
Google Cloud's Cloud Run automatically adjusts the size of containers in answer to incoming requests. When a request arrives, it creates and starts containers as is required for cold starting. Java applications using Cloud Run will often find that could start latency has been lowered using techniques such as: In Cloud Run, Java applications are turned into efficient container images that improve performance. Keeping the image small and with few dependencies makes Cloud Run launch the container much more quickly. Cloud Run includes a feature called "minimum instances," which ensures that a predefined number of containers remain running even when traffic is low. As a result, containers do not have to be initialized each time, so cold starts occur less frequently. Cloud Run manages Java memory in Java applications by responding to the number of users at any moment. It helps decrease the load of giving out extra resources, which may increase cold start latency. Still, launching Java applications can take a lot of time, especially when the code and its dependencies are not simple. Cloud Run uses container technology and fine-tunes the JVM start to limit the effect of cold start latency on Java applications (Heydari Beni, 2021).

7. Cold Start Mitigation Strategies in Cloud Run/Spring Boot

The initial launching of application instances in Google Cloud Run by Spring Boot can be slow and may delay performance. These delays are caused by having to start an application from the beginning, making users' response time longer. Strategies such as better application startup, certain Google Cloud services, and adjusted Docker images can reduce any of these troubles.

7.1 Optimizing Application Startup

A good method to reduce cold start times is to shorten the startup time of a Spring Boot application. Customizing how Spring Boot starts through useful configurations can help. For example, turning off unused auto-configuration classes and removing non-essential default beans speeds up application startup for a given service. Additionally, using Spring.main.lazy-initialization, which postpones bean initialization until they are requested, can make the startup much quicker. Another technique is to limit the project's dependencies. Applications built in Spring Boot often result in large classpath sizes (Sadakath, 2018). For every dependency used, the framework takes extra time to scan all the classes at startup. By ensuring the application has only essential modules and is structured as smaller services, the application will initialize much more quickly. Turning Spring Boot applications into native images with GraalVM can significantly speed up their launch. Since native images are precompiled, unlike Java bytecode, the programs behind them start in record time, usually within milliseconds. Although producing native images takes longer, the boost in cold start performance can be beneficial for Cloud Run, where fast response times are critical.



Heading	Lazy Initialization	Eager Initialization
Initialization time	Bean initialized when it is first made use of in the application	Bean initialized at startup of the application
Default	NOT Default	Default
Code Snippet	@Lazy OR @Lazy(value=true)	@Lazy(value=false) OR (Absence of @Lazy)
What happens if there are errors in initializing?	Errors will result in runtime exceptions	Errors will prevent application from starting up
Usage	Rarely used	Very frequently used
Memory Consumption	Less (until bean is initialized)	All beans are initialized at startup
Recommended Scenario	Beans very rarely used in your app	Most of your beans

Figure 8: Lazy Initialization and Eager Initialization in Spring

7.2 Leveraging Warm-up Activities

It is important to take time to warm up, as it reduces the shortcomings of cold starts. It is common to ping the application every several minutes to prevent it from going cold. As a result of this technique, Cloud Run instances continue to function and answer any requests from real users. However, the cost and design of the system may make this solution impractical in some instances. Cold starts can be reduced by setting up the concurrency settings on Cloud Run. Allowing several requests into a single container helps Cloud Run require a smaller number of instances to deal with new traffic, so new instances do not have to undergo so many cold starts. Appropriately setting up the number of concurrent tasks depends on expected usage, which helps decrease both cold start issues and their linked latencies (Singh, 2022).

7.3 Optimizing Docker Images for Java

Cold start in Cloud Run for a Java application depends greatly on the size of the Docker image. The time it takes for a webpage to start up increases with each large image it shows. For this reason, it's necessary to optimize Docker images for Java applications to overcome the cold start problem. One way to enhance Java Docker image performance is to use multi-stage builds by copying the essential dependencies from a build image into a running image. Choosing base images that are trimmed down, for example, openjdk:11-jre-slim, also matters. By shrinking the image size, the time it takes for Cloud Run to pull it during launch is shortened. Valuable benefits for Java applications include using JLink, a tool that packages a Java runtime based on user needs. An improved cold start occurs because eliminating unneeded modules from the runtime reduces the Docker image's size (Lin et al, 2020).

