**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

**Department of Electronics, Telecommunications and Computers Engineering**


# Monitoring Resources in Function-as-a-Service Platforms


**Beatriz Bila Veiga Guerreiro**


Bachelor's degree

Dissertation to obtain the Master's Degree
in Computer Science and Engineering


Advisors : Prof. Filipe Bastos de Freitas
Prof. José Manuel de Campos Lages Garcia Simão


Jury:

President: Prof. Carlos Jorge de Sousa Gonçalves

Referees: Prof. Manuel Martins Barata
Prof. Filipe Bastos de Freitas

**December, 2023**

**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

**Department of Electronics, Telecommunications and Computers Engineering**


# Monitoring Resources in Function-as-a-Service Platforms


**Beatriz Bila Veiga Guerreiro**


Bachelor's degree

Dissertation to obtain the Master's Degree
in Computer Science and Engineering


Advisors : Prof. Filipe Bastos de Freitas
Prof. José Manuel de Campos Lages Garcia Simão


Jury:

President: Prof. Carlos Jorge de Sousa Gonçalves

Referees: Prof. Manuel Martins Barata
Prof. Filipe Bastos de Freitas

**December, 2023**

# Abstract

Functions deployed in Function-as-Service (FaaS) platforms need to be monitored regarding resource consumption, errors, and application-specific metrics. Platforms like Google Cloud Functions or Azure Functions have dashboards and Web APIs that expose information about the execution of functions. However, the current approach collects data of general metrics about the function as an all and imposes a vendor-specific way for monitoring events. To the best of our knowledge, fine-grained function metrics are not available in FaaS platforms, e.g., the time that takes to execute only part of the function's code.

This work aims to explore how to build a system to provide fine-grained monitoring to FaaS platforms for developers presenting an architecture for this system, the metrics library developed for FaaS platforms, the evaluation of the proposed solution performance and the discussion of some challenges. In the obtained results it is possible to analyse the costs when obtaining metrics with such a high level of detail.

**Keywords:** FaaS; Metrics; Cloud; Monitoring; Fine-grained monitoring.

# Resumo

As funções publicadas nas plataformas de FaaS devem ser monitorizadas tendo em conta o consumo de recursos, os erros e outras métricas mais específicas da aplicação. Plataformas como a Google Cloud Functions ou a Azure Functions têm os seus próprios painéis gráficos e as suas próprias interfaces Web que expõem informação sobre a execução de funções. No entanto, a abordagem atual coleciona dados de métricas genéricas sobre as funções como um todo e impõe dependências das plataformas para realizar a monitorização. É do nosso conhecimento, que não estão disponíveis nas plataformas de FaaS, métricas com uma granularidade muito específica, como por exemplo, o tempo que demora a executar apenas uma parte do código da função.

Este trabalho tem como objetivo explorar-se a construção de um sistema que disponibiliza, em plataformas FaaS, a monitorização com uma granularidade mais específica. Serão apresentados a arquitetura deste sistema, a biblioteca de métricas para plataformas FaaS que foi desenvolvida, a avaliação da performance da solução proposta e a discussão de alguns desafios. Nos resultados obtidos é possível analisar os custos ao obter métricas com este nível detalhe mais específico.

**Palavras-chave:** FaaS; Métricas; Nuvem; Monitorização; Monitorização específica.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**API** Application Programming Interface. 2, 6, 7, 9, 11, 19, 28, 33, 34, 41, 42

**AWS** Amazon Web Services. 7

**CLI** Command-line Interface. 8, 19, 41, 42, 45

**CPU** Central Processing Unit. 7, 13, 14, 20, 21, 42

**CRUD** Create, Read, Update and Delete. 35

**DTO** Data Transfer Object. 35

**ER** Entity Relationship. 20

**FaaS** Function-as-a-Service. 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, 18, 19, 24, 25, 27, 28, 40, 51, 52, 53

**FMS** Flexible Monitoring Solution. 10, 11, 14

**GB** Gigabyte. 45

**GCP** Google Cloud Platform. 44, 45, 46

**GUI** Graphical User Interface. 19, 41, 45, 46

**HTTP** Hypertext Transfer Protocol. 7, 8, 9, 12, 19, 20, 28, 29, 31, 32, 41, 42, 45, 48, 53

**I/O** Input/Output. 12

**JAR** Java Archive. 9, 27, 41
**JDBC** Java Database Connectivity. 19
**JMX** Java Management Extensions. 22
**JSON** JavaScript Object Notation. 12, 29, 32
**JVM** Java Virtual Machine. 8, 22

**KB** Kilobyte. 44

**MB** Megabyte. 44, 45, 46
**MiB** Mebibyte. 47, 48

**NATS** Neural Autonomic Transport System. 7

**RAM** Random-access Memory. 7, 8
**RDT** Resource Director Technology. 12
**REST** Representational State Transfer. 34

**SNMP** Simple Network Management Protocol. 10
**SQL** Structured Query Language. 35
**SQS** Simple Queue Service. 7

**URL** Uniform Resource Locator. 30, 33

**YAML** Yet Another Markup Language. 41, 42

# 1

# Introduction

This chapter presents the domain studied in this work, which is related to monitoring in Function-as-a-Service (FaaS) environments. It begins by presenting the context and the motivation for this work, then introduces the problem of major cloud platforms, followed by a brief description of the objective of this work and how we will contribute with a new approach to FaaS monitoring through the implementation of a library. Then, the research paper published about this study is presented as a way to enhance this work. Finally, the structure of this document is provided.

## 1.1 Context

In recent years, Serverless Computing [60] has become a popular method for deploying, managing, and monitoring services and applications, primarily in the cloud. This technology replaces traditional monolithic architectures with containerized environments, freeing developers from the need to manage operational logic and application infrastructure. This allows for increased productivity and a focus on business strategies. With Serverless Computing, the computational resources used are those that the application needs, as the infrastructure adapts to the demands of the application. The event-driven nature of Serverless Computing, combined with its fine-grained payment system, reduces costs for clients and allows more efficient infrastructure management for providers.

Function-as-a-Service [1], is a specific type of Serverless Computing that allows developers to deploy individual functions, or small units of code. FaaS functions are executed only in response to specific events, such as HTTP requests or file uploads, and are executed for only the duration of the request. FaaS offers good reasons to exist within the already rich cloud service landscape. It provides high-level abstractions of distributed computing elements, reducing the need for users to be experts in distributed systems, or to manage complex microservice-based architectures themselves. [45]

Monitoring is a critical aspect of Serverless Computing, as it provides insight into the performance and behavior of Serverless applications. Monitoring enables organizations to optimize their Serverless deployments, reduce costs, improve performance, and increase scalability. With the rise of Serverless Computing, there has been a growth in the availability of tools and platforms that make monitoring Serverless applications more efficient.

## 1.2 Problem

Google Cloud Functions [18] and Azure Functions [5] are two of the major FaaS Platforms used in the cloud, they provide dashboards for developers to monitor the execution of the functions, e.g., execution time or the number of times the function was called. They also provide Web APIs for developers to obtain this information. However, all the available information is only about the complete execution of the function, and there is no way to obtain information about the time consumed by a certain part of the function when using the existing APIs from those cloud providers.

Functions fine-grained monitoring can be hard due to this limitation and can consume a lot of development time. One way to deal with this issue is to use a log-based approach where developers log messages during the function execution in the FaaS platform and periodically process the logs. However, this solution requires all the logs to be stored in the FaaS platform and needs post-processing from the developer.

## 1.3 Proposal

In this work, was created a system that gives the developer the tools to be able to instrument function execution and collect information from monitoring tools, for example, Prometheus [48]. To achieve this, a metrics module to work in FaaS platforms

and a service to process and store the generated data, were developed. The module will allow developers to register what they need, e.g., the time it takes to execute only part of the function or count the number of times a condition occurs. Initially, the registered data is going to be stored locally, only during the execution of the function, then it is pushed to the service that processes and stores it, e.g., increments the number of times a condition occurs and then stores that value. The service also provides an endpoint for monitoring tools. This solution allows us to: (1) have a platform-independent system; (2) minimize the data stored in the FaaS platform; (3) post-data processing is simplified and (4) the developer can monitor the function execution using monitoring tools, like Prometheus.

## 1.4 Contributions

During the development of this dissertation, and leveraging the research conducted for the analysis and preparation of the foundations and tools that would allow the implementation of the proposed monitoring system, an article on the topic of this work was written and submitted to the "CISTI 2023 - 18th Iberian Conference on Information Systems and Technologies." This article, with the title "Monitoring in Function-as-a-Service Platforms" [2], was accepted and subsequently published and presented at the conference. This contribution, with a paper based on the research work conducted, was something that encouraged the continuation of this dissertation.

For this work was also created a GitHub repository [52] to keep publicly available all the code developed to implement and test the proposed solution.

## 1.5 Outline

This document is organized into 6 chapters as follows. The current chapter, which is Chapter 1, as we just verified, presents the context of what will be studied, exposes a summary of the problem and the contributions of this work, and presents the research paper that was published as a contribution to the study to the scientific and technological community. Chapter 2 presents related work within the domain of this dissertation, namely some related concepts and similar tools that may be necessary for the solution. In Chapter 2, still following the related work, some articles were explored where the same problem presented in this dissertation is considered. Chapter 3 presents the structure of the proposed solution, including there, the main architecture of the system, the metrics module and how to use it. In Chapter 4, the entire

implementation of this solution is covered, and we will see how metrics are collected, processed, and stored. After presenting and describing the solution and the implemented system, Chapter 5 describes the evaluation processes and presents the results and respective conclusions. Finally, in Chapter 6, conclusions are drawn from the entire work, and points to consider for continuing and improving this work in the future are indicated.

# 2

# Background and Related work

In this chapter, we start by providing a brief description of the concepts of Serverless Computing and Function-as-a-Service, with special emphasis on issues related to monitoring these types of systems. Then, we present FaaS platforms, including both open-source and proprietary platforms, how monitoring is done, and what metrics are available for each of them. To complement this, we compare some monitoring tools. Finally, we analyze some related work on this topic.

## 2.1 Serverless Computing

Serverless Computing is a set of various cloud-related technologies that can be summarized by the following principles: users do not have to worry about the infrastructure of systems and applications; users only pay for the resources that are used; and it is an event-driven model and that allows for elastic scaling of resources [1].

Lately, Serverless Computing has become a very chosen way for deploying, managing, and monitoring services and applications in the cloud. This technology replaces traditional monolithic architectures with containerized environments, such as Docker [12]. This way developers do not need to manage operational logic and application infrastructure. This increases productivity and the focus on business strategies.

As mentioned above, in a Serverless Computing model, the computational resources used on the platforms are only those that are necessary for the applications to work.

There is here the capacity of the resources to adapt to the software demands, such as during usage spikes or when the application is not being used at all. This mentioned capacity of adapting and the fact that this is an event-driven model allows us to only pay for reduced and strictly necessary costs in cloud providers.

Monitoring in Serverless Computing environments is also a very important aspect and it should be considered because that way we obtain better performance in the developed applications. Throughout this dissertation, we will see that monitoring is the principal focus of the study, and it can be said that taking this aspect into account makes organizations optimize their deployment processes and increase scalability. The rise of Serverless Computing technologies has also led to the increase of monitoring tools for these environments to make them more efficient.

## 2.2 Function-as-a-Service

Function-as-a-Service, or FaaS, is a specific type of Serverless Computing that allows developers to deploy individual functions, or small units of code, as opposed to larger applications. When choosing to use FaaS, the user does not need to have much knowledge in distributed systems or even how to implement microservice-based architectures because this mechanism exists also so that it is possible to have a high level abstraction for complex issues related to the inherent existence of distributed environments in the world of cloud technologies [1].

In Figure 2.1 are presented the layers of a generic FaaS platform. Each of them has its responsibilities. The main focus will be on the Function Management layer and the Resource Orchestration layer since they will be studied as a solution for monitoring the functions. Some FaaS providers have their APIs and integration with third-party monitoring tools to allow for more customized monitoring and analysis.

