UNISINOS

Programa de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Gustavo André Setti Cassel

HIERARCHICAL FOG-CLOUD ARCHITECTURE TO PROCESS PRIORITY-ORIENTED HEALTH SERVICES WITH SERVERLESS COMPUTING

São Leopoldo, 2023

Gustavo André Setti Cassel

HIERARCHICAL FOG-CLOUD ARCHITECTURE TO PROCESS PRIORITY-ORIENTED HEALTH SERVICES WITH SERVERLESS COMPUTING

Master Thesis proposal presented as a partial requirement to obtain the Master's degree by the Applied Computing Graduate Program of the Universidade do Vale do Rio dos Sinos — UNISINOS

Advisor:
Prof. Dr. Rodrigo da Rosa Righi

Co-advisor:
Profa. Dra. Marta Rosecler Bez

São Leopoldo
2023

C344h Cassel, Gustavo André Setti.

Hierarchical fog-cloud architecture to process priority-oriented health services with serverless computing / by Gustavo André Setti Cassel. – 2023.

156 p. : il. ; 30 cm.

Master thesis (master's degree) — Universidade do Vale do Rio dos Sinos, Applied Computing Graduate Program, São Leopoldo, RS, 2023.

Advisor: Dr. Rodrigo da Rosa Righi.

Co-advisor: Dra. Marta Rosecler Bez.

1. Healthcare. 2. Architecture. 3. Internet of things. 4. Priority. 5. Response time. 6. Throughput. 7. Serverless computing. I. Title.

UDC: 004.738.5:614

Cataloging in Publication (CIP):
Librarian Alessandro Dietrich - CRB 10/2338

ABSTRACT

Smart cities and healthcare services have been gaining much attention in recent years, as the benefits provided by this field of research are significant and improve quality of life. Systems can proactively detect health problems by monitoring a person's vital signs and making automated decisions in order to prevent these problems from worsening. Examples include health services sending notifications to the user's smartphone when a health problem is detected, or automatically calling an ambulance when vital signs indicate that a severe problem is about to happen in the next minutes. With this context in mind, we highlight two essential requirements that architectures for smart cities should consider to achieve high quality of experience in the field of health. The first is to execute health services with short response times when ingesting high-priority vital signs, so people with comorbidities can have health problems identified as soon as possible. The second is to employ scalability techniques to deal with high usage peaks caused by people concentrating in specific city neighborhoods. Related works already propose solutions to minimize response time, but we argue that considering the semantics of user priority and service priority in the field of health is essential to ensure the appropriate quality of experience. Our understanding is that users with comorbidities should have more priority than healthy users when computing resources are scarce, and specific health services should have higher priority than others. With this in mind, this thesis contributes to this field of research by proposing SmartVSO - a computational model of a hierarchic, scalable, fog-cloud architecture, which executes health services with optimized execution throughput and minimized response time for critical vital signs. We employ fog computing to achieve short response times and cloud computing to achieve virtually infinite computing resources. A first heuristic favors critical vital signs when disputing for scarce, low-latency resources during high usage peaks. This is encompassed by calculating a ranking for the incoming vital sign, which considers both user and service priorities that semantically represent the vital sign's importance. When vital signs collide with the same calculated ranking, a second heuristic uses forecasting techniques to favor health services that will complete faster, with the goal of optimizing execution throughput. We consider serverless computing as the primary technology for deploying and running health services because this allows authorized third parties to implement their own health services in a distributed and pluggable approach, without recompiling the proposed decision-making modules. Finally, we introduce a recursive mechanism that offloads vital signs to parent fog nodes when local computing resources are overloaded, until the vital sign can be processed on a fog node with available computing resources, or is offloaded to the cloud as the last resort. An experiment with 80.000 vital signs indicates that our solution processes 60% of critical vital signs in no more than 5,3 seconds, while a naive architecture that does not employ fog computing and does not favor critical vital signs takes up to 231 minutes (around 3 hours and 51 minutes) to process 60% of critical vital signs.

