# Serverless Function for Prime Number Management: Implementation and Performance Analysis

1st João Martins
*Informatics Department*
*UBI*
Covilhã, Portugal
baltazar.martins@ubi.pt

2nd Vasco Senra
*Computer Science Department*
*UBI*
Covilhã, Portugal

***Abstract*—This paper presents the implementation of a serverless function that generates prime numbers less than 1000. The function is deployed on a cloud computing platform, Firebase Cloud Functions [1]. The performance of the function is evaluated in terms of latency and cold starts, both in single and concurrent usage scenarios. The report includes details about the function's implementation, deployment, and performance analysis. The choice of the platform, the implementation, and the deployment of the function are discussed, along with an analysis of the performance results. The paper concludes with a summary of the findings and potential future work.**

## I. Introduction

The advent of serverless computing has revolutionized the way applications are developed and deployed. By abstracting away server management, it allows developers to focus on the core functionality of their applications, leading to increased productivity and reduced time to market. One of the key features of serverless computing is the use of functions as a service (FaaS), where individual functions are deployed and executed in the cloud.

In this context, this paper presents the implementation of a serverless function that generates prime numbers less than 1000. The function is implemented on a serverless platform chosen by the students, such as IBM Cloud Functions, Firebase Cloud Functions, Supabase Edge Functions, or Azure Functions. The choice of the platform is based on various factors, including ease of use, scalability, and cost-effectiveness.

After the function is implemented and deployed, it is executed to measure and evaluate its latency and cold starts, both in single and concurrent usage scenarios. The results of this evaluation provide valuable insights into the performance characteristics of serverless functions.

The remainder of this paper is organized as follows: Section II provides an overview of serverless computing and functions as a service. Section III discusses various platforms for serverless computing. Section IV details the choice of the platform, the implementation of the function, and its deployment. Section V presents a detailed analysis of the function's performance. Finally, Section VI concludes the paper and suggests future work.

This report aims to provide a comprehensive understanding of serverless computing, the implementation of serverless functions, and their performance characteristics. It is hoped that the findings of this study will contribute to the growing body of knowledge in this emerging field.

## II. Serverless Computing and Functions as a Service

Serverless computing, as the name suggests, is a computing paradigm where the management of servers is abstracted away from the application developer. This does not mean that there are no servers involved; rather, the responsibility of server management is shifted from the developer to the cloud service provider. This includes tasks such as server configuration, capacity planning, patching, and scaling, which are traditionally handled by system administrators.

In serverless computing, the focus is on the services - compute, storage, database, messaging, API gateways, and more - where configuration, management, and billing of servers are invisible to the end user. This allows developers to concentrate on writing code and delivering value to the business, rather than worrying about infrastructure management. The billing model in serverless computing is also different, as users are charged based on the actual consumption of resources, rather than pre-allocated capacity.

A key component of serverless computing is Functions as a Service (FaaS). In a FaaS model, developers write and deploy individual functions, which are executed in response to events such as HTTP requests, database changes, or messages from a queue. Each function is a self-contained piece of business logic that performs a specific task. These functions can be written in various programming languages, depending on the support provided by the cloud service provider.

FaaS brings several benefits to application development. It promotes the microservices architecture, as each function can be developed, deployed, and scaled independently. It also enables event-driven programming, where functions are executed in response to events, leading to efficient resource

utilization. Moreover, since functions are stateless, they can be easily scaled up or down to match the demand.

However, FaaS also has its challenges. Functions are ephemeral, meaning they are short-lived and do not maintain state between invocations. This can make it difficult to handle long-running tasks or workflows that require maintaining state. Additionally, functions can experience ”cold starts”, which is the delay in function execution when it is invoked after being idle for some time.

Despite these challenges, serverless computing and FaaS have gained significant popularity due to their benefits. They have become a key part of modern application development and deployment strategies, enabling developers to deliver applications faster and more efficiently.