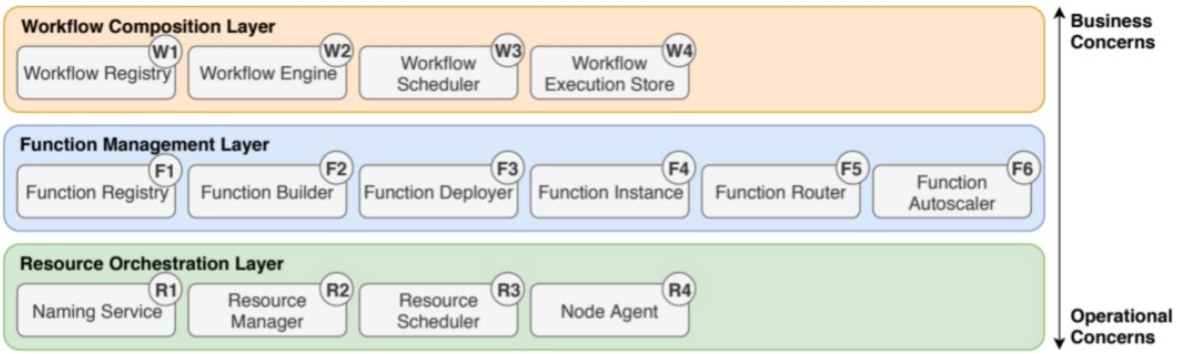


Figure 2.1: Layers of a generic FaaS platform. Adapted from [1]

.

## 2.3 FaaS Providers

Next, we will explore existing FaaS tools to understand their components and examine their monitoring capabilities. The goal was to determine if any of these tools allowed for the customization of metrics in some way.

OpenFaaS and OpenWhisk are two of the FaaS tools that will be studied and both of them are open-source. On the other hand, Google Cloud Functions and Microsoft Azure Functions are two of the most popular FaaS tools, but they are not open source. Monitoring in these platforms may involve using the cloud providers' storage and monitoring solutions.

### 2.3.1 OpenFaaS

OpenFaaS [38] is a container-based framework for building serverless functions and it integrates with Docker and Kubernetes [31] [13]. To create functions with OpenFaaS we have to use the existing templates provided by the tool for each language. This is also a tool that supports auto-scaling, and to invoke the functions it can be used event-triggers such as Apache Kafka [29], NATS [35], AWS SQS [3], and others [59].

Users can manage the function containers in OpenFaaS with its API Gateway component [39]. OpenFaaS also provides some support for monitoring, and through the endpoints made available by this OpenFaaS Gateway, it is possible, to not only manage the functions but also observe some of the metrics exposed by the tool through Prometheus (that will be presented forward in this chapter) [48] [59].

The Gateway uses the Prometheus Go client library [50] that defines and exposes internal metrics via HTTP endpoint. These metrics are scraped with a scrape interval defined and are not customizable. In the Table 2.1, we can observe which metrics are exposed for analysing the functions. CPU and RAM usage metrics are only made available in OpenFaaS Pro [40].

### 2.3.2 Apache OpenWhisk

Apache OpenWhisk [41] is an open-source similar to OpenFaaS. It is used to build and execute serverless functions and also uses Docker containers to manage the infrastructure.

Table 2.1: Metrics exposed by OpenFaaS API Gateway. Adapted from [40].

| | | |
|---|---|---|
| gateway_functions_seconds | histogram | Function invocation time taken |
| gateway_function_invocation_total | counter | Function invocation count |
| gateway_service_count | counter | Number of function replicas |
| gateway_service_ready_count | counter | Number of function replicas which are in a ready state |
| gateway_service_target | gauge | Target load for the function |
| gateway_service_min | gauge | Min number of function replicas |
| pod_cpu_usage_seconds_total | counter | FaaS seconds consumed by all the replicas of a given function |
| pod_memory_working_set_bytes | gauge | Bytes of RAM consumed by all the replicas of a given function |

In OpenWhisk functions are called Actions and can be written in several programming languages. Function invocations can be triggered in response to events from external sources, which the platform refers to as Feeds, or from HTTP requests. The platform also provides a Command Line Interface (CLI) to help manage functions and do operations like the deployment of the functions [41].

OpenWhisk has two types of metrics: system metrics, which contain information about the system itself, and metrics related to the events that execute the functions, which the platform calls user metrics [42].

Systems metrics can be registered in log files or sent and processed using Kamon [30] (Kamon Telemetry is a set of libraries for instrumenting applications running on the JVM).

On the other hand user metrics provide data about the performance of the functions. However, OpenWhisk needs another service to expose these metrics. That service is developed by Apache and is called OpenWhisk User Events [43]. It uses Kafka and Kamon to publish topics with metrics for other tools as Prometheus.

This process is being done by using the Prometheus JVM Client [51] and, like in OpenFaaS, it is exposed to an HTTP endpoint from where the Prometheus server will pull the metrics with a scrape interval.

OpenWhisk has execution metadata in the functions such as `duration`, `statusCode` or `memory` and the system can also produce the following events which they call metrics: `ConcurrentInvocations`, `ConcurrentRateLimit` and `TimedRateLimit` but, like in OpenFaaS, these events and metrics are not customizable.

### 2.3.3 Google Cloud Functions

Google Cloud Functions is a tool from Google Cloud Platform [22] that allows to creation of functions giving the abstraction to the developers of the servers and runtime environments [18].

To deploy a Google Cloud Function it is necessary, or to put the code of the function in a file in a format such as ZIP or JAR, or to create a Docker container image. These are two ways to allow that the deployment of the functions does not need to have specific characteristics of any programming language and that can include dependencies such as third-party libraries. However, although we can have this abstraction, the function will always be stored and managed by Google's internal tools. HTTP triggers or event triggers, such as messages from a Pub/Sub topic, can be used to run the function.

Google Cloud Functions metrics are available in Cloud Monitoring [19]. However, there is also the Google Cloud Managed Service for Prometheus [20] which is a Google Cloud's solution for Prometheus metrics. It collects the metrics from Prometheus exporters and the monitoring can be supplemented with Cloud Monitoring. Managed Service for Prometheus is built on top of Monarch [32], which is the database used for Google's monitoring, so it uses the same backend and APIs as Cloud Monitoring. The tool has some collectors to obtain the metrics data that scrape Google's implemented exporters and send them to Monarch. These collectors can be used in other clouds [20].

### 2.3.4 Microsoft Azure Functions

Microsoft Azure Functions [5] is the solution from Azure to run serverless functions in the cloud. The functions can be written in multiple languages and the framework also provides the infrastructure and resources needed.

Microsoft Azure Functions integrates with Azure Application Insights [4] which is an extension of Azure Monitor [6] to obtain performance monitoring from the functions execution.

Like in Google, there is also in Azure a Prometheus dedicated service, to manage Prometheus metrics. It is Azure Monitor managed service for Prometheus [7]. This is a service to collect metrics using Prometheus and it is a component of Azure Monitor Metrics.

## 2.4 Monitoring Tools

Next, will be explored some monitoring tools to determine which ones are better suited for systems involving FaaS. These tools have architectures that allow them to integrate with data from FaaS platforms and aggregate this data for presentation in graphs and dashboards. Prometheus, Nagios and Pandora FMS are examples of monitoring tools.

### 2.4.1 Prometheus

Prometheus is an open-source tool for monitoring and alerting and all the metrics information it collects includes always a timestamp. It can also store optional labels alongside with the metrics. This is a monitoring solution indicated for FaaS because it is a modern software which can be integrated easily and can act as middleware. It is usually integrated with Grafana [25] to display data, as the Prometheus user interface is quite basic [48].

Prometheus have Exporters [49] that expose metrics in a format that Prometheus can scrape, allowing it to collect metrics from different sources and systems since they can be written in several programming languages. Exporters act as a bridge between Prometheus and the systems being monitored. Prometheus supports a wide range of exporters and more can be found in the Prometheus Github [17] repository. Additionally, custom exporters can be written to support metrics from systems or applications that are not already supported.

### 2.4.2 Nagios

Nagios [34] is a reference in IT monitoring. It provides monitoring of infrastructure components such as applications, services, operating systems, network protocols, systems metrics, and network infrastructure. For that, it uses agents that are installed on servers and uses SNMP checks. Despite its ability to scale Nagios is a tool more related to network monitoring and not compatible with FaaS.

### 2.4.3 Pandora FMS

Pandora FMS [44] is a flexible monitoring framework that can monitor different types of systems and applications, including networks, servers, virtual environments, cloud infrastructures, and more. Pandora FMS can also monitor FaaS but only indirectly. The

results from the executions need to be in its database and then, with the Pandora FMS implemented agents, the monitoring is made. Furthermore, it will always depend on a specific cloud provider and the APIs it exposes.

## 2.5 Related Work

Monitoring the behaviour of a system, either by observing logs or measuring the use of fundamental resources is an essential part of the development, testing, and operation of any application. In the context of FaaS, this task can be more difficult because of the layers of abstraction introduced by the cloud provider. Researchers [53–56] have noted that additional capabilities should be provided to find bottlenecks, trace errors, and provide a better understanding of the function execution. Furthermore, as serverless functions run for short amounts of time, and there is the potential for many instances to be running, it can be hard to identify problems and bottlenecks.

This work relates to the challenges of observability and monitoring, including techniques used in benchmarking [9, 16, 58, 61], profiling [11, 14], and log analysis [57]. However, there is currently a lack of an infrastructure to allow fine-grained monitoring at the application level.

### 2.5.1 Troubleshooting through log messages

The probability of errors in FaaS can be bigger and resolving these errors can consume a lot of time. Researchers have presented a semi-automated troubleshooting process, evaluated then by a prototype named SeMoDe, to improve fault detection and resolution for Serverless functions through log messages [57]. This way the function quality is enhanced and the number of test cases could be more and better.

There are problems when logging in FaaS. Those problems are, for example, the addition of non-functional code to the function source code and scaling problems related to manual analysis of the logs if adequate tooling to extract the information is missing. Also, generating logs consumes resources, the execution time can increase and this can result in higher costs. Furthermore, it is an extra backend service to add to these costs because of the communication with the FaaS platform and the cloud database space that is necessary to save the logs. However, the authors state that this solution has advantages. They are, the log history persisted in a database, and knowing the order of the events that allow recreating bugs offline. Even so, there will be always a separation between the debug process and the FaaS provider.

Cloud functions are black boxes executed in an isolated environment. Given that, the authors argue that logging the input, context and output of cloud functions is sufficient to reproduce bugs a posteriori. This is done by adding appropriate logging statements inside the code to establish a custom data schema. A more generic solution without modifications to the business code is the use of interceptors or annotations. The parameters are logged in a standardized, machine-readable format like JSON [28].

With the metadata resultant, the developer can process the error resolution manually or it can trigger automated test generation. The analysis of the logs starts by grouping events that represent the same execution. This can be made by ordering the identifiers present in the metadata. Then the logs are filtered (for example, by searching for the word Exception). Finally, the input, context and output parameters are extracted to build the test skeletons and a template approach is chosen, where placeholders are replaced with the extracted data from the step before.

This approach of having log messages for function troubleshooting has something similar to what is exposed in the proposed solution chapter. It will also have the addition of execution statements inside the code. However, to monitor the functions, will not be necessary to save logs and consume extra space in a database since a Prometheus server will be used.

### 2.5.2 Profiling

FaaSProfiler [14] is a FaaS platform for testing and profiling that uses the Apache OpenWhisk. This platform's goal is to investigate and identify the architectural implications of FaaS serverless computing. Before this platform was implemented the focus was only to validate serverless systems from a black-box point of view.

With FaaSProfiler, the authors, are trying to analyse hardware performance and profile the functions getting metrics related to latency, execution time, wait time and also resource metrics. To use this tool we need to specify which parameters we want to profile and how we want the function invocations in a JSON script. The interaction with OpenWhisk is made via HTTP connections and the configuration scripts can also specify the runtime profiling tools to be used, for example, `perf` (Linux performance counters profiling), `pqos-msr` (Intel RDT Utility), and `blktrace` (Linux block I/O tracing).