7.4 Applying Google Cloud Tools

With Google Cloud, developers can use special features that minimize the time needed for serverless applications to run again. Cloud Run automatically changes the number of running containers based on the amount of traffic it receives. As a result, resources will be distributed more efficiently, which will help reduce the effect of cold starts. Setting a low minimum number lets users prevent many cold starts when the app is not jam-packed. Another important part of Cloud Run is support for controlling concurrency settings. The system produces fewer new container instances because Cloud Run allows requests to be sent to the same container instance many times (Zhang et al, 2010). This setting can help managers ensure containers stay fast and accessible, with minimal cold starts.

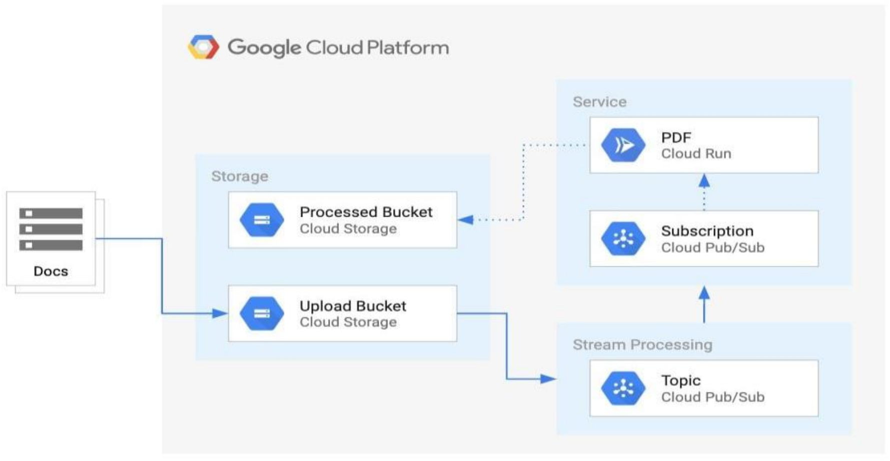


Figure 9: Switching to Serverless with Google Cloud Platform

8. Research Methodology

This research aims to study and assess techniques for reducing cold starts in serverless Java programs deployed on Google Cloud Run with Spring Boot. Running Java apps on a serverless platform can be slow due to the significant problem of cold start latency. This research examines several approaches to solving cold-start problems and enhancing performance and user experience. A mix of qualitative and quantitative techniques was chosen in this study to examine the problem and its answers.

8.1 Research Design

This study uses both qualitative and quantitative methods to provide a combined approach. Different techniques are applied to better understand the theory behind cold start latency with Java applications running on Cloud Run. The research involved a complete survey of studies related to serverless computing, cold starts, and Spring Boot applications. The investigation examines actual performance in case studies and measurements against set targets to examine the results of various cold start mitigation methods. The research design also examines case studies of serverless applications on Google Cloud Run. These case studies explain some of the difficulties developers face when using Java with serverless services. The design includes carrying out both benchmarking and latency tests on Java applications built with Spring Boot that run in Cloud Run (Christudas, 2019). This method will show how different approaches to cold starts change the app's response times and performance.

8.2 Data Collection

Most of the data for this research is collected from Google Cloud monitoring tools and testing with Java applications on Cloud Run using Spring Boot. Cloud Run monitoring tools give real-time information about cold starts, service delays, and resource use. This information can help us identify trends in cold start behavior, which leads to discovering ways to address this issue. Applications created with Java will be deployed to Cloud Run, and their latency during cold start events will be reviewed. Monitor response time after fixed waits before and after enhancements are applied to the code. Testing in a controlled way ensures that the experiments all use the same variables. The Google Cloud Run logs will support data collected from runs, revealing what resources the apps need during down and upstarts (Bhattacharjee, 2009). This approach lets us study how various ways to fix bad caches influence how well a Java application operates and its resource use.