## III. Platforms for Serverless Computing

There are several platforms available today that offer serverless computing capabilities. These platforms provide the infrastructure and services necessary to build and deploy serverless applications. Here, we discuss a few popular ones:

- **AWS Lambda** [2]: Amazon Web Services (AWS) Lambda is one of the most widely used serverless computing platforms. It allows developers to run their code without provisioning or managing servers. AWS Lambda supports several programming languages, including Node.js, Python, Java, and C#. It automatically scales applications in response to incoming traffic and only charges for the compute time consumed.
- **Google Cloud Functions** [3]: Google Cloud Functions is Google's serverless execution environment. It allows developers to build and deploy applications that respond to events without the need to manage a server. Google Cloud Functions supports Node.js, Python, and Go, and it integrates well with other Google Cloud services.
- **Azure Functions** [4]: Azure Functions is Microsoft's serverless computing service. It allows developers to write code that responds to events, without the need to manage infrastructure. Azure Functions supports a wide range of programming languages, including C#, Java, JavaScript, TypeScript, and Python. It also provides seamless integration with other Azure services.
- **IBM Cloud Functions** [5]: IBM Cloud Functions is IBM's Function as a Service (FaaS) platform. It is based on Apache OpenWhisk, an open-source serverless computing platform. IBM Cloud Functions supports several programming languages, including JavaScript, Python, Swift, and PHP.
- **Cloud Functions for Firebase** [1]: Firebase, a Google subsidiary, offers Cloud Functions for Firebase. This platform allows developers to run backend code that responds to Firebase and HTTPS events. It is particularly useful for mobile and web application development.
- **Vercel** [6]: Vercel is a cloud platform for static sites and Serverless Functions that fits perfectly with modern frameworks like Next.js, Nuxt, Angular, Vue, and more.
- **Netlify Functions** [7]: Netlify Functions is a feature of Netlify's platform for building, deploying, and managing modern web projects. It allows developers to deploy serverless Lambda functions without an AWS account, and with function management handled directly within Netlify.

Each of these platforms has its strengths and weaknesses, and the choice between them depends on various factors, such as the specific requirements of the application, the preferred programming language, the required integrations with other services, and the cost.

## IV. Choice of Platform, Implementation and Deployment of the Function

### A. Choice of Platform

After careful consideration and evaluation of various serverless computing platforms, we decided to choose Firebase Cloud Functions for the implementation of our serverless function. Firebase Cloud Functions is a robust and versatile platform that allows developers to run backend code in response to events triggered by Firebase features and HTTPS requests.

The choice of Firebase Cloud Functions was influenced by several factors:

- **Ease of Use and Intuitive Documentation**: Firebase Cloud Functions provides a straightforward and developer-friendly environment. It allows for easy deployment of functions, and its integration with other Firebase services simplifies the development process. The platform's intuitive documentation and tutorials were particularly helpful, making the learning curve much smoother.
- **Language Support**: Firebase Cloud Functions supports JavaScript and TypeScript, popular and widely used languages, which makes it an attractive choice for many developers.
- **Scalability**: Firebase Cloud Functions automatically scales up to meet the demands of incoming traffic. This auto-scaling feature ensures that our function can handle multiple requests concurrently without any additional configuration.
- **Integration**: Firebase provides a suite of tools and services that are well-integrated with Cloud Functions. This includes Firebase Authentication, Firestore, and Firebase Hosting, among others. These services can be easily used in conjunction with Cloud Functions to build comprehensive applications.
- **Cost-Effectiveness**: With the Blaze plan, we were able to deploy our function at zero cost, making it an economical choice for our project. The pay-as-you-go pricing model of the Blaze plan ensures that we only pay for what we use.
- **Professional Advice**: The positive feedback and recommendations about Firebase Cloud Functions from our professor also played a significant role in our decision. Their

insights, based on extensive experience and knowledge, were invaluable in guiding our decision-making process.

In the following sections, we will detail the implementation of our prime number generation function and its deployment on Firebase Cloud Functions. We will also discuss the performance analysis of the function in terms of latency and cold starts.

### B. Implementation and Deployment of the Function

The implementation and deployment of our serverless function was carried out on Firebase Cloud Functions and the entire process is documented on our GitHub repository: Serverless-FunctionGenPrime [8].

The function was implemented in JavaScript, a language supported by Firebase Cloud Functions. Our function was designed to generate prime numbers, a task that requires careful algorithmic design to ensure efficiency. The code for the function can be found in the `index.js` file in the `functions` directory of the repository.