It is intended with this tool to obtain a more fine-grained introspection and have a more flexible solution for FaaS functions profiling. But, despite FaaSProfiler doing functions profiling, it always uses tools related to the operating system, for example, `perf` from

Linux. The monitoring consists of running some scripts before and after the function execution. This is not as fine-grained as suggested in the solution from Chapter 3 since it is intended to have more control in the function context.

### 2.5.3 Benchmarking

The following four tools presented are primarily to evaluate the capacities of the FaaS providers using benchmarks to compare them. They collect the metrics made available by the providers or, like in FaaSProfiler, from the operating system, contrary to the solution in Chapter 3, since it is wanted to evaluate the function itself and not the infrastructure and it should be possible to have more insight into the functions and not only some examples to be tested later. Our approach is more fine-grained since we want to give the developer more control in the function context, as discussed in the following chapter.

ServerlessBench [58] is an open-source tool that provides a benchmark suite with some metrics for characterizing serverless platforms. Its test cases analyse, for example, communication efficiency and startup latency. The goal of this benchmark suite was to evaluate the most popular serverless computing platforms but the main problem of this tool is that it only tests scenarios with synchronous function invocations which is not typical for most of the platforms.

FunctionBench [16] is also a tool that provides a suite of benchmarks to evaluate some aspects of the FaaS model. According to the authors, most published articles are to evaluate very specific computer resources such as CPU, memory, disk and network and all the obtained results were to evaluate each of them individually. In FaaS realistic scenarios it is supposed to evaluate the behaviour of all of these characteristics together. Given this, the tool measures the performance of various resources together in the context of realistic serverless applications, using simple system calls.

Another research made to improve the serverless performance analysis was presented with the implementation of ServiBench [61]. Since there was a lack of existing approaches applied to distributed tracing and which do not consider asynchronous applications, the ServiBench tool gives a benchmarking suite that covers the gaps mentioned and that supports the following performance factors of FaaS functions: median latency, cold starts, tail latency, scalability and dynamic workloads.

At last, regarding benchmarking, it was designed and presented an application named BeFaaS [9]. This application is a benchmarking framework for FaaS platform evaluation. It is stated in this research work that this *"is extensible for new workload profiles*

*and new platforms, supports federated benchmark runs in which the benchmark application is distributed over multiple providers, and supports a fine-grained result analysis"*. For this application to run is necessary the FaaS function itself which also has integrated the benchmark application in the code, the profiling configuration and the deployment configuration that describes the function environment and the FaaS platform used.

BeFaaS seems to be easy to configure because it uses some automation and, for a better analysis of the functions, it does fine-grained monitoring by obtaining measurements, such as identifying cold starts or other request-level effects. It also can easily be extended with additional benchmarks or adapted for other FaaS platforms.

### 2.5.4 COSE

COSE [11] is a framework that uses Bayesian Optimization to find the optimal configuration for serverless functions and using these statistical learning techniques collect the samples itself and predict the cost and the execution time of a function. This way the framework can find the configuration parameters for running the functions.

According the research, software developers generally configure a small set of parameters considered simple, for their applications, e.g., function timeout, memory size, and cloud providers. This development way frees developers from underlying resource management. However, some may be willing to configure the fine-grained resource policies to effectively improve the overall quality of service of applications. The serverless platform also can optionally support some fine-grained configuration options about the underlying runtime information.

COSE has the advantage of finding itself the best configuration for the function to be executed in the FaaS provider reducing the costs. However, the performance analysis is based only on CPU and memory resource consumption.

## 2.6 Summary

To understand the meaning of Serverless FaaS, it was presented in this chapter the concept of Serverless Computing. Was introduced then, what is Function-as-a-Service and explored some existing FaaS tools like OpenFaaS, OpenWhisk, Google Cloud Functions and Azure Functions. Additionally, we explore monitoring tools that are well-suited for systems involving FaaS or not and explain why, like Prometheus, Nagios and Pandora FMS.

During the research work, several articles exploring the topic of monitoring in FaaS were discovered, as FaaS is a widely used approach. Among the tools and research presented, none appeared to provide monitoring with a specific context as the one presented in the solution proposed in the next chapter, which will introduce a generalized architecture.

# 3

# Proposed Solution

In this chapter, we are going to present a Generic Architecture for fine-grained monitoring, specifying the role of each module presented and how communication is established between them. Also, it will be discussed the type of metrics and how they are inserted into the solution. Additionally, will be shown how the developer will use the proposed library, describing some functionalities with code examples. At the end of this chapter, we see the platforms that we need to implement the solution and some challenges that were faced.

## 3.1 Overview

After analyzing some FaaS-related tools for function monitoring, namely FaaS providers and more generic monitoring tools, it was possible to verify that none of them obtains metrics within such a specific context for each function. The goal of this solution is to enable fine-grained monitoring in FaaS platforms, giving developers the ability to analyze the characteristics and performance of parts of the function code, with data registered through a monitoring system. This solution is not specific to each cloud provider or monitoring system, as communication between the tools should be loosely coupled but it will be necessary to use those tools for implementing the solution.

## 3.2 Monitoring System

A metric can be understood as a quantifiable value used to measure the performance or register specific behaviour of a system or environment. Metrics can be used for comparisons and quality measurements within a specific context, such as in the case of serverless environments in the cloud.

A monitoring system involves a set of tools and technologies that process and store performance data represented by the obtained metrics in the environment being analyzed. In these systems, there is a source of data, for example, in FaaS, the data source is the function. This data can be aggregated, as we will see further, where various types of metrics are defined. It can leverage historical data and can also be used to trigger notifications or alerts that inform developers when specific goals or thresholds are not met or when there are errors in the code. Metrics allow for a detailed analysis, enabling customization of environments to achieve defined objectives. The ultimate goal is to achieve optimal performance.

### 3.2.1 Generic Architecture

In the Figure 3.1, a generic architecture of the implemented solution is presented, which aims to demonstrate the monitoring of FaaS functions. This monitoring can be highly specific, down to a single line of code. In the image, we can see the components that make up the system: FaaS Platform, Service, and Monitoring Tools. A description of the functionality of each of these components is provided below.

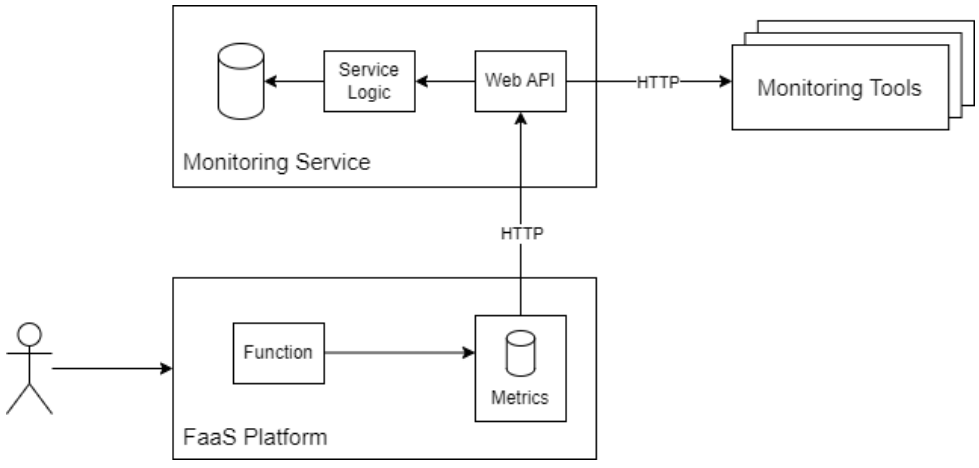


Figure 3.1: Generic Architecture of the proposed solution

### FaaS Platform

The FaaS Platform component has a Function module that represents a deployed function in a FaaS Platform which is responsible for executing and managing the life cycle of the function. For example, OpenFaaS can be used, which will allow the developer to create, delete, run, deploy, and also get a function or a list of functions. Communication with the FaaS platform can be done by the developer via HTTP, using the endpoints provided by the API of the platform or can be done using the CLI or Web GUI that could be also provided by the FaaS platform.

Then, there is the Metrics module that provides an interface to be used in the function to register information about the function execution and has the implementation with the calculation of the metrics. This module also contains a small data storage that is going to be used only during the execution of the function, when the function finishes the data is going to be pushed to the monitoring service via HTTP request. This module is deployed with the function in the deployed bundle, just like another library. Since this module is part of the function the communication between the two modules is made programmatically calling the Metrics module functionalities that are demonstrated below, in a section dedicated to detail the Metrics module.

### Monitoring Service

The Monitoring Service component includes the Web API module that exposes the endpoints for the Metrics module and the Monitoring Tools to be able to interact with the service.

It also includes the Service Logic module, which manages the received data. This module is responsible for validating, processing the data received, and then storing the result in a database, e.g., the metrics module pushes the duration time of execution of part of the function, the Logic module validates the value, it may store the value directly in the database or it also may use it to get metrics aggregations like calculate the average duration.

In addition to receiving the metrics and processing them, the Monitoring Service has also some functionalities to help manage the metrics in the functions, like, for example, calculating the identifier of the metrics. The communication between the Web API module and the Service Logic module is made programmatically since they belong to the same application. On the other hand, the connection with the database is more proper for example JDBC for Java applications.

**Monitoring Tools**

The Monitoring Tools component represents the tools that can interact with our service to provide a monitoring interface to the developer. For the connection with the Prometheus server, for example, a configuration file must be created that exposes an HTTP endpoint and several attributes such as the scrape interval. This configured Prometheus server, in turn, will pull the obtained and calculated metrics and store them, as the Prometheus server is a database of time series values. These metrics must be in the proper Prometheus format, meaning they must have a unique name associated with a value, if, for example, it is only a counter. It is intended to release some storage from the service after the data is consumed by the monitoring tool.

### 3.2.2 Metrics Definition

Various types of metrics can be obtained, and the categorization of metrics often relates to what the metric intends to characterize. Metrics can be broadly divided into categories like Execution Metrics and Resource Usage Metrics, among others. However, this characterization is inherently tied to the specific aspect of the function or system being measured. For example, we might want to calculate execution time or track the number of invocations of a particular action within a function, or we may want to collect metrics related to memory consumption, CPU usage, or network utilization.

Moreover, to collect metrics in a system and later perform aggregations and present them with suitable characteristics for analysis, metrics can take on various formats. For example, they can be viewed as counters, gauges, histograms, and other formats. A counter typically represents a value that can be incremented to measure occurrences or events, while a gauge represents instantaneous values that can fluctuate both up and down over time, such as memory usage measurements.

In the proposed solution, metrics are represented as instantaneous values throughout their processing in the various components. The Figure 3.2 illustrates that in this solution, with an Entity Relationship (ER) diagram of the Metric. A Metric is represented by an identifier, a name, an identifier of the environment/function where it is collected, the timestamp when it was obtained, and a numerical value quantifying that metric. This representation allows for capturing and organizing essential information about each metric for subsequent analysis and monitoring.

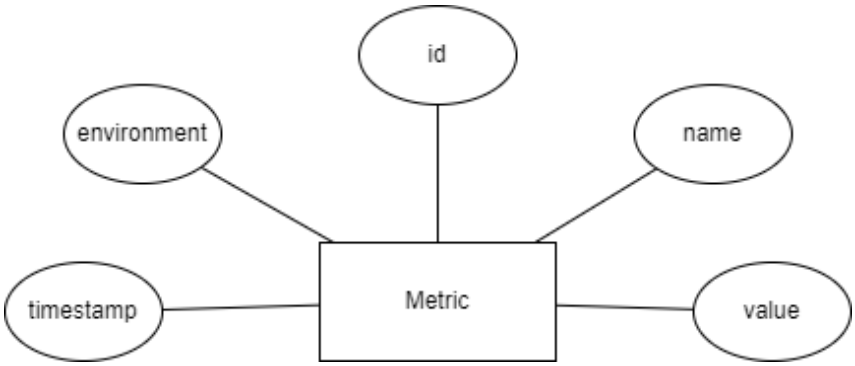


Figure 3.2: ER diagram of the Metric

### 3.2.3 Metrics Module

The metrics module is represented by a library that enables the calculation, aggregation, and collection of data about the function where it is utilized. As mentioned earlier, the metrics module is deployed alongside the function, and we will now see how its functionalities can be used to obtain specific metrics.