Keywords: Healthcare. Architecture. Internet of Things. Priority. Response time. Throughput. Serverless computing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
DRL	Deep Reinforcement Learning
EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
FaaS	Function as a Service
FCFS	First-Come-First-Serve
HES	Holt Exponential Smoothing
IoT	Internet of Things
IoMT	Internet of Medical Things
JVM	Java Virtual Machine
LPO	Layout and Placement Optimizer
QoS	Quality of Service
RTT	Round Trip Time
S3	Simple Storage Service
SmartVSO	Smart Vital Sign Offloading
SBC	Single Board Computer
SDE	Serverless Deployment Engine
SLA	Service Level Agreement
SLO	Service Level Objectives
SOA	Service Oriented Architecture
SPP	Service Placement Problem
SQS	Simple Queue Service
SSR	Serverless Service Request
TCP	Transmission Control Protocol
TS	Telemetry Service
UI	User Interface

CONTENTS

1 INTRODUCTION

Computing for healthcare applications has become increasingly important over the past few years, with a large number of scientific studies proposing solutions in this area. The Internet of Medical Things (IoMT) has greatly influenced this evolution: in this context, wearable devices continuously collect human vital signs, which health systems monitor in real-time with a preventive approach. People can achieve a better quality of life with systems that automatically send notifications or call an ambulance before the health problem effectively happens and worsens, for example. Another interesting use case for computing in the field of health is home healthcare, where patients can be monitored at home and computer systems perform intelligent actions when vital signs drop a certain threshold (HAGHI KASHANI et al., 2021; HARTMANN; HASHMI; IMRAN, 2022; BUYYA; SRIRAMA, 2019; HABIBI et al., 2020).

Along with the increasing number of health services, advanced computing techniques must be employed to deal with requirements appearing in this area. Health services have become critical because they can be part of people's daily lives. Therefore, achieving short response times is essential when dealing with people in critical conditions, as delayed responses are unacceptable for real-time monitoring. In addition, scalability is another essential requirement, especially when dealing with unexpected workloads and a large number of people using health services in a smart city. It is also important that health services are reliable and do not fail, providing a positive and engaging experience. Data privacy is also a concern when designing systems for the field of health, as sending sensitive vital signs across the network can be risky (SANTOS; MONTEIRO; ENDO, 2020; HAGHI KASHANI et al., 2021; ZHANG; NAVIMIPOUR, 2022; HARTMANN; HASHMI; IMRAN, 2022).

With this context in mind, this thesis proposes SmartVSO (Smart Vital Sign Offloading), a computational model of a scalable architecture for running health services with high quality of experience. This model considers priorities for users and health services, so critical vital signs from users with comorbidities are processed on physically closer computing resources, resulting in shorter response times. Strategies to achieve scalability are also employed in a transparent and automatic manner, in a way that this model can be used in smart cities with a large number of users and still handle unexpected workloads. Finally, SmartVSO improves the experience for software engineers, who can easily create and deploy health services as serverless functions.

1.1 Motivation

Different surveys related to Internet of Medical Things have been published in the past few years, indicating interesting challenges and issues in this field of research and elucidating contributions that can be addressed in future work. For example, the survey introduced by Gasmi et al. (2022) indicates that minimizing communication delay in healthcare applications is especially important during emergencies. Fog computing can be used in this scenario because it brings

processing efforts to locations physically closer to the end user, which reduces communication latency. The authors of this survey also indicate that priorities are essential in fog computing because computing resources on the fog are limited. Therefore, selecting which tasks should execute on fog nodes and which should execute elsewhere is essential. This is especially useful when resources are getting overloaded, while only critical tasks should be executed on these valuable, limited computing resources that provide short response time.