After writing and testing the function locally, we proceeded to deploy it on Firebase Cloud Functions. The deployment process was straightforward, thanks to Firebase's intuitive command-line interface and clear documentation. We used the Firebase CLI (Command Line Interface) to initialize a new project, link it to our Firebase account, and deploy our function.

The command `firebase init functions` was used to set up a new Firebase Cloud Functions project. This command creates a new directory in our project with all the necessary files and dependencies for our function.

After our function was in place, we used the command `firebase deploy --only functions` to deploy our function to Firebase Cloud Functions. This command uploads our function to the Firebase servers, where it is ready to be triggered by incoming requests.

Once deployed, our function was accessible through a unique URL provided by Firebase. The URL for our deployed function is: https://us-central1-testecnultima.cloudfunctions.net/generatePrimes

In the next section, we will discuss the performance analysis of our function, focusing on aspects such as latency and cold starts.

## V. Performance Analysis

In this section of our report, we will delve into the detailed examination of our serverless function's performance. This analysis is crucial to understand the efficiency and effectiveness of our function in a real-world scenario.

The performance analysis will be divided into two main subsections: "Latency and Latency in Concurrent Use" V-A and "Cold Starts" V-B.

In the first subsection mentioned, we will measure the time taken by our function to execute a request and return a response. We will also evaluate how the function performs under concurrent use, i.e., when multiple requests are made simultaneously. This will provide insights into the scalability of our function and its ability to handle high traffic.

The second subsection mentioned will focus on the time taken by our function to start executing after being idle for a certain period. Cold start is a common phenomenon in serverless computing, where the function takes longer to respond after a period of inactivity. Understanding the cold start behavior of our function will help us optimize its performance and improve user experience.

Through this performance analysis, we aim to identify potential bottlenecks and areas of improvement in our serverless function. The findings from this section will guide our future work in enhancing the function's performance and efficiency.

### A. Latency and Latency in Concurrent Use

We utilize Apache JMeter, a powerful open-source software tool designed for load testing and measuring performance. JMeter simulates multiple users sending requests to a target server and provides comprehensive results that allow us to analyze various performance metrics, including latency.

Apache JMeter is a Java application that operates in a multitude of protocols including HTTP, HTTPS, JDBC, FTP, JMS, and LDAP. It is designed to simulate a heavy load on a server, network, or object to test its strength or analyze overall performance under different load types.

To analyze latency and concurrent latency with JMeter, we follow these steps:

1) **Create a Test Plan**: This is the starting point of any JMeter test. It describes a series of steps JMeter will execute when run.
2) **Add a Thread Group**: This simulates multiple users making requests to our server. We can specify the number of users and the ramp-up period to simulate different levels of concurrent load.
3) **Add an HTTP Request Sampler**: This component allows us to specify the details of the requests we want to send to our server.
4) **Add Listeners**: These components provide the output of the test. For analyzing latency, we use listeners such as *View Results in Table* and *Summary Report*. These listeners provide detailed information about the latency of our requests.
5) **Run the Test**: We execute the test and collect the results.
6) **Analyze the Results**: After the test, we analyze the results provided by the listeners. The *Latency* column in the *View Results in Table* listener shows the latency for each individual request. The *Summary Report* listener provides statistics about the latency across all requests.

In JMeter, "latency" is defined as the time from when a request was sent until the first response was received. This includes the time taken to send the request and the time the server took to process it and start responding. It does not include the time taken to receive the entire response.