For example, the developer can register the execution time of a function or register the number of times a function is called. The metrics module has for now the following functionalities to achieve that:

1. `startTime` that will save the current timestamp and return an id to use to stop the time for this metric;
2. `endTime` that if it is called with the id returned by `startTime`, it will calculate the time interval between the two calls. This way the developer can analyze how long it took to run some code;
3. `startCounter` that starts a counter with the value 0 and returns an identifier.
4. `incCounter` increments a counter if the instruction with the call to this method is reached. If called more times passing the same id as a parameter will increment the same counter initiated by the call to `startCounter`.

In addition to the execution metrics presented, more functionalities could have been added to measure, for example, resource metrics, specifically features to measure CPU and memory usage. However, given the very specific granularity desired with this solution, it becomes a difficult and complex task to implement.

First, due to the nature of this type of function, particularly because they are serverless functions, we could take advantage of the platform used to manage the function.

However, not all platforms provide sufficient context to obtain such data, which comes from the operating system and the runtime environment.

Second, as these resources are part of the operating system or the runtime environment, and since the FaaS platform does not guarantee sufficient monitoring points for access to this information, the most that could be calculated would be rough estimates based on the complexity of the code, which in turn would be very imprecise.

An alternative would be to add third-party libraries or technology to the metrics module to assist in performing these types of measurements, but that would require consuming many more resources to deploy these libraries along with the function bundle and could be necessary to manage more network connections with the function container. As known, and as discussed in this dissertation, more resources can imply higher costs for FaaS providers.

One example of these technologies that can be used in Java environments is Java Management Extensions (JMX). JMX allows to monitor and manage Java applications, services, and the JVM itself. However, when using JMX, even locally, we should still consider resource usage, scalability, and platform-specific constraints. Using local JMX within the same container in FaaS environments will only simplify some aspects related to containerization and long-running processes.

### 3.2.4 Use Cases

Next, it will be shown how the developer can use the Metrics module. The Listing 3.1 contains the function code that is not using the module. This function starts to call two internal functions ( `func1()` and `func2()` ) and then verifies a condition, if the condition is true then another internal function is called ( `func3()` ).

Listing 3.1: Original Function

```
1 function faasFunction() {
2     func1();
3     func2();
4
5     if(condition) {
6         func3();
7     }
8 }
```

In Listing 3.2, we present the function using the Metrics module, in this case, the developer wants to register the execution time of `func1()` and `func2()` and wants to register the number of times `func3()` is called. For that, we need to make four calls to the Metrics module as can be seen, and at the end of the function is also necessary to call the `end()` method to indicate that the registered data can be sent to the service.

In the library, we need to store some data to guarantee the consistency of the metrics. For the examples presented next, is stored the timestamp obtained from the `startTime()` call and also the counter calculated in the `incCounter()` call. This data is stored in memory only during the execution of the function.

Listing 3.2: Function using the Metrics Module

```
function faasFunction() {
    var idTime = metrics.startTime();
    func1();
    func2();
    metrics.endTime(idTime);

    var idCounter = metrics.startCounter();
    if(condition) {
        func3();
        metrics.incCounter(idCounter);
    }

    metrics.end();
}
```

In the Figure 3.3 is presented a diagram with the steps when using the proposed architecture, already described in the Figure 3.1. With this diagram, we can have a visual representation of how the user interacts with the system, and how the system components interact with each other and we can also identify what are the dependencies.

1. The developer will start to write a function and to use the Metrics module it had to be included in the function bundle;
2. The developer has to deploy the function he writes to the FaaS Platform;
3. Then the developer can invoke the function;

4. When the function is invoked, the code in the Metrics module will send a request to the service asking to calculate the identifiers of each metric registered;
5. At the end of the function execution, the metrics are sent to the service;
6. The Service will process those metrics;
7. And then will store the data in the database;
8. The metrics can also be requested by a Monitoring Tool.

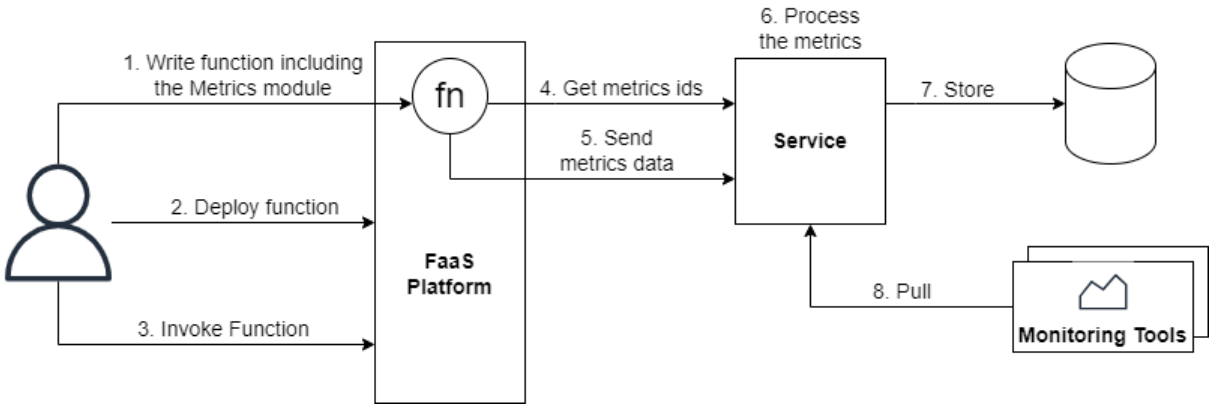


Figure 3.3: Communication diagram of the solution

## 3.3 Platform Dependencies

This section will present the platform dependencies that we have external to the implemented solution and which of them are necessary for the proposed monitoring system to work.

### 3.3.1 FaaS Platforms

As previously mentioned in this chapter, the proposed solution's architecture includes a module that represents a FaaS platform. The dependency on such a platform, such as OpenFaaS or Google Cloud Functions, exists because it is where the functions will be deployed and run. These platforms provide the runtime environment for executing the functions. The advantages of this approach are that developers do not need to concern themselves with infrastructure management, and hosting functions in the cloud ensures better availability due to more effective server management for hosting the functions.

### 3.3.2 Monitoring Tools

The possibility of integrating the solution with monitoring tools was also presented before. It is not a strict requirement for the solution to operate, but there are several benefits to integrating this solution with monitoring tools, especially if they are well-suited for function monitoring and FaaS environments, as is the case with Prometheus.

- **Data Aggregation:** These tools allow for data aggregation and filtering of metrics, enabling more valuable insights and analysis.
- **Alerting and Notifications:** They typically include alerting and notification systems based on the metrics collected. This ensures that issues or anomalies can be promptly addressed.
- **Visualization:** Integration with graphical visualization tools enables the creation of charts and tables for better data analysis and understanding.
- **Historical Data:** Monitoring tools often store historical metric records. This historical data is crucial for trend analysis, performance monitoring, and capacity planning.

## 3.4 Summary

We have seen in this chapter a brief description of a monitoring system that will provide us with the possibility to measure fine-grained metrics in FaaS scenarios. It described the proposed architecture and its components and how they interact. To understand how to use the library some use cases were presented too. Now that we have seen what is intended to be obtained as a monitoring system, we can see in the next chapter how that is applied to the tools and platforms with the details of the implementation.

One challenge that will be discussed in the next chapter is the metrics identifier. Initially, the identifier for each metric was given by the developer in the code but the intention was for the library to be able to generate the identifiers automatically. However, in FaaS environments, it can be challenging to generate meaningful and unique identifiers for metrics. Functions are often short-lived and stateless, making it difficult to associate metrics with specific function instances.

# 4

# Implementation

This chapter provides detailed information about the previously presented solution. It specifies the technologies used and describes the respective diagrams of each component in the proposed architecture to create a fine-grained monitoring system. This chapter starts with an overview of the components deployed and that consists of three main sections: **Metrics Collection**, which presents the functionalities of the library used to collect the metrics; **Metrics Processing**, which describes the service responsible for processing the metrics; and **Metrics Storage**, which explains the implementation of the database used in this system.

## 4.1 Overview

Through the component diagram shown in the Figure 4.1, we can see the components of the monitoring system presented in the previous chapter, making it easier to describe them. In the diagram, we have two nodes.

The first one is labelled **FaaS Platform**, which represents the execution environment where the deployed function runs and it can be on any platform, like those shown, as an example in the figure. Along with the function which is written in Java [26], the JAR containing the metrics library is also deployed, named **FineGrainedMetricsLib.jar**.

The second node represents a **Cloud Service Provider** because it is through the tools of those providers, such as the ones depicted in the image, that both the monitoring

service and the database are deployed and they are deployed separately. Communication between the function and the service, which establishes the connection between the two nodes in this diagram, is done via HTTP, with the characteristics that we will define in the following sections of this chapter.

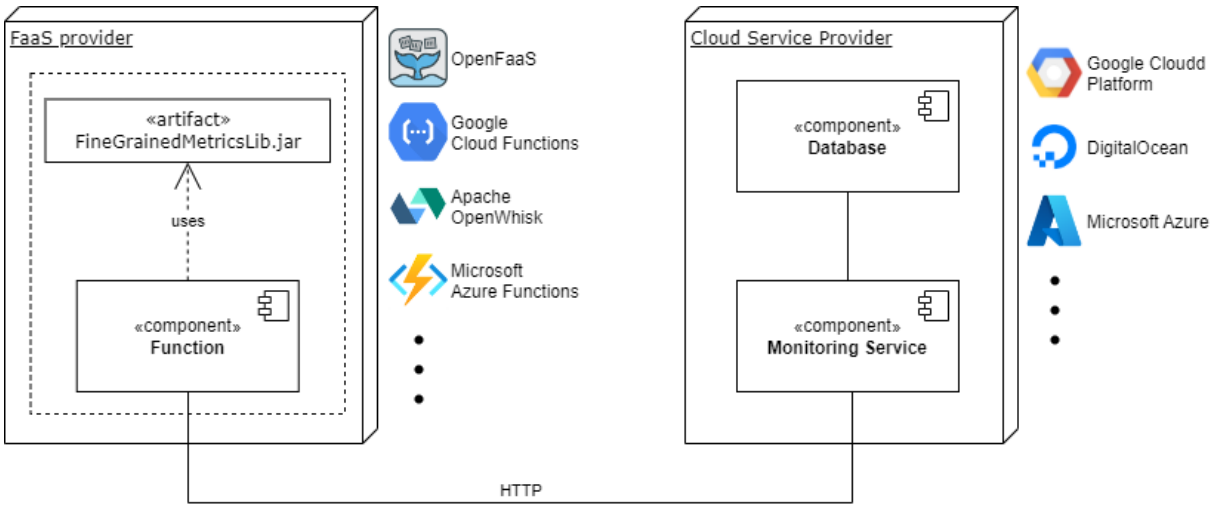


Figure 4.1: Components diagram

## 4.2 Metrics Collection

To collect metrics during the execution of FaaS functions, a monitoring library was implemented in the Java language. This library currently has two functionalities: the first functionality is the ability to collect the execution time of a code snippet, and the second functionality is the ability to increment a counter.

### 4.2.1 API Definition

In the Listing 4.1, we can see the signatures of the public methods of the API that constitute the library, as well as the data structures necessary to store the metrics.

Because of what is stated in the previous chapter, we already know that when calling the `startTime()` method and then the `endTime()` method, passing the identifier returned by the `startTime()` , we are registering a metric of the time that passed between the two calls. And, we know too, that `startCounter()` starts a counter with value 0 to register a metric with a counter and that `incCounter()` increases that counter. Finally, the `end()` method sends all the metrics stored to the monitoring service.