8.3 Analysis Approach

Cold start mitigation strategies are assessed using two statistical methods in the analysis approach. First, cold start times will be examined, checking how much the application's latency has been reduced by the new strategies. This means calculating the average startup time and how much the values vary using standard deviation. Reductions in how long it takes to start the application after applying specific strategies are a sign that the strategies are successful. The research will compare how Java applications perform when deployed using a traditional virtual machine and when deployed using Cloud Run (Kumar, 2019). This section discusses cold start latency, resource utilization, and application behavior during both high and low traffic periods. The aim is to determine if serverless works better than other options when cold start strategies are in place. The research will also analyze how Spring Boot and Cloud Run affect serverless architecture and cold start behavior. The study will help identify which strategies best serve Java serverless applications by examining the pros and cons of several cold-start solutions.

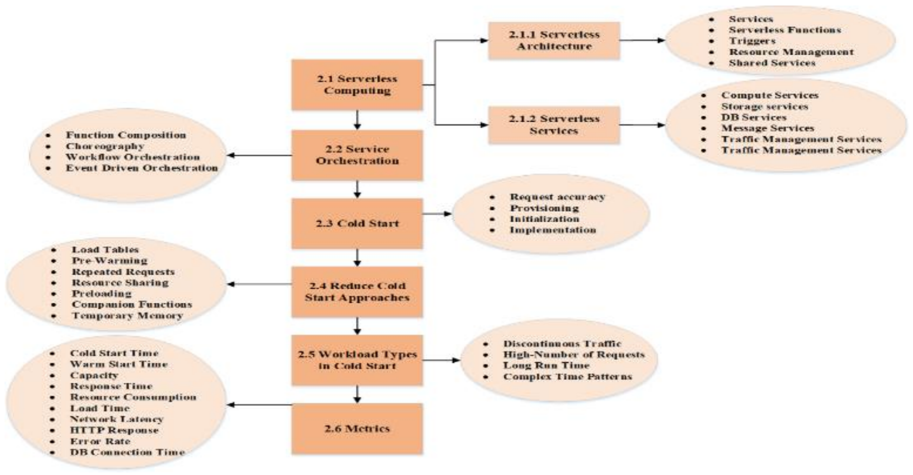


Figure 10: A survey on the cold start latency approaches in serverless computing

9. Real-World Examples and Case Studies

Many use serverless computing because microservices need to be scalable and easy to operate. Still, cold starts—the wait for a container without recent activity to start up again—can be a major problem for Java applications, for example, those built using Spring Boot. This paper discusses three case studies to show how to address cold starts in Cloud Run and assess Java performance in serverless environments.

9.1 Case Study 1: Optimizing a Spring Boot Application for Cloud Run with Cold Start Mitigation

Spring Boot's architecture, which a mid-sized financial services firm had adopted for years, was switched to microservices that use Google Cloud Run. Observations from the first set of tests showed that Spring Boot-supported user authentication services took 2.5 to 4 seconds to run when little activity was present. Therefore, the team built extra optimization layers, among them GraalVM, which compiled the Spring Boot application into a native image and thus decreased the JVM startup time. At the same time, the team turned on concurrency settings in Cloud Run and arranged for minimum instances to ensure one was always warm. Because of these improvements, cold start latency was reduced below 800 milliseconds, making authentication endpoints much smoother for users (Joosen et al, 2023).

9.2 Case Study 2: Performance Improvement Observed After Implementing Warm-Up Strategies and Image Optimization

Customers worldwide could not purchase because of high latency problems during the final stage of the checkout. The checkout, run on Cloud Run using Spring Boot, faced cold starts frequently when the traffic was low. Engineers decided to deploy a lightweight pinging technology, set to execute routinely and call the endpoint to avoid making the container fall asleep. They also worked on improving the container image by making it smaller by removing extra things and choosing a basic image (e.g., distroless Java). After using these approaches, the time it takes to start a car dropped from 3 seconds to under 1 second. Such improvements follow industry standards for handling scalability and expenses in cloud-based applications, allowing companies to preserve performance with minimal new costs (Chavan and Romanov, 2023).