*1) Latency - Results:* The results, exposed in 1, indicate the performance of an HTTP request with a **single user** sending

8 requests. Here's an analysis of the results with a focus on latency:

- **Label:** The name of the HTTP request sampler in your JMeter test is "HTTP Request".
- **# Samples:** The total number of HTTP requests sent during the test is 8.
- **Average:** The average latency, or the time taken to receive a response after a request is made, is 573 milliseconds. This is a moderate latency, but depending on the nature of the application, it may or may not be acceptable.
- **Min:** The minimum latency observed is 235 milliseconds, which is the best case scenario among the 8 requests.
- **Max:** The maximum latency observed is 2586 milliseconds. This is the worst case scenario among the 8 requests and is quite high. This could potentially be a point of concern as it might indicate some requests are taking a very long time to get a response.
- **Std. Dev.:** The standard deviation is 762.98 milliseconds. This is a measure of variation in the latency. A high standard deviation indicates that the response times are spread out over a large range. In this case, the standard deviation is quite high, indicating a significant variation in latency among the 8 requests.
- **Error %:** The error percentage is 0.000%, which means no errors were encountered during these 8 requests. This is a good sign indicating that the server is handling the requests successfully.
- **Throughput:** The throughput is 0.07907 requests/second. This indicates the rate at which the server is processing requests. With a single user, this value is not as significant, but it's a useful metric when testing with multiple users to understand the server's capacity.
- **Received KB/sec:** This is the rate at which the client received data from the server, in this case, 0.07 KB/sec.
- **Sent KB/sec:** This is the rate at which the client sent data to the server, in this case, 0.02 KB/sec.
- **Avg. Bytes:** The average size of the server's response is 957.0 bytes.

In summary, with a single user, the server handled the requests with an average latency of 573 ms. However, the maximum latency was quite high at 2586 ms, and there was a significant variation in latency as indicated by the high standard deviation.

This conclusions can guide towards potential areas for performance improvements.

1) **Reduce High Latency:** The maximum latency is quite high at 2586 ms. Investigating why some requests take this long is a good start. This could be due to various reasons such as inefficient code, slow network connections, or server configuration issues.
2) **Address Variability in Latency:** The high standard deviation in latency indicates that the response time is inconsistent. This could lead to a poor user experience as users could occasionally experience slow responses. Identifying the cause of this variability could help improve the consistency of the application's performance.
3) **Optimize Data Transfer:** The average size of the server's response is 957.0 bytes. If the application deals with large amounts of data, you might want to look into ways to reduce the size of the data being transferred. This could include techniques like data compression or removing unnecessary data.
4) **Improve Throughput:** The throughput is 0.07907 requests/second. While this might be acceptable for a single user, it could become a bottleneck when multiple users are making requests concurrently. Optimizing the code or increasing the server's resources could help improve the throughput.

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/... | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP Request | 8 | 573 | 235 | 2586 | 762.98 | 0.00% | 4.7/min | 0.07 | 0.02 | 957.0 |
| TOTAL | 8 | 573 | 235 | 2586 | 762.98 | 0.00% | 4.7/min | 0.07 | 0.02 | 957.0 |

Fig. 1. Results for Latency

*2) Latency in Concurrent Use - Results:* The results, exposed in 2, indicate the performance of an HTTP request with **100 concurrent users** sending a total of 108 requests. Here's an analysis of the results with a focus on latency:

- **# Samples:** The total number of HTTP requests sent during the test is 108.
- **Average:** The average latency, or the time taken to receive a response after a request is made, is 4148 milliseconds. This is quite high, especially considering that this is an average value. High latency could lead to a poor user experience.
- **Min:** The minimum latency observed is 235 milliseconds, which is the best case scenario among the 108 requests.
- **Max:** The maximum latency observed is 6551 milliseconds. This is the worst case scenario among the 108 requests and is extremely high. This could potentially be a point of concern as it might indicate some requests are taking a very long time to get a response.
- **Std. Dev.:** The standard deviation is 1415.59 milliseconds. This is a measure of variation in the latency. A high standard deviation indicates that the response times are spread out over a large range. In this case, the standard deviation is quite high, indicating a significant variation in latency among the 108 requests.
- **Error %:** The error percentage is 0.000%, which means no errors were encountered during these 108 requests. This is a good sign indicating that the server is handling the requests successfully, even under high load.
- **Throughput:** The throughput is 0.10058 requests/second. This indicates the rate at which the server is processing requests. With 100 concurrent users, this value is quite low, suggesting that the server might be struggling to handle this level of concurrency.
- **Received KB/sec:** This is the rate at which the client received data from the server, in this case, 0.09 KB/sec.
- **Sent KB/sec:** This is the rate at which the client sent data to the server, in this case, 0.02 KB/sec.