Regarding the data structures present in the Listing 4.1, the `time` map stores data about timestamps, based on their identifiers, whenever the `startTime()` method is called. The `counter` map stores data about counters whenever the `startCounter()` method is called and is also based on an identifier.

The last map we see, named `metrics` stores all the metrics in a format that allows the data to be transformed into JSON format and sent in an HTTP request when the `end()` method is called. This data, stored based on an identifier, is of the `Metric` type, which includes `name`, `value` and `timestamp` fields. This type has the same definition as the class used in the business model of the monitoring service, as we will see later in this chapter, and, as stated before, all of this data is only in memory during each function execution.

Although the `metrics` map seems redundant, it is used to simplify the conversion of the metrics data into a format suitable to send in the HTTP request to the monitoring service at the end of the function execution.

Listing 4.1: API definition

```
1  public class MetricsLib {
2
3    private final Map<String, Long> time;
4    private final Map<String, Long> counter;
5    private final Map<String, Metric> metrics;
6
7    public String startTime() { ... }
8
9    public Metric endTime(String id) { ... }
10
11   public Metric startCounter() { ... }
12
13   public Metric incCounter(String id) { ... }
14
15   public String end() { ... }
16 }
```

### 4.2.2 Deployment Configuration

To understand how the metrics library is deployed along with the function, it is necessary to know the environment variables that we need to indicate when defining the deployment configuration. The next paragraphs describe what is relevant about those environmental variables.

Communication with an external service from the function is something that can increase the execution time of the function or even decrease performance. Even without considering this possibility, the user may simply not need or want to use the monitoring service to register their metrics, for example, the user could just want to do some debugging for analysis or to print the data somehow.

If the user does not specify that they want to take advantage of the monitoring service provided, by default, the library will not communicate with the service. However, when deploying each function, it is possible to specify an environment variable to determine whether to use the monitoring service or not. If the user decides to have their function communicate with the service, they need to include the environment variable **SERVICE**. In this specific case should be **SERVICE=true**.

Two more environment variables may need to be specified during the deployment if the user wants to communicate with the service. One of them is for when the user wishes to communicate with the service and he must also include the **HOST** environment variable, specifying the URL where the service is located and running.

Another environment variable that was necessary to add when there is communication with the service is the **FUNCTION** environment variable, which contains the name of the function. Since we need to store the metrics in a database or any other type of storage, their identifier must have something that distinguishes the metrics from different functions. As described earlier, one of the difficulties in generating an identifier for each metric is how to distinguish the environment in which the function runs or know which function is being invoked. This is challenging because there is a variety of tools, and they differ from one another. Some platforms even include an environment variable with the function name, but the name of this variable is different from one provider to another, and in some providers, it does not exist at all. Hence, the need to add this last variable to the function's deployment.

### 4.2.3 Metrics Identifier

When the library is configured to communicate with the service, whenever one of its methods is called to get one of the metrics mentioned in the functionalities, it sends an

HTTP request to the service to request the calculation of the metric's identifier being measured. For this, it needs to send an HTTP request that includes the metric's name and the function's name.

When there is no communication with the service, the library itself calculates the identifier using only the metric's name because in this case, where there is no need to persist this data, we only need to identify the metric within the scope of the function itself.

In the Listing 4.2 and the Listing 4.3, we can see two code snippets that are related to the metric's id calculation.

The first one is the method that calculates the identifier by sending an HTTP request to the service, and the second one is the method that calculates the identifier internally in the library, respectively. If the communication with the the service is not intended, when the `getMetricId` method is called, will call the `getMetricIdNoService` method.

Listing 4.2: Calculation of the metric identifier when communicating with the monitoring service

```
1  private String getMetricId(String metricName)
2  {
3    if(!useService)
4    {
5      return getMetricIdNoService(metricName);
6    }
7
8    final HttpUrl.Builder urlBuilder = HttpUrl.parse(host + "/id").
     newBuilder()
9        .addQueryParameter("metric", metricName)
10       .addQueryParameter("envId", functionName);
11
12   Request request = new Request.Builder()
13       .headers(Headers.of("Content-Type", "application/json"))
14       .url(urlBuilder.build().toString()).build();
15
16   Response response = client.newCall(request).execute();
17   return response.body().string();
18 }
```

Listing 4.3: Calculation of the metric identifier when not communicating with the monitoring service

```
1  private String getMetricIdNoService(String metricName)
2  {
3    String uniqueId = metricName;
4    int index = 1;
5    while(noServiceIdsList.contains(uniqueId))
6    {
7      uniqueId = metricName + index;
8      index++;
9    }
10   noServiceIdsList.add(uniqueId);
11   return uniqueId;
12 }
```

We can see mentioned in the listings presented above, the `useService`, `host`, and `functionName` variables, which are three environment variables declared as instance variables of the class. These variables are instantiated each time the library is instantiated within the function and a new object class is created.

In the `getMetricIdNoService()` method, is visible a reference to a list called `noServiceIdsList` that stores the identifiers already used when calculating them internally in the library.

Lastly, there is another instance variable named `client`. To perform HTTP requests between the library and the service, two external libraries were required and added as dependencies: `com.squareup.okhttp3.okhttp` [36] and `com.squareup.okio.okio` [37]. The `client` variable, which is used for making HTTP requests, occupies memory space each time it is instantiated. Therefore, instead of instantiating it every time an HTTP request is needed, it is instantiated only once when the metrics library is instantiated within the function.

Additionally, another dependency for another library, `org.json.json` [27], was added to allow parsing the metric data stored across multiple calls to the library, into a JSON format for sending in the HTTP request.

## 4.3 Metrics Processing

For processing the metrics received from multiple functions, a Web API was implemented using the Spring Framework [62], which was written in the Java language. The base URL of the API may vary depending on which cloud provider the service is deployed, but below are its endpoints for the available resources related to metric processing:

- **GET /api/metrics/id?metric={name}&envId={id}**
- **POST /api/metrics**

The first request is a GET request that returns an identifier for a metric. It includes the `metric` and the `envId` path parameters that correspond to the metric name and the function identification, respectively.

The second request is a POST request to create a new metric. To perform this last one, we need to send the required data in the request body. An example of how the request body can be structured is visible in the following Listing 4.4. In this example, are presented two metrics: one with the `name` counter in which the `value` is **1** and the `timestamp` is 1694128248393955000, and the other with the `name` time in which the `value` is 296682 and the `timestamp` is 1694128286071881000. The `timestamp` is reported in nanoseconds as long as the `value` for the time metric.

Listing 4.4: Body of the POST request to create a new metric

```
[
  {
    "name": "counter",
    "value": "1",
    "timestamp": "1694128248393955000"
  },
  {
    "name": "time",
    "value": "296682",
    "timestamp": "1694128286071881000"
  },
]
```

### 4.3.1 Spring Application

The class diagram of the service module is present in the Figure 4.2. This web API has a three-layered architecture, dividing the application into Controller, Service, and Repository. In this case, since it is an API, there was no need to implement a component representing the view.

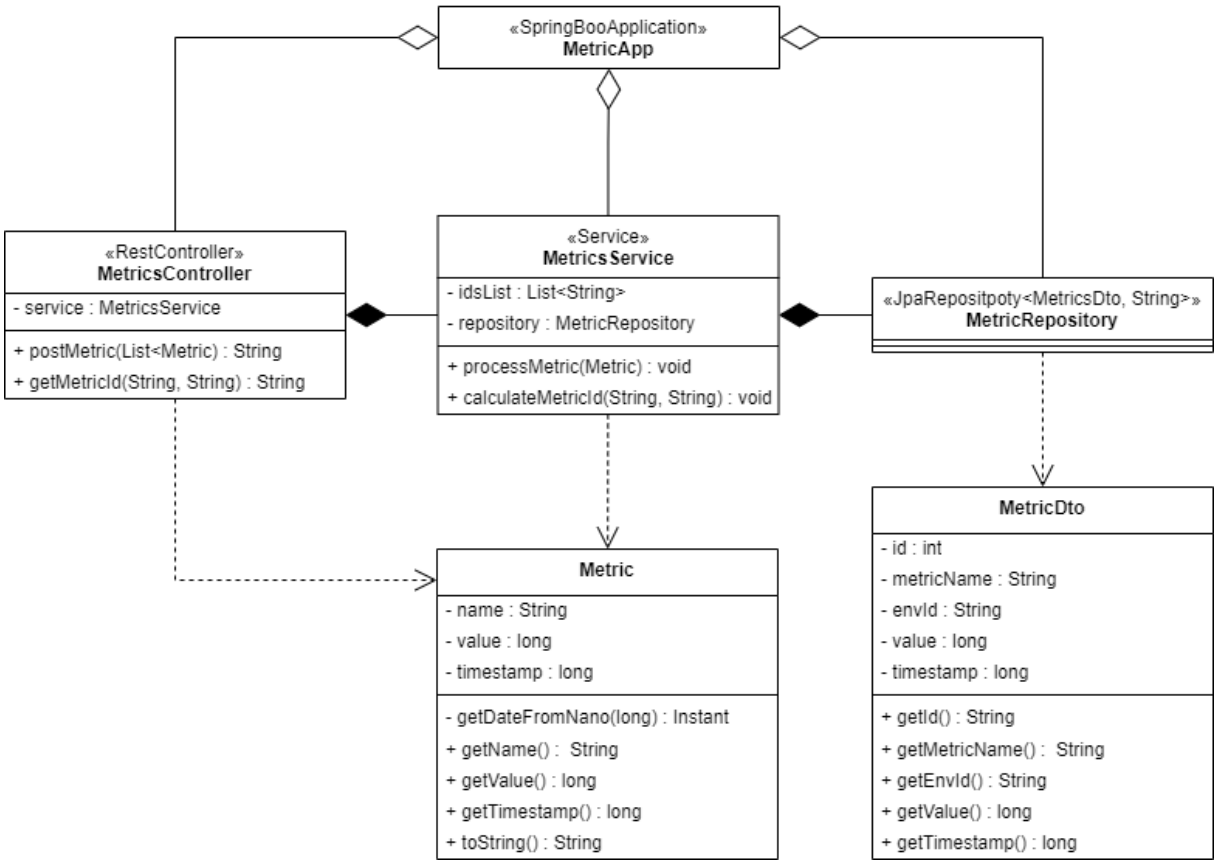


Figure 4.2: Class diagram of the Service module

In the figure, there is a controller represented by the `MetricsController` class, where, through the `@RestController` annotation, a REST API is constructed with the previously mentioned endpoints. This architecture, widely used in web APIs, emphasizes stateless communication and resource-oriented design.

On the other hand, there is the `MetricsService` class, which, with the `@Service` annotation from Spring, contains the business logic of the application. As seen in the figure, it has two operations.

One operation calculates an identifier for a Metric and receives two parameters: the metric's name and an identifier of the function's environment from which this metric

was collected (that could be for example the function's name). The other operation creates a DTO (Data Transfer Object) to be processed by the data access layer.

The data access layer, represented by the `MetricRepository` interface, has the responsibility to provide CRUD (Create, Read, Update and Delete) operations for the `MetricDto` object. These operations are inherited because `MetricRepository` extends the `JpaRepository` interface from Spring, which, in turn, has `MetricDto` and `String` as generic types. `MetricDto` is the type of the entity on which the operations are applied, and `String` is the type of the identifier for that entity. With this repository implementation, we can, for example, save new metric data in the database.

Although was not obvious in the description, this implementation of the repository has inherited also read operations, which combined with the way the data is stored (explained in more detail in the section above) allows for the service aggregation of the metrics data. The user can get for example the time execution information from the last one hundred invocations. For that has to be implemented a query that filters the metrics in the database by the environment identification and by the metric name in question, which in this specific case is called `time`.