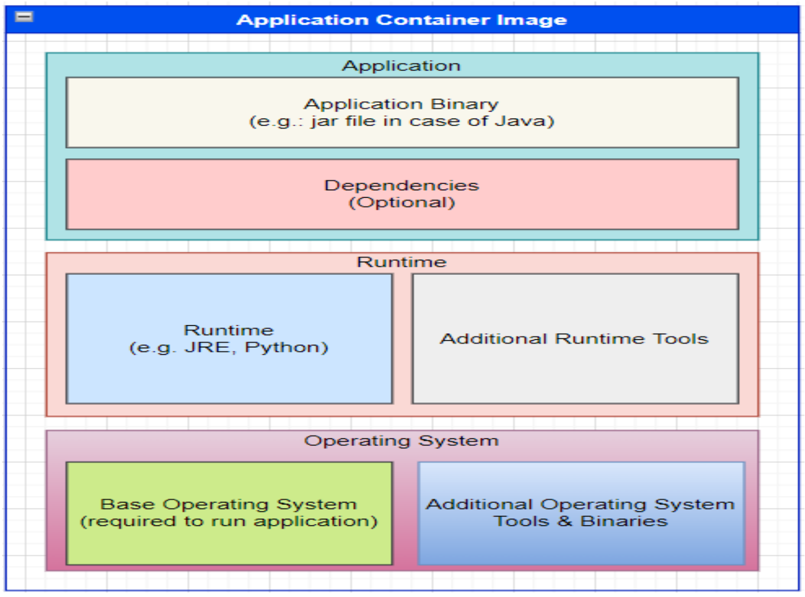


Figure 11: Distroless Image in Docker and Kubernetes Environment

9.3 Case Study 3: Comparing Java Applications' Performance on Cloud Run vs.AWS Lambda

The healthcare tech startup analyzed both serverless platforms to pick the right one for a patient intake application written in Java. The analysis examined how quickly both Google Cloud Run and AWS Lambda start, how easy they are to set up, and how much they cost. While AWS Lambda managed faster cold starts for Java programs, its performance fell behind when starting the complete Spring Boot application, which usually took more than 5 seconds. Alternatively, adjusting configurations and keeping warm instances in Cloud Run resulted in better, more predictable performance for large Spring Boot apps than anywhere else. The analysis found that compared to AWS Lambda, Cloud Run provides more support for larger Java functions by giving developers more ways to set up and use them efficiently (Villamizar et al, 2017).

10. Tools and Techniques for Cold Start Mitigation

Using Java technologies such as Spring Boot and deploying to Google Cloud Run makes cold starts a serious issue in serverless computing. Latency happens with a cold start when a serverless function is called after a long idle period. This part of the discussion covers useful tools and approaches for developers who want to handle slow activation of serverless Java applications, focusing on GraalVM native compilation, optimizing Spring Boot, altering configuration in Cloud Run, and observing performance.

10.1 GraalVM and Native Compilation

Using GraalVM for native image compilation can prevent many cold starts in Java applications. Traditional Java applications start on the JVM and wait for bytecode processing and JIT compilation, which results in significant slowness when launched. GraalVM handles this problem through AOT compilation, which enables Java applications to be turned into executables ahead of running them. Reducing startup time and the need for memory, it is perfect for use in serverless settings. By running unused code with GraalVM, the full capacity of the hardware can be utilized for continued fast execution (Fumero et al, 2015). Quickly optimizing in the cloud plays a significant role in making workflows more efficient, a concept GraalVM excels at. If JIT compilation is not performed at runtime, GraalVM-native binaries start up faster, reducing the time to start applications in Google Cloud Run.

10.2 Optimizing Spring Boot configurations

Strong as it is, Spring Boot is recognized for its sluggish startup process, which is caused by its dependency injection and auto-configuration systems. Still, there are several useful steps that can speed up cold start configuration. Spring Boot suggests that developers first use lazy initialization, so resources are not created until they are necessary. This decreases the amount of work needed to start up the application. Another practical approach is to exclude automatic configurations and drop unnecessary dependencies. Moving away from Tomcat and using other thin options like Netty or Undertow can speed up web application startup. With these improvements, Spring Boot applications work well in serverless deployments. Spring Boot 3 and Spring Native enhance the process by smoothly combining with GraalVM, providing native compilation support, and performing better in cold-environment startups. Combining optimized configuration with native compilation gives developers a strong way to deploy Java applications in serverless setups (Al-Maamari, 2016).