- **Avg. Bytes:** The average size of the server's response is 955.7 bytes.

Most important conclusions:

- With 100 concurrent users, the server handled the requests with an average latency of 4148 ms.
- The maximum latency was extremely high at 6551 ms.
- There was significant variation in latency as indicated by the high standard deviation.
- The throughput was quite low, suggesting that the server might be struggling to handle this level of concurrency.

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/... | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP Request | 8 | 573 | 235 | 2586 | 762.98 | 0.00% | 4.7/min | 0.07 | 0.02 | 957.0 |
| TOTAL | 8 | 573 | 235 | 2586 | 762.98 | 0.00% | 4.7/min | 0.07 | 0.02 | 957.0 |

Fig. 2. Results for Latency

### B. Cold Starts

In this section, we focus on the "cold start" phenomenon. A cold start in serverless computing refers to the situation where a function needs to be initialized from scratch, which can lead to increased latency for that particular invocation.

To analyze cold starts, we utilize Google Cloud's Stackdriver Trace, a distributed tracing system that collects latency data from your applications and displays it in near real-time in the Google Cloud Console.

Here are the steps to enable Stackdriver Trace and analyze cold starts:

1) **Enable Cloud Trace API**: The first step is to enable the Cloud Trace API in the Google Cloud Console. This API allows us to send and retrieve trace data from Stackdriver Trace.
2) **Install Stackdriver Trace Agent**: Next, we install the Stackdriver Trace Agent in our function's codebase using npm, the Node.js package manager. The command for this is `npm install --save @google-cloud/trace-agent`.
3) **Start Stackdriver Trace Agent in Function Code**: We then start the Trace Agent at the beginning of our function's code with the line `require("@google-cloud/trace-agent").start();`. This allows the agent to collect trace data from our function.
4) **Deploy Function**: We deploy our function to Firebase with the command `firebase deploy --only functions`. This makes our function available to be invoked.
5) **View Function's Trace in Stackdriver**: Finally, we can view the trace data for our function in the Stackdriver section of the Google Cloud Console. This data includes information about the latency of our function, which can help us identify and analyze cold starts.

The following graph 3 represents the relationship between latency (green) and the number of calls (blue). We left it on and didn't make calls for a few minutes, after this period we made requests and noticed that the first calls took longer and then returned to normal operation after a few calls. This pattern was repeated throughout the various tests carried out during this experiment.

Fig. 3. Results for Cold Starts

Cold starts affect cloud performance because they involve the initialization of serverless functions or virtual machines (VMs) from scratch. This process includes allocating resources, loading code, and establishing runtime environments, which can take several seconds. During this time, applications experience increased latency, leading to slower response times for users.

To minimize the impact of cold starts, developers can use strategies such as keeping functions warm by periodically invoking them, optimizing initialization code to reduce setup time, and using provisioned concurrency for serverless functions. These approaches help ensure that resources are readily available, reducing the delay associated with cold starts and improving overall application performance.

## References

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

## References


[1] Firebase, "Cloud Functions for Firebase," 2022. [Online]. Available: https://firebase.google.com/docs/functions?hl=pt. [Accessed: 2024-05-15].

[2] AWS, "AWS Lambda," 2022. [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 2024-05-15].

[3] Google Cloud, "Google Cloud Functions," 2022. [Online]. Available: https://cloud.google.com/functions. [Accessed: 2024-05-15].

[4] Microsoft Azure, ”Azure Functions,” 2022. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Accessed: yyyy-mm-dd].
[5] IBM, ”IBM Cloud Functions,” 2022. [Online]. Available: https://www.ibm.com/cloud/functions. [Accessed: 2024-05-15].
[6] Vercel, ”Vercel,” 2022. [Online]. Available: https://vercel.com/. [Accessed: 2024-05-15].
[7] Netlify, ”Netlify Functions,” 2022. [Online]. Available: https://www.netlify.com/products/functions/. [Accessed: 2024-05-15].
[8] J. Martins, ”ServerlessFunctionGenPrime,” 2024. [Online]. Available: https://github.com/jmbmartins/ServerlessFunctionGenPrime. [Accessed: 2024-05-15].