In turn, the survey introduced by Haghi Kashani et al. (2021) brings scalability as an important challenge for IoT in the field of health. The authors indicate that health systems must continue working regardless of the amount of data that needs to be processed, such as an increase in the number of people using health systems that will lead to more vital signs being processed. Interoperability is also an interesting challenge, as different standards exist for exchanging vital signs between sensors and computing nodes. In addition, the authors highlight the need for high quality of service in healthcare systems, mainly because this field of research deals with life-critical services that need to be processed promptly. Authors complement that fog computing is fundamental to providing short response times in emergencies, where each second matters to save a person's life. With this context in mind, minimizing the amount of data transferred to the cloud is another essential aspect to be considered as an attempt to reduce response time. Nonetheless, energy consumption and data privacy are also important topics to be addressed. Other studies reiterate some important topics to be studied, such as quality of service, power consumption, response time, and scalable solutions to deal with thousands of users (SANTOS; MONTEIRO; ENDO, 2020; YANG et al., 2022; ZHANG; NAVIMIPOUR, 2022; HARTMANN; HASHMI; IMRAN, 2022). Finally, the survey introduced by Cassel et al. (2022) indicates serverless computing as a promising technology to create, deploy, and execute services for Internet of Things within a sandboxed and isolated environment. This technology has been increasingly used over the past few years in the field of IoT, while healthcare applications are some of the use cases of serverless computing.

Although some of these contributions have already been addressed by related works to some extent, there is still a lack of a scalable model that processes critical vital signs with short response time considering user and service priorities with semantics in the field of health, as well as providing a straightforward approach for software engineers to create and deploy health services. The opportunity to address the challenges mentioned above in the field of health, combined with a modern environment that software engineers can use to create health services with serverless computing, is really promising and can lead to several advances in this area. Services can benefit from short response times and virtually infinite computing resources by combining fog and cloud computing, therefore dealing with high usage peaks and achieving the expected quality of experience for end-users. In addition, serverless functions can be easily deployed to the serverless platform and run in an isolated fashion, which prevents specific health services from messing up with the underlying shared infrastructure.

1.2 Research question

Existing models for executing health services have limitations regarding user and service priorities, response time minimization, and throughput maximization. The following statement defines the main problem this work addresses: *given a distributed architecture to execute health services in a smart city, response time for high-priority vital signs is not effectively minimized during high usage peaks.* In addition, the following items complement the problem that is being solved:

- Existing solutions do not combine user and service priorities with semantics in the field of healthcare to favor critical vital signs when computing resources are overloaded;
- Existing solutions in the field of health do not ingest vital signs in fog nodes arranged in a hierarchical manner, by recursively offloading vital signs and allowing extra computing capacity to be easily added to the architecture as needed.

The following question will be answered throughout this thesis: *how could be a computational model for running health services with short response time and high quality of experience by considering user and service priorities and dealing with high usage peaks in a transparent manner?*

1.3 Goals

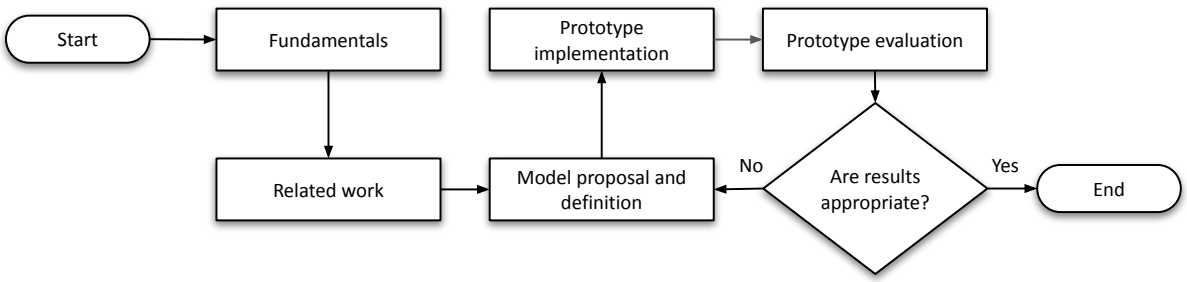
The main goal of this work is *propose a* ***computational model*** *of a* ***scalable architecture*** *to execute* ***health services*** *that integrates* ***fog and cloud computing*** *with* ***user and service priorities****, focusing on* ***short response time****,* ***high throughput****, and* ***quality of experience*** *for end-users*. Specific goals are also defined to achieve this main goal:

- Propose a mechanism that recursively offloads vital signs to the upper layer (parent fog node or cloud) of the hierarchy when local resources are overloaded or vital signs have low priority;
- Efficiently manage limited computing resources on the fog layer by favoring users and services with higher priority during high usage peaks;
- Reduce response time when running health services by proposing heuristics that favor critical vital signs, and favor health services that complete faster by using forecasting techniques based on historical durations when vital signs collide with the same priority;
- Implement prototypes that are used to evaluate the SmartVSO model, as well as collect metrics of response time, throughput, offloading operations, and used CPU during tests.

In summary, our goal is to combine fog and cloud layers to run time-sensitive, distributed health services with high throughput and short response time for high-priority vital signs. This work describes a computational model named SmartVSO that runs different health services with vital signs collected by wearable devices, such as smartwatches collecting body temperature, heartbeat, and oxygen level, among others. This model considers parallel execution of a wide range of health services while transparently managing underlying infrastructure and maximizing execution throughput for critical vital signs. SmartVSO has a module that chooses the most appropriate layer (local fog node or parent fog node) to execute health services, at the same time the cloud is employed as a last option when resources on the fog are overloaded, so both quality of experience for end-users and scalability are guaranteed. The idea is to take advantage of low-latency responses offered by the fog and virtually infinite computing capabilities provided by the cloud, which allows the execution of critical services with the lowest possible response time, while continuing to work during high usage peaks. Conditions to select between processing the vital sign on the local fog node or offloading it to the upper layer include how important a vital sign is and how much CPU is currently available on the fog node. Finally, forecasting techniques are employed to improve throughput when incoming vital signs collide with the same priority. This allows health services that complete faster to execute locally and avoid the extra network latency of being offloaded to the upper layer of the hierarchy.

1.4 Research development stages

Figure 1: Flowchart of main development steps.



Source: prepared by the author.

Figure 1 presents the main steps for proposing this thesis and implementing the model prototypes. The first step is to study fundamental concepts, which are important to deepen our knowledge in the field of interest, including fog and serverless computing. This is followed by a search for related work in the fields of health and computational offloading, which helps us understand what existing works already propose and what gaps exist in this field of research. Once we have this information, we propose a hierarchic fog-cloud architecture for the execution of health services with vital signs of different priorities. Once the model is formulated and presented, a development cycle starts. This cycle is composed of implementing the prototype,

evaluating metrics and results, and reformulating details of the model when results are not satisfactory. Finally, the cycle ends when the prototype has been thoroughly evaluated and the results are satisfactory.

1.5 Text organization

This thesis is organized as follows. The current chapter has contextualized the reader on the field of healthcare and presented what this thesis aims to achieve. This is followed by Chapter 2, which presents fundamental concepts regarding fog and serverless computing. In addition, Chapter 3 presents existing works in the field of computational offloading and service placement, highlighting topics that researchers have been proposing over the past few years and indicating existing gaps in the literature. Chapter 4 is the core of this thesis and shows how the SmartVSO model works, along with a hierarchical fog-cloud architecture to process health services with user and service priorities. This is followed by Chapter 5, which presents the evaluation methodology with scenarios we considered to run the experiments. Chapter 6 presents the results after running the experiments with different scenarios and different variations of the architecture, as well as indicates scenarios where the architecture performs well and scenarios where it does not. Finally, Chapter 7 concludes this thesis by summarizing what has been presented so far and indicating contributions, limitations, and suggestions for future work.

2 BACKGROUND

This chapter presents fundamental concepts that support the SmartVSO model proposed in this thesis, including fog and serverless computing. The former provides short response times because of its physical proximity to the end user, which is essential for our proposed architecture in the health field. The latter, in turn, is the primary technology we employ for executing health services on fog and cloud layers and will be presented in detail.