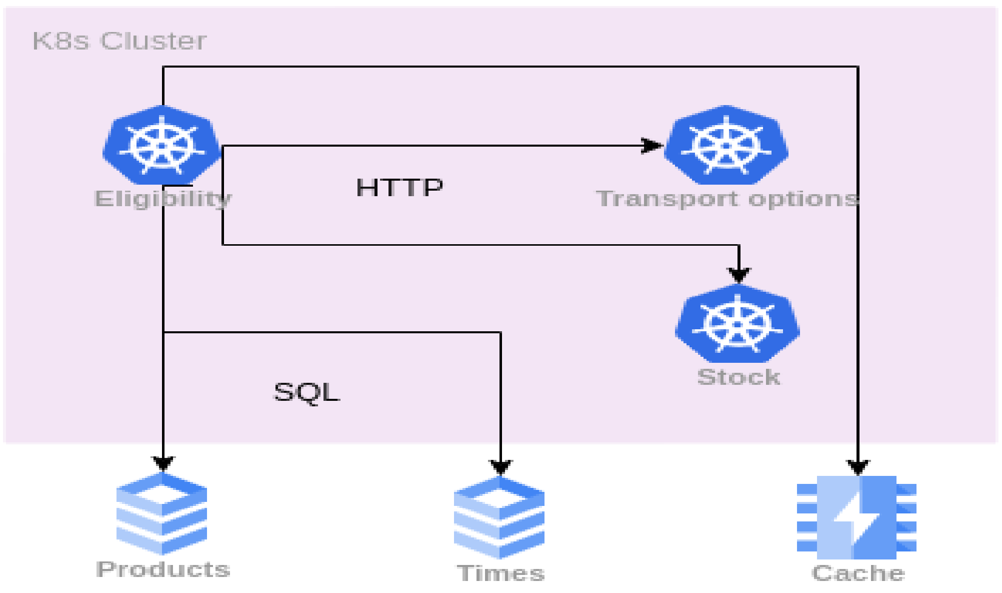


Figure 12: A service with Spring Boot 3 and GraalVM native in the real world

10.3 Cloud Run Best Practices

With Google Cloud Run, several configuration settings can be adjusted to lower cold start latency. One approach is the min instance configuration, where certain container instances are kept running to ensure there are no cold starts for a predicted percentage of regular traffic. There may be extra costs, but it dramatically helps latency-sensitive applications work more efficiently. To optimize images, developers can shrink them, pick slim base images (such as distroless), and avoid layers that are not required. As a result, containers initialize faster. Injecting the environment and secrets during the build process can keep the container startup speed as high as possible. To add, using a smaller request concurrency limits the number of simultaneous requests an instance receives, which can make it operate more smoothly. These best practices agree with Nyati's (2018) focus on efficiency via algorithms that fit the needs of a system.

10.4 Performance Monitoring Tools

Monitoring and being observable are very important for fixing cold-start issues. Developers can rely on Cloud Monitoring and Trace, both included in Google Cloud solutions, to watch startup times and measure how fast instances are made and the duration of requests. Based on data, these tools can optimize serverless applications. These logs and metrics allow developers to find patterns, establish alarms, and activate policies that predict when demand will grow and prevent delays from cold starts. Knowing where problems occur allows teams to fix them quickly.