## 4.4 Metrics Storage

In the implemented monitoring system, it was necessary to create a database to store metric data. This database not only aims to persist metric data but also to maintain a historical record of all the information collected over time. The database was implemented using PostgreSQL [47] version 15 and consists of two entities: `metrics` and `metrics_historic`. In Figure 3.2 presented in the previous chapter, we can see the definition of the `metrics` entity. The `metrics_historic` entity has the same fields in its composition.

To ensure that historical data is written to the `metrics_historic` table, two SQL triggers and their respective functions were created.

In the Listing 4.5 and the Listing 4.6, we can see the code that inserts rows into the `metrics_historic` table whenever insertions occur in the `metrics` table, copying the original rows into the historic table.

In the Listing 4.7 and the Listing 4.8, we can see the code that deletes old entries from the `metrics` table whenever new rows are inserted into the `metrics_historic` table, using the `metric_name`, `env_id` and `id` attributes to identify which entries to delete.

Listing 4.5: Trigger to insert data on metrics_historic table

```
CREATE TRIGGER insert_history_trigger
AFTER INSERT
ON metrics
FOR EACH ROW
EXECUTE FUNCTION insert_history_func();
```

Listing 4.6: Function from the trigger to insert data on metrics_historic table

```
CREATE OR REPLACE FUNCTION insert_historic_func()
RETURNS trigger
begin
insert into metrics_historic select new.*;
return null;
end;
```

Listing 4.7: Trigger to delete old data from the metric table

```
CREATE TRIGGER delete_old_trigger
AFTER INSERT
ON metrics_historic
FOR EACH ROW
EXECUTE FUNCTION delete_old_func();
```

Listing 4.8: Function from the trigger to delete old data from the metric table

```
CREATE OR REPLACE FUNCTION delete_old_func()
RETURNS trigger
begin
delete from metrics
where metric_name = new.metric_name
and env_id = new.env_id
and id != new.id;
return null;
end;
```

## 4.5 Summary

In this chapter, the implementation of the monitoring system proposed for achieving fine-grained monitoring was demonstrated. This involves the identification of the technologies used, and the exploration of the software design decisions made for the entire system.

In the next chapter, the performance of the system will be evaluated. Comparisons and analyses of the results will be made between some scenarios and platforms to identify the advantages and challenges encountered when using this solution.

# 5

# Evaluation

This chapter will start to define some performance factors to measure the impacts and performance of the proposed solution. Some test scenarios were performed in two different environments: one completely local which made it easier to debug the implementation and perform the first tests, and another that was a remote environment where we would have a more realistic approach to dealing with monitoring in the context of cloud technologies. At the end of this chapter, after the demonstration and the analysis of the results, conclusions are presented.

## 5.1 Overview

To test and evaluate the system proposed above, it is recommended to analyze some common performance factors in the evaluation of systems that involve serverless functions, such as function execution time, function response time, scalability, reliability, cost, or security.

For the evaluation of this specific case, where we have an extra library being deployed with the function and a metrics service that could increase the latency when using this solution, the question arises as to how the presented monitoring service will behave when there is only one function reporting data or when there are one hundred functions reporting data, both for the monitoring platform and for a database. In addition to this question, and given that the initial function is being extended to calculate certain

metrics, it would be important to test whether the execution time degrades proportionally or whether there is a significant loss of performance as more metrics are added to the function. In this sense, it is intended to evaluate the scalability at the function level and to verify whether the proposed fine-grained monitoring of FaaS functions is worthwhile in terms of the costs that it may incur.

## 5.2 Use Cases

To proceed with the evaluation of the solution presented, two use cases were defined, each with its function. The first function consists of simply printing some messages to the console and represents the simulation of a low-processing function.

On the other hand, the second test function receives an image encoded in base64 [8] when called, and transforms that information into a grayscale image, returning it as the result, also in base64 format. This operation simulates processing that requires more resources than in the first example and is therefore considered a high-processing function from now on. Both functions were written in the Java language.

## 5.3 Performance Factors

The performance factors defined for evaluating the previously presented use cases were the average execution time, the average memory usage during the function execution, and the storage occupied by the function. This process was initially carried out in a locally installed environment on a virtual machine and later performed in a remote environment created using cloud resources.

The purpose of these performance factors is to make a comparison among three scenarios: (1) when the function has no reference to the metrics library; (2) when the function uses the library but does not communicate with the service; (3) and when the library is used to send data to the service to report the metrics. The communication with the service includes not only sending the metrics data to the service but also the request to obtain the metrics identification.

A second experiment aimed to observe how the performance of using the library evolves based on the quantity of metrics calculated in the function. This includes the average execution time and the average memory usage. To create the functions for this case, it was developed a code generator to add the metrics instructions in the function of a given distribution.

Finally, the storage occupied by the function was also calculated with and without the library.

## 5.4 Local Evaluation

To acquire initial evaluation results, a local environment was established. This not only provides an initial impression that allows us to draw preliminary conclusions but also simplifies the process of testing the system and performing debugging tasks.

### 5.4.1 OpenFaaS Environment

For the local evaluation, the Function-as-a-Service framework used was OpenFaaS. To set up OpenFaaS locally, the process began by creating a virtual machine using the Multipass tool [33], which had the Ubuntu version 22 operating system [63]. The cloud-init method [10] was used, and by providing a YAML file [64], the virtual machine was instantiated with the necessary packages and software. In this case, faasd [15] was installed, which contains the core services required to run OpenFaaS without the cost and complexity of Kubernetes. This allowed the use of the OpenFaaS CLI and GUI to perform the necessary operations for local testing.

Once OpenFaaS was installed, Docker was also installed, and a container registry was created to deploy container images of the functions. In this setup, faasd was configured to use a self-hosted registry container from GitHub.

The functions in OpenFaaS were created using a Java template provided by the OpenFaaS tool itself, ensuring they had a compatible format for being invoked through HTTP requests. This template can be modified by the user, and in this case, it was altered to include the library JARs where needed. Java functions used Gradle [24] as the build tool, so the function template was also modified to enable the interpretation of files contained in the JAR with the library.

Once a function was created, three steps were performed, with CLI commands having similar names to Docker commands: `build`, `push`, and `deploy` (using the `faas-up` CLI command, all three operations can be done in a single instruction). The build step compiles the function into an image in the local Docker library, the push step sends the image to the remote container registry, and the deploy step publishes the function on the virtual machine. Of these three steps, only the deploy step can be done in the OpenFaaS GUI or by sending HTTP requests to the OpenFaaS API Gateway. Therefore, except for function invocation (which was done through HTTP requests),

the deploy and log querying operations were performed using the CLI. Environment variables can also be specified in the deploy command, but they can also be included in the YAML configuration files for each function image.

### 5.4.2 Methodology

Before the functions were invoked, the service and the database server were started locally. Then, for the tests in OpenFaaS, a separate function was deployed for each of the tests performed. Each of these functions was invoked one hundred times through HTTP requests made to the OpenFaaS API Gateway, programmatically. The request had the structure in Listing 5.1 (when sending an image, it had to be added to the request's body, and the `content-type` field had to be set to `image/png`):

Listing 5.1: HttpRequest to invoke function in OpenFaaS

```
1 HttpRequest request = HttpRequest.newBuilder()
2   .POST(HttpRequest.BodyPublishers.noBody())
3   .uri(URI.create(HOST/function/FUNCTION_NAME)))
4   .header("Authorization", "Basic " + Base64.getEncoder().
     encodeToString((USERNAME:PASSWORD).getBytes()))
5   .build();
```

The performance factors related to the function execution time were collected programmatically using the response body returned from the HTTP request, to find out the time it took for each invocation of the function to execute. In this manner, with the hundred invocations, was obtained the average.

The memory and CPU usage data were not obtainable because these metrics are only available in the paid version of OpenFaaS, to which we did not have access, and the amount of storage occupied was obtained from the local file system, which gives us the resource memory occupied in the disk.

### 5.4.3 Analysis

In the Figure 5.1, we observe that in the low processing function, when there is no communication with the service, the average execution time of the function takes 8,4 times longer than when the library is not used at all, and when there is communication

with the service takes 67,1 times longer. In contrast, in the high processing function, there is a 28,8% increase in the average execution time when there is no communication with the service and a 49,1% increase when there is communication with the service.

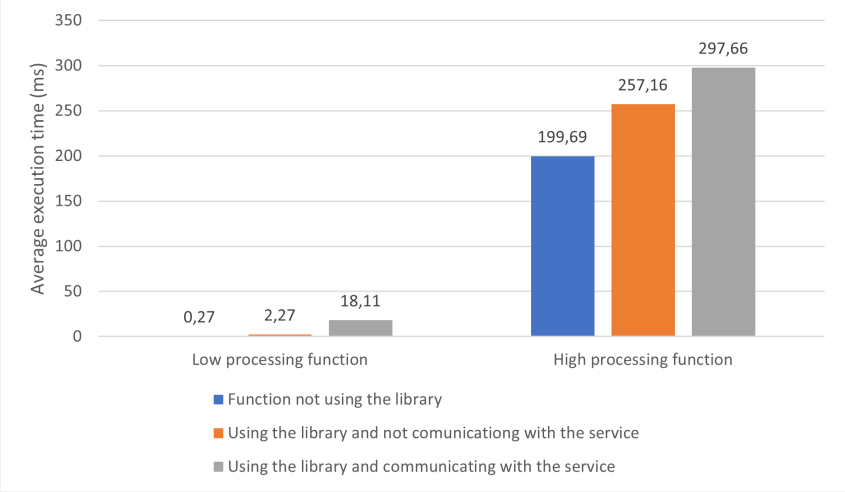


Figure 5.1: Execution time in local environment

When evaluating the average execution time, about the number of metrics obtained by the library, we can observe in the Figure 5.2 that there is an increase proportionally to the number of metrics calculated when there is communication with the service due to the amount of data being transmitted in the request. When there is no communication with the service, it can be seen that there is almost no variation in the average execution time.

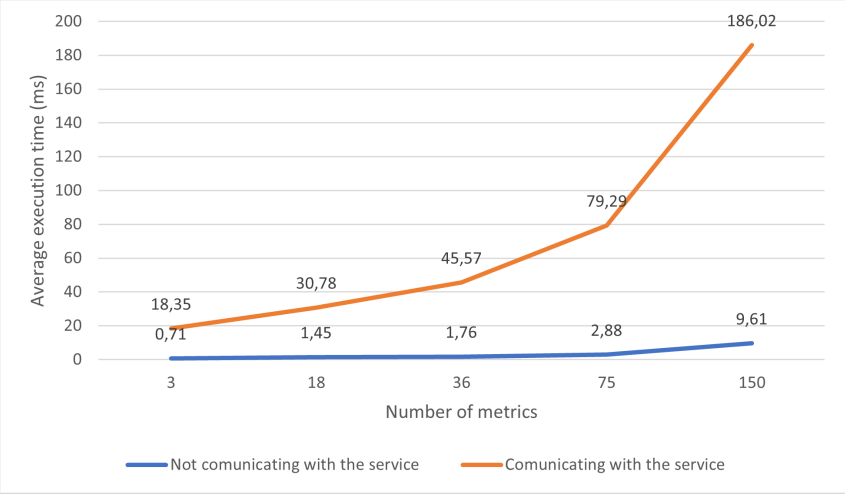


Figure 5.2: Execution time by number of metrics in local environment

Regarding the occupied storage in the file system, we can observe in the Figure 5.3 that the low processing function, when including the library, occupies an additional 7,3 KB, and the high processing function occupies an additional 7,7 KB. However, each container image allocates 284,66 MB for each function.