2.1 Fog computing

Fog computing is an intermediate layer between physical devices and the cloud, which together represent a three-tier hierarchy (SHEIKH SOFLA et al., 2021). This kind of computing is an important layer to support the concept of Internet of Things because it employs computing capabilities physically close to end devices. Therefore, it provides a shorter response time when compared to sending data to the cloud or executing tasks on the cloud. Fog computing has been a research trend from various perspectives during the past few years and can be understood as an extension of edge and cloud computing. However, we emphasize that fog is not meant to replace cloud or edge devices because all of them have pros and cons and can be more appropriate according to the use case. Fog complements their features by working together with other computing layers in a cooperative and orchestrated manner, so it is important to interplay between layers. Specific tasks should be executed on the fog and others on the cloud, for example, according to use-case requirements. Examples include time-sensitive services that require short response times, which are appropriate for running in the fog, and resource-intensive services for big data analytics tasks that are appropriate for the cloud (BUYYA; SRIRAMA, 2019; HABIBI et al., 2020; GASMI et al., 2022).

Fog computing is also promising in the field of health because many healthcare applications are time-sensitive and require real-time results, with the lowest possible delay. The network latency is reduced when compared with the cloud because of the short physical distance. Executing health services physically close to the person being monitored will make him or her receive notifications about the health status promptly, which is especially important for critical services. People in a hospital, for example, need to be closely monitored and receive responses in real-time in case they have a problem that requires extreme attention (GASMI et al., 2022; HARTMANN; HASHMI; IMRAN, 2022; PAREEK; TIWARI; BHATNAGAR, 2021).

In addition, several IoT devices such as wearables and specific smartwatches are resource-constrained, so they cannot execute health services locally with the vital signs they have just collected (PAREEK; TIWARI; BHATNAGAR, 2021). This data needs to be offloaded and processed elsewhere. However, sending this data to the cloud can be impractical for time-sensitive services, such as critical healthcare applications for people in emergency conditions who cannot tolerate delayed responses. Fog nodes can be helpful in this situation by reducing response time

while being physically and strategically placed between edge devices and cloud providers. In addition, another benefit of fog computing is that it reduces problems of intermittent connectivity between edge devices and the cloud, as the latter is physically distant and is more feasible to have communication problems. In practice, communication problems can also happen between edge devices and fog nodes, but chances are reduced because the physical distance is shorter (BUYYA; SRIRAMA, 2019; HABIBI et al., 2020).

Fog nodes are also heterogeneous in a way that different machines can be found in the same fog computing layer. Examples include high-end servers and commodity hardware, which may run specific operating systems depending on their computing capabilities. This also means that not all health services could run on a specific fog node, as it depends on how powerful the fog node is and how many computing resources a given health service requires. It is also interesting to analyze the most appropriate layer to run a specific service, including edge, fog, and cloud layers. Specific lightweight tasks may be more relevant to run on the edge, while heavier tasks may be more appropriate to run on the fog or the cloud. A concrete example is situated in the context of smart cities, where fog nodes make real-time decisions, and complex, computational-heavy decisions are made by tasks executed on the cloud. However, no rule or limitation says that big data analysis could not be conducted on the fog, for example. It depends on the task that is being executed and also depends on latency requirements for the specific use case. Each situation must be analyzed independently to decide which computing layer best fits each scenario (HABIBI et al., 2020).

Finally, fog computing can also work in a distributed and possibly cooperative manner. The fog layer may be composed of several interconnected fog nodes physically located in different places. Examples include fog nodes in different neighborhoods of a smart city or even in other cities, depending on how the architecture is organized. It is also interesting that fog nodes may work together to process tasks toward the same final goal. An example would be a complex computational problem divided into several tasks and processed in parallel by different fog nodes, to achieve results faster. However, the service should be developed with this mindset of a distributed architecture, to take the most out of distributed machines. The feasibility of this approach also depends on how