11. Future of Serverless Java and Cold Start Mitigation

Overcoming the persistent problem of cold starts is changing the outlook for serverless Java using Google Cloud Run and other platforms (Wyld, 2009). Java suffers significant startup delays because the JVM works in combination with the language. Even so, new trends and technologies could make a big difference. One key trend in serverless Java is the rise of AOT compilation, especially because of tools such as GraalVM. Unlike other JVM deployments, GraalVM can build Java applications as native binaries before they are run. So, they load much faster, reducing the amount of cold start experienced. More efficient handling of binaries by serverless platforms means Java is more likely to work well in places like Cloud Run. Also, new features on Cloud Run, such as saving extra CPU for suited times and preheating containers before use, help containers stay active partway through the day. These updates, new autoscaling, and predictive features indicate that the cloud will automatically reduce cold starts. Using container snapshotting to archive a warm container's memory and state could lead to almost immediate recovery for serverless containers. Improvements in JVM optimization continue to make a big difference. Project Leyden is developed by the Java community to improve the speed to start, resource use, and overall performance of Java applications. With improvements in static image generation and configuration trimming during builds, the JVM may close the gap in performance with languages that are native compilers. When these innovations are made standard, developer teams can use Java's strengths without facing large delays.

Performance and reliability can clash, mainly when systems choose speed above all other aspects (Abbott, 1990). This proves that having a balanced system design is critical when using serverless Java. Cold start latency should be decreased, but developers must avoid steps affecting application stability or keeping data inconsistent. This is especially necessary for servers like MongoDB, which work better when carefully orchestrated through connection pools and transactions in stateless environments. In the future, tools for avoiding cold starts will be more advanced and easier to use throughout the entire software development process. They may help serverless platforms detect periods of high demand and start additional computing resources on demand. At the same time, refining how certain operations run in real time might become an ordinary practice in serverless Java development. The future promises great things for serverless Java in platforms like Cloud Run, with improvements in JVM performance, AOT compilation, and cloud features combined to deal with the cold start issue. Performance and reliability need to go hand in hand, and this idea will shape the growth of serverless Java and cold start mitigation (Dhanagari, 2024).

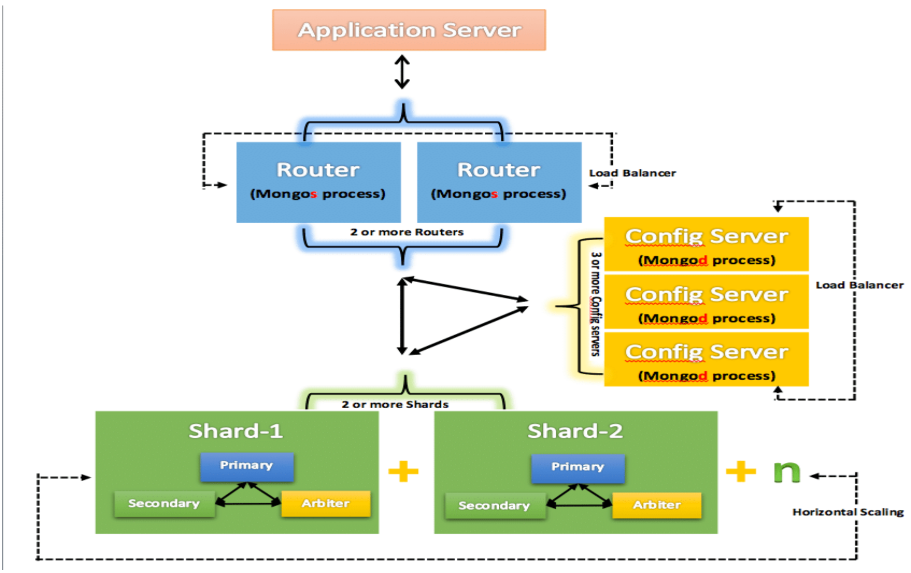


Figure 13: MongoDB: An Overview

12. CONCLUSION

Launching Java apps in serverless settings such as Google Cloud Run often faces the most significant difficulty known as cold start latency. With standard architectures, applications run all the time, but with serverless, application instances are created only when demanded, which adds an initial delay. Because Java applications are slow to launch, use much memory, and rely on JIT, the delay they experience at a startup or their cold start is a significant difficulty. Cold starts can be a problem in performance-critical applications, leading to user experience and a loss of agreement with service-level goals. This research explores the causes of cold starts in serverless Java deployments and explains several strategies to minimize cold starts in Cloud Run and Spring Boot systems. Methods for these strategies are to optimize container images, rely on automatic image compilation with GraalVM, allot CPU power from Cloud Run during downtime, switch to lightweight frameworks like Micronaut and Quarkus, and fine-tune the starting process of theapplications. All these strategies are balanced differently in terms of how complex they are, how well they perform, and how they fit into a network, making it important to pick the strategy that fits a particular application or what workers need to do.