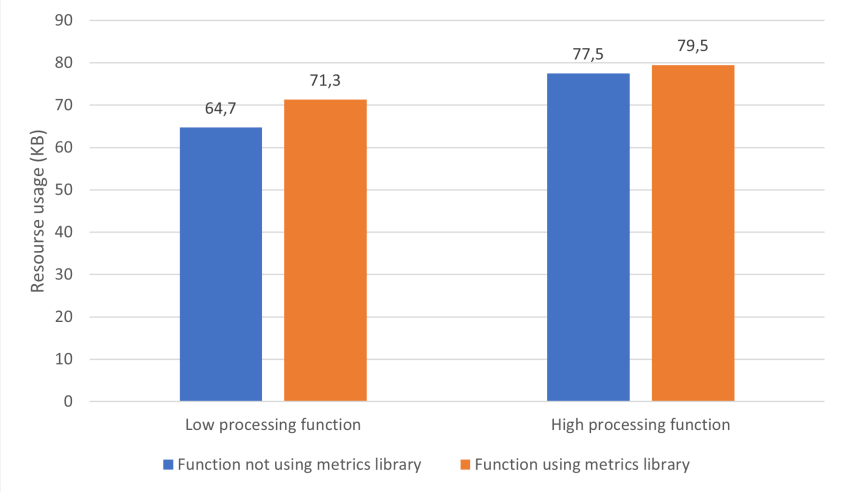


Figure 5.3: Memory resources occupied in local environment

## 5.5 Remote Evaluation

To gain a better understanding of how the proposed monitoring system behaves, a remote environment was also configured. The tests made in this environment provide us with additional insights, enabling us to obtain more informed conclusions.

### 5.5.1 Google Cloud Platform Environment

For the remote evaluation, the GCP (Google Cloud Platform) [22] was used as the cloud framework, and the same functions and use cases as in the local evaluation were performed. The deployment of functions was done using the Google Cloud Functions tool, which allows the creation, management, and execution of serverless functions. The metrics service was deployed using the App Engine tool [21], and the database was hosted on a Cloud SQL instance [23] within the platform.

In Table 5.1, Table 5.2 and Table 5.3 we can see the configurations, used in the cloud, for the functions, the service and the database, respectively.

Table 5.1: Configurations used in GCP for the functions

| Location | Runtime | Memory allocated |
|---|---|---|
| europe-west1 | Java17 | 256 MB |

Table 5.2: Configurations used in GCP for the service

| Location | Runtime | Size |
|---|---|---|
| europe-west1 | Java17 | 64,2 MB |

Table 5.3: Configurations used in GCP for the database

| Location | Version | vCpus | Memory | SSD strorage |
|---|---|---|---|---|
| europe-west1 | PostgreSQL 15.2 | 1 | 3,75 GB | 10 GB |

For the creation of functions and the service, the gcloud, that is the Google Cloud CLI tool, was installed locally, and login credentials were set up to enable sending data to the platform through local command-line instructions. The functions were created by ensuring they implemented the `HttpFunction` class so that they could be triggered by HTTP requests. Gradle was used as the build tool, and in its configuration, the dependency for the metrics library was specified when needed. For the deployment, the region was specified, along with the environment variables, the entry point class, the runtime, and the trigger field (in this case HTTP).

The service was deployed using the command line tool and the database was instantiated through the GUI, but for creating the schema and viewing the data, a local connection was made using the pgAdmin tool [46]. Access authorization had to be granted in the network settings of the database instance in the cloud for this connection to work.

### 5.5.2 Methodology

To observe the results of the tests in GCP, the Google Cloud Monitoring tool was used. In this case, as well, a separate function was deployed for each of the tests conducted. Each of these two functions was invoked one hundred times using HTTP requests as the trigger, which was done programmatically. The request had the structure presented in the Listing 5.2 (when sending an image, it had to be added to the request's body, and the `content-type` field had to be set to `image/png` ):

Listing 5.2: HttpRequest to invoke function in GCP

```
1 HttpRequest request = HttpRequest.newBuilder()
2     .POST(HttpRequest.BodyPublishers.noBody())
3     .uri(URI.create(HOST/FUNCTION_NAME))
4     .build();
```

The performance factors related to the function execution (average time and average memory usage) were collected from the GUI of Google Cloud Monitoring. The values were filtered by status ok, grouped by function name using the mean as the grouping function and were used all the values collected in a time interval of 15 minutes.

The amount of the exact storage occupied in disk was not obtained from GCP but it is known that the memory allocated to the container of each function is 256 MB.

### 5.5.3 Analysis

In the Figure 5.4, it is possible to observe that in the low processing function, when there is communication with the service, the average execution time of the function takes 19,6 times longer than when the library is not used, and when there is communication with the service takes 8.072 times longer, which is considered a big variation between the two cases. In the high processing function, there was a 22,7% increase in the average execution time when there is no communication with the service and a 58,1% increase when there is.

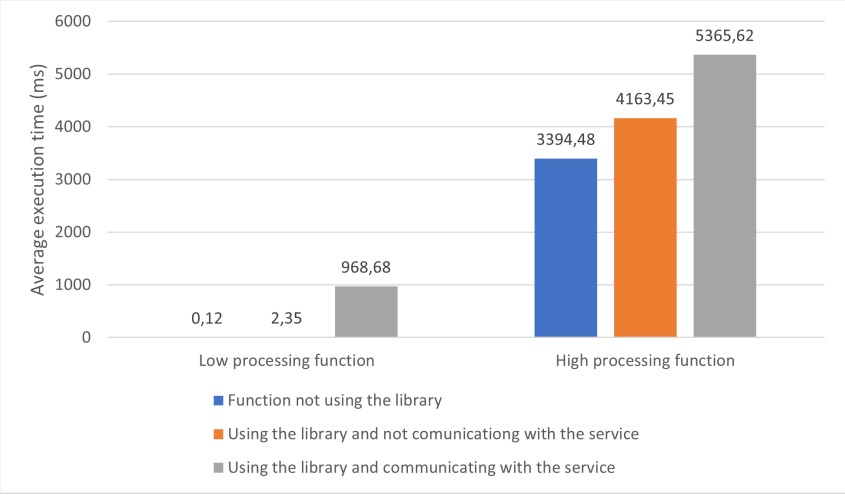


Figure 5.4: Execution time in remote environment

Regarding memory usage in the Figure 5.5, it is possible to observe that in the low processing function, there is practically no difference when there is communication with the service but a 101,3% increase when there is communication with the service. In the high processing function, again there is not a big difference between the three cases. They vary between 159,49 MiB and 194,48 MiB.

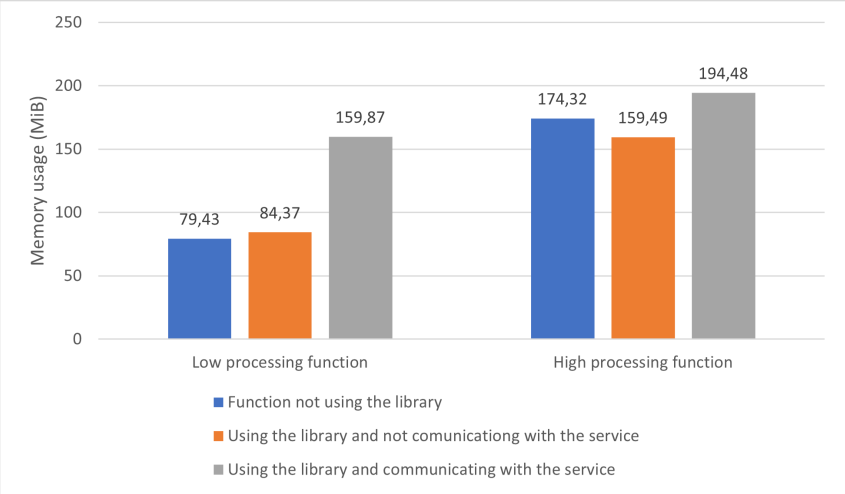


Figure 5.5: Memory usage in remote environment

When evaluating the average execution time of the low processing function, about the number of metrics obtained by the library, we can observe in the Figure 5.6 that there is a significant increase when there is communication with the service due to a larger amount of data being transmitted in a single request. When there is communication with the service, it can be seen that there is not a variation so visible in the average execution time of the functions.

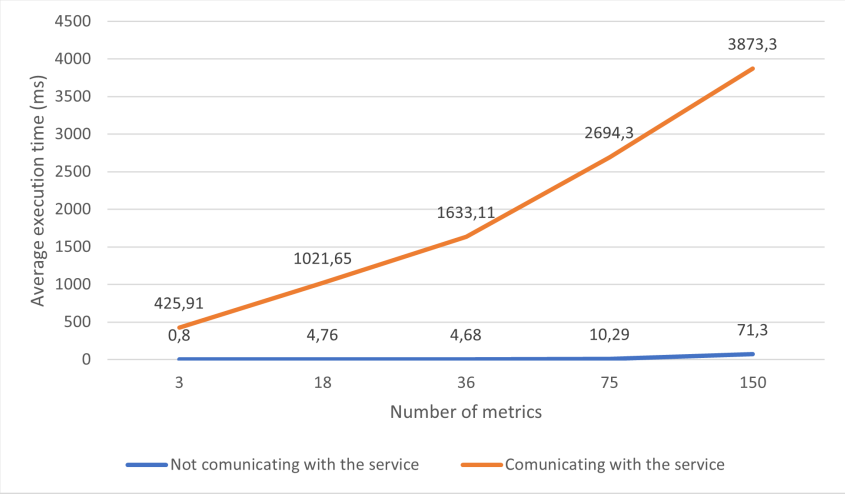


Figure 5.6: Execution time by number of metrics in remote environment

In memory usage, the same does not occur, and we can observe in the Figure 5.7 that there is not a significant variation in each line of the chart.

When there is communication with the service the memory usage varies between 150,29 MiB and 165,11 MiB and when there is no communication with the service vary between 83,95 MiB and 119 MiB. Therefore, this performance factor is independent of the number of metrics recorded but the amount of data sent to the service can influence it.

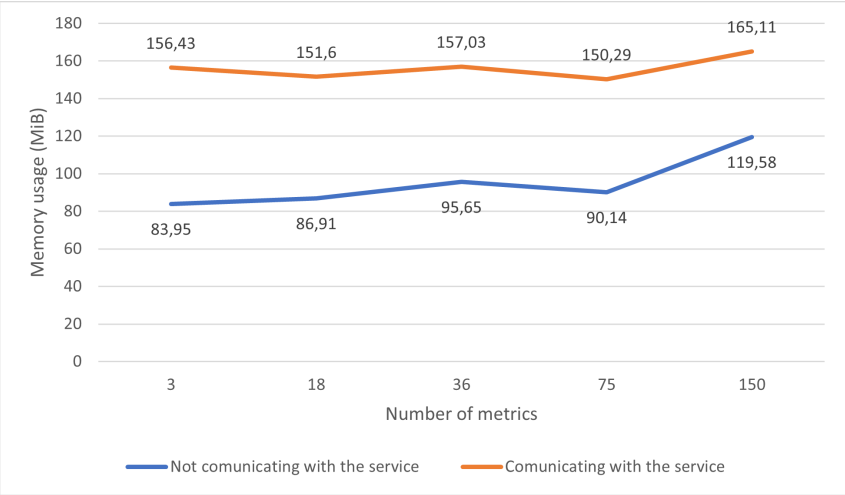


Figure 5.7: Memory usage by number of metrics in remote environment

## 5.6 Summary

This chapter presented the two use cases used for the evaluation of the proposed solution. It also described how the environment was set up for performing the tests, both locally and remotely. For each test performed, the calculated performance factors were presented, followed by an analysis of the obtained values.

The evaluation conducted offers valuable insights into the impact of simultaneously having communication with the monitoring service while obtaining metrics. We can see that this can significantly extend the execution time of functions within the system. This prolongation of execution time can be a trade-off in this monitoring system solution.

In contrast, when evaluating memory usage, the findings reveal a different conclusion on the system's performance. Notably, there is no substantial impact on memory consumption, whether only using the metrics library or opting for having HTTP communication with the monitoring service.

The decrease in the function performance observed is the cost of obtaining metrics with a high level of detail during its execution, and it is possible to further improve the results by addressing some of the remaining challenges, as described in the future work section in the next chapter.

# 6

# Conclusions

This chapter summarizes the main findings and contributions of the research done throughout the development of this work and its report. It will be analyzed and discussed the implications and limitations found and how the results obtained during the evaluation, affected the conclusions about that. Also, it will be described what can be done in terms of future work and future research too, since the solution can be improved in some aspects.