Cloud Run allows for easy scaling, but special planning is required for how applications boot and manage their resources as a container-based serverless service. Using traditional application servers and big frameworks in a Java environment can lead to intolerable latency unless steps are taken to optimize them. So, developers should implement new approaches such as limiting dependents, loading components on demand, and warming up services when feasible. The new features in Cloud Run, such as setting a CPU that never goes idle and adjusting the number of running instances, now allow developers to manage performance and costs easily. While Spring Boot is extensively valued for its ease of use and support of multiple ecosystems, it is not optimized for quick serverless startups. However, if changes are included—including turning off autoconfiguration, using lazy initialization, and having some work done in the background—Spring Boot will fit better in Cloud Run. They might also take advantage of Spring Native or AOT, which helps Java projects use Spring better and boosts how fast the application comes online.

The future for serverless Java is bright. GraalVM's growth, enhanced support for native compilation, and use of serverless as a focus has reduced the distance between Java and faster languages like Node.js or Go. The platforms from cloud providers are being updated to make Java workloads easier by providing more control over resources, additional observability tools, and warm instances that keep running. All things considered, using Java serverless architecture should not be discouraged by the possibility of cold starts. With suitable combinations of features, architecture, and code improvements, Java applications run smoothly and quickly when launched in Cloud Run. Because the ecosystem is evolving, it will soon become much easier to mitigate cold starts, so developers will not have to trade off performance for the benefits of scalability and lower expenses. Following these tips now helps businesses prepare for tomorrow's cloud world on the reliable foundation of the Java ecosystem.

REFERENCES

1. Abbott, R. J. (1990). Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys (CSUR)*, *22*(1), 35-68.
2. Al-Maamari, T. A. A. (2016). *Aspects of event-driven cloud-native application development* (Master's thesis).
3. Bhattacharjee, R. (2009). *An analysis of the cloud computing platform* (Doctoral dissertation, Massachusetts Institute Of Technology).
4. Chavan, A. (2022). Importance of identifying and establishing context boundaries while migrating from monolith to microservices. *Journal of Engineering and Applied Sciences Technology, 4*, E168. http://doi.org/10.47363/JEAST/2022(4)E168
5. Chavan, A. (2024). Fault-tolerant event-driven systems: Techniques and best practices. *Journal of Engineering and Applied Sciences Technology, 6*, E167. https://doi.org/10.47363/JEAST/2024(6)E167
6. Chavan, A., & Romanov, Y. (2023). Managing scalability and cost in microservices architecture: Balancing infinite scalability with financial constraints. *Journal of Artificial Intelligence & Cloud Computing, 5*, E102. https://doi.org/10.47363/JMHC/2023(5)E102
7. Christudas, B. (2019). *Practical microservices architectural patterns: event-based java microservices with spring boot and spring cloud*. Apress.
8. Crescenzi, A., Kelly, D., & Azzopardi, L. (2016, March). Impacts of time constraints and system delays on user experience. In *Proceedings of the 2016 acm on conference on human information interaction and retrieval* (pp. 141-150).
9. Dhanagari, M. R. (2024). MongoDB and data consistency: Bridging the gap between performance and reliability. *Journal of Computer Science and Technology Studies, 6*(2), 183-198. https://doi.org/10.32996/jcsts.2024.6.2.21
10. Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., ... & Stoica, I. (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, *28*(13), 2009.
11. Fumero, J. J., Remmelg, T., Steuwer, M., & Dubach, C. (2015). Runtime code generation and data management for heterogeneous computing in java. In *Proceedings of the principles and practices of programming on the java platform* (pp. 16-26).
12. Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2024). Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys*, *57*(3), 1-36.
13. Heydari Beni, E. (2021). Deployment Efficiency and Data Security for the Cloud.
14. Jandl, A. (2020). *IoT and edge computing technologies for vertical farming from seed to harvesting* (Doctoral dissertation, Wien).

15. Jani, Y. (2020). Spring boot for microservices: Patterns, challenges, and best practices. *European Journal of Advances in Engineering and Technology*, *7*(7), 73-78.

16. Joosen, A., Hassan, A., Asenov, M., Singh, R., Darlow, L., Wang, J., & Barker, A. (2023, October). How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (pp. 443-458).

17. Khatiwada, P., & Dhakal, P. (2024). Evaluating Serverless Machine Learning Performance on Google Cloud Run. *arXiv preprint arXiv:2406.16250*.

18. Konneru, N. M. K. (2021). Integrating security into CI/CD pipelines: A DevSecOps approach with SAST, DAST, and SCA tools. *International Journal of Science and Research Archive*. https://ijsra.net/content/role-notification-scheduling-improving-patient

19. Kumar, A. (2019). The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. *International Journal of Computational Engineering and Management, 6*(6), 118-142. https://ijcem.in/wp-content/uploads/THE-CONVERGENCE-OF-PREDICTIVE-ANALYTICS-IN-DRIVING-BUSINESS-INTELLIGENCE-AND-ENHANCING-DEVOPS-EFFICIENCY.pdf

20. Larsson, M. (2023). *Microservices with Spring Boot 3 and Spring Cloud: Build Resilient and Scalable Microservices Using Spring Cloud, Istio, and Kubernetes*. Packt Publishing Ltd.

21. Lin, C., Nadi, S., & Khazaei, H. (2020, September). A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 371-381). IEEE.

22. Manner, J., Endreß, M., Heckel, T., & Wirtz, G. (2018, December). Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (pp. 181-188). IEEE.

23. Mouat, A. (2015). *Using Docker: Developing and deploying software with containers*. " O'Reilly Media, Inc.".

24. Nyati, S. (2018). Revolutionizing LTL carrier operations: A comprehensive analysis of an algorithm-driven pickup and delivery dispatching solution. *International Journal of Science and Research (IJSR), 7*(2), 1659-1666. https://www.ijsr.net/getabstract.php?paperid=SR24203183637

25. Pérez, A., Risco, S., Naranjo, D. M., Caballer, M., & Moltó, G. (2019, July). On-premises serverless computing for event-driven data processing applications. In *2019 IEEE 12th International conference on cloud computing (CLOUD)* (pp. 414-421). IEEE

26. Reddy, K. S. P. (2017). *Beginning Spring Boot 2: Applications and microservices with the Spring framework*. Apress.

27. Sadakath, M. S. (2018). *Spring Boot 2.0 Projects: Build production-grade reactive applications and microservices with Spring Boot*. Packt Publishing Ltd.

28. Sardana, J. (2022). Scalable systems for healthcare communication: A design perspective. *International Journal of Science and Research Archive*. https://doi.org/10.30574/ijsra.2022.7.2.0253

29. Singh, V. (2022). Intelligent traffic systems with reinforcement learning: Using reinforcement learning to optimize traffic flow and reduce congestion. *International Journal of Research in Information Technology and Computing*. https://romanpub.com/ijaetv4-1-2022.php

30. Singh, V. (2022). Multimodal deep learning: Integrating text, vision, and sensor data: Developing models that can process and understand multiple data modalities simultaneously. *International Journal of Research in Information Technology and Computing*. https://romanpub.com/ijaetv4-1-2022.php

31. Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., ... & Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, *11*, 233-247.

32. Wen, J., Chen, Z., Liu, Y., Lou, Y., Ma, Y., Huang, G., ... & Liu, X. (2021, August). An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 416-428).
33. Wu, S., Mao, B., Lin, Y., & Jiang, H. (2017). Improving performance for flash-based storage systems through GC-aware cache management. *IEEE Transactions on Parallel and Distributed Systems*, *28*(10), 2852-2865.
34. Wyld, D. C. (2009). *Moving to the cloud: An introduction to cloud computing in government*. IBM Center for the Business of Government.
35. Zhang, S., Zhang, S., Chen, X., & Wu, S. (2010, January). Analysis and research of cloud computing system instance. In *2010 second international conference on future networks* (pp. 88-92). IEEE.