## 6.1 Recap Research Goals

This work goes into the details of designing a system for a meticulous monitoring of functions within the domain of FaaS. The main purpose of this research was to obtain a versatile monitoring solution capable of integrating into various FaaS platforms.

For this achievement was imperative to have a fine-grained function monitoring at the code level, within FaaS environments. Function execution data is fundamental for enhancing system performance and diagnosing issues. Because of that, the research and the development were to achieve a comprehensive system that not only gets important data but also facilitates its accessibility through other monitoring tools, e.g., Prometheus.

One of the challenges in this was to create a system that transcends platform-specific limitations. In essence, the designed solution aspires to be platform agnostic, ensuring

compatibility with multiple FaaS providers and monitoring tools. Moreover, it was wanted that the system minimize the local storage to optimize resource utilization.

By addressing these points, the research aimed to contribute to the evolving landscape of cloud computing, where efficient resource management and performance optimization are a priority.

## 6.2 Achievements

Given the implemented solution and the goals mentioned above, we provide the user the ability to monitor functions with a fine-grained specificity. The solution can also be used in different FaaS platforms and the implementation is made taking into account the integration with a monitoring tool like Prometheus.

We can also guarantee with this solution some efficiency by using a small local storage. Besides, it was proven by the evaluation of some performance factors that this solution can consume a minimal amount of resources and that, along with the fact of using small local storage, will save users in cloud resources. In conclusion, this is a solution very simple for users to use and that provides some degree of customization for cost effective approaches.

## 6.3 Limitations

Some obstacles and difficulties emerged during this project, causing the solution not to align precisely with the initial proposal. These challenges, while unexpected, provided valuable insights and opportunities for growth:

1. **Limited Documentation:** One of the primary obstacles encountered was the lack of documentation, particularly in the deployment of functions within the OpenFaaS framework. This information gap added complexity to the development process and has increased the amount of trial and error.

2. **Debugging Challenges:** Debugging functions was another challenge. Identifying and resolving errors during the development was a time-consuming task.

3. **Platform Differences:** A critical challenge derives from the differences between FaaS platforms. Distinguishing a specific function and the environment in which

it executes became a complex task. FaaS platforms often have different deployment templates and configurations for functions so for each platform the developer has to write a different function. Also, the functions deployed on various FaaS platforms may rely on different environment variables for configuration and runtime information.

4. **Scope of Evaluation:** The evaluation process should have been more exhaustive but it was conducted on a limited set of platforms. Expanding the evaluation to more platforms would have provided a better understanding of the system's performance.

## 6.4 Future Work

From now on, there is work that can be continued in both research and development to make this a better solution and provide more knowledge in the technological field, specifically in the areas of cloud computing, FaaS platforms, and function monitoring.

One of the aspects that needs improvement is the integration with a monitoring platform, particularly Prometheus. This integration will allow for metric aggregations, such as averages, for example, and could potentially be valuable for visualizing the data. This would make the analysis faster and more intuitive.

Another crucial aspect to consider is implementing an approach that separates the sending of metric data from the library to the service, from the actual function execution. As we saw in the conclusions drawn in the evaluation chapter, the function's execution time can increase significantly when an HTTP request is made to the service. The idea would be to analyze and test the possibility of implementing a solution that involves different threads to perform this task.

Improvements to be made include enhancing the metric library used within the function to have more metric calculation functionalities. Additionally, the service itself could be improved to have data aggregation logic. This would make the solution more independent of an external monitoring service, although that is not necessarily a disadvantage.

Something that was also missing was adapting the solution to work in parallel execution scenarios. In this case, the implementation would need improvements, and it would be necessary to perform a wider range of tests to enrich the evaluation. Specifically, run tests that take into account, for example, the execution of the same function a hundred times, but with the invocations made simultaneously.

# References


[1] Erwin van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup, "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms", *IEEE Internet Computing*, vol. 23, no. 6, pages 7–18, 2019. DOI: 10.1109/MIC.2019.2952061.

[2] Beatriz Guerreiro, Filipe Freitas, and José Simão, "Monitoring in function-as-a-service platforms", in *2023 18th Iberian Conference on Information Systems and Technologies (CISTI)*, 2023, pages 1–4. DOI: 10.23919/CISTI58278.2023.10212053.

[3] "Amazon SQS". (Feb. 2023), [Online]. Available: https://aws.amazon.com/sqs/.

[4] "Azure Application Insights". (Feb. 2023), [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview?tabs=net.

[5] "Azure Functions". (Feb. 2023), [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview.

[6] "Azure Monitor". (Feb. 2023), [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/overview.

[7] "Azure Monitor managed service for Prometheus". (Feb. 2023), [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/prometheus-metrics-overview.

[8] "Base 64 Encoding". (Sep. 2023), [Online]. Available: https://www.rfc-editor.org/rfc/rfc4648#section-4.

[9] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms", in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pages 1–8. DOI: 10.1109/IC2E52221.2021.00014.

[10] "cloud-init Tool". (Sep. 2023), [Online]. Available: https://cloud-init.io/.

[11] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta, "Cose: Configuring serverless functions using statistical learning", in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pages 129–138. DOI: 10.1109/INFOCOM41043.2020.9155363.

[12] "Docker". (Dec. 2022), [Online]. Available: https://www.docker.com/.

[13] "faas-netes". (Dec. 2022), [Online]. Available: https://github.com/openfaas/faas-netes.

[14] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff, "Architectural implications of function-as-a-service computing", in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, 1063–1075. DOI: 10.1145/3352460.3358296.

[15] "OpenFaaS faasd". (Sep. 2023), [Online]. Available: https://docs.openfaas.com/deployment/faasd/.

[16] Jeongchul Kim and Kyungyong Lee, "Functionbench: A suite of workloads for serverless cloud function service", in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pages 502–504. DOI: 10.1109/CLOUD.2019.00091.

[17] "GitHub". (Sep. 2023), [Online]. Available: https://github.com/.

[18] "Google Cloud Functions". (Feb. 2023), [Online]. Available: https://cloud.google.com/functions/docs.

[19] "Google Cloud Monitoring". (Feb. 2023), [Online]. Available: https://cloud.google.com/monitoring.

[20] "Google Cloud Managed Service for Prometheus". (Feb. 2023), [Online]. Available: https://cloud.google.com/stackdriver/docs/managed-prometheus.

[21] "Google App Engine Application Platform". (Sep. 2023), [Online]. Available: https://cloud.google.com/appengine?hl=en.

[22] "Google Cloud Platform". (Sep. 2023), [Online]. Available: https://cloud.google.com/?hl=en.

[23] "Google Cloud SQL". (Sep. 2023), [Online]. Available: https://cloud.google.com/sql?hl=en.

[24] "Gradle Build Tool". (Sep. 2023), [Online]. Available: https://gradle.org/.

[25] "Grafana". (Feb. 2023), [Online]. Available: https://grafana.com/.

[26] "Java Programming Language". (Sep. 2023), [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html.

[27] "Json java Library". (Sep. 2023), [Online]. Available: https://square.github.io/okio/.

[28] "JSON Data Format". (Sep. 2023), [Online]. Available: https://www.json.org/json-en.html.

[29] "Apache Kafka". (Feb. 2023), [Online]. Available: https://kafka.apache.org/.

[30] "Kamon". (Feb. 2023), [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/metrics.md.

[31] "Kubernetes". (Nov. 2022), [Online]. Available: https://kubernetes.io/.

[32] Colin Adams, Luis Alonso, Ben Atkin, John P. Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George T. Talbot, Adam Jacob Tart, and Nick Taylor, Eds., *Monarch: Google's Planet-Scale In-Memory Time Series Database*, 2020, 3181–3194.

[33] "Multipass VM Tool". (Sep. 2023), [Online]. Available: https://multipass.run/.

[34] "Nagios". (Feb. 2023), [Online]. Available: https://www.nagios.org/.

[35] "NATS". (Feb. 2023), [Online]. Available: https://nats.io/.

[36] "OkHttp Library". (Sep. 2023), [Online]. Available: https://square.github.io/okhttp/.

[37] "Okio Library". (Sep. 2023), [Online]. Available: https://square.github.io/okio/.

[38] "OpenFaaS". (Jan. 2023), [Online]. Available: https://www.openfaas.com/.

[39] "OpenFaaS API Gateway". (Jan. 2023), [Online]. Available: https://docs.openfaas.com/architecture/gateway/.

[40] "OpenFaaS Metrics". (Jan. 2023), [Online]. Available: https://docs.openfaas.com/architecture/metrics/.

[41] "Apache OpenWhisk". (Feb. 2023), [Online]. Available: https://openwhisk.apache.org/.

[42] "OpenWhisk Metric Support". (Feb. 2023), [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/metrics.md.

[43] "OpenWhisk User Events". (Feb. 2023), [Online]. Available: https://github.com/apache/openwhisk/tree/master/core/monitoring/user-events.

[44] "Pandora FMS". (Feb. 2023), [Online]. Available: https://pandorafms.com/en/.

[45] Pedro Rodrigues, Filipe Freitas, and José Simão, "QuickFaaS: Providing Portability and Interoperability between FaaS Platforms", *Future Internet*, vol. 14, 2022.

[46] "pgAdmin Tool". (Sep. 2023), [Online]. Available: https://www.pgadmin.org/.

[47] "PostgreSQL". (Sep. 2023), [Online]. Available: https://www.postgresql.org/.

[48] "Prometheus". (Feb. 2023), [Online]. Available: https://prometheus.io/.

[49] "Prometheus Exporters". (Nov. 2022), [Online]. Available: https://prometheus.io/docs/instrumenting/exporters/.

[50] "Prometheus Go client library". (Dec. 2022), [Online]. Available: https://github.com/prometheus/client_golang.

[51] "Prometheus JVM Client". (Feb. 2023), [Online]. Available: https://github.com/prometheus/client_java.

[52] "Dissertation Solution Repository: Monitoring Resources in Function-as-a-Service Platforms". (Dec. 2023), [Online]. Available: https://github.com/BeatrizGuerreiro/meic2223.

[53] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski, "The rise of serverless computing", *Communications of the ACM*, vol. 62, no. 12, pages 44–54, 2019.

[54] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes, "The spec cloud group's research vision on faas and serverless architectures", in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pages 1–4.

[55] Guilherme Da Cunha Rodrigues, Rodrigo N Calheiros, Vinicius Tavares Guimaraes, Glederson Lessa dos Santos, Marcio Barbosa De Carvalho, Lisandro Zambenedetti Granville, Liane Margarida Rockenbach Tarouco, and Rajkumar Buyya, "Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions", in *Proceedings of the 31st annual ACM symposium on applied computing*, 2016, pages 378–383.

[56] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, *et al.*, "Serverless computing: Current trends and open problems", *Research advances in cloud computing*, pages 1–20, 2017.

[57] Johannes Manner, Stefan Kolb, and Guido Wirtz, "Troubleshooting Serverless functions: a combined monitoring and debugging approach", *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, pages 99–104, 2019. DOI: 10.1007/s00450-019-00398-6.

[58] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen, "Characterizing serverless platforms with serverlessbench", in *Proceedings of the 11th ACM Symposium on Cloud Computing*, Association for Computing Machinery, 2020, 30–44. DOI: 10.1145/3419111.3421280.

[59] "Introduction to Serverless on Kubernetes". (Dec. 2022), [Online]. Available: https://www.edx.org/course/introduction-to-serverless-on-kubernetes.

[60] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski, "The Rise of Serverless Computing", *Commun. ACM*, vol. 62, no. 12, 44–54,

[61] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup, *Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications*, 2022. DOI: 10.48550/ARXIV.2205.07696.

[62] "Spring Framework". (Sep. 2023), [Online]. Available: https://spring.io/projects/spring-framework.

[63] "Ubuntu Operating System". (Sep. 2023), [Online]. Available: https://multipass.run/.

[64] "YAML Markup Language". (Sep. 2023), [Online]. Available: https://yaml.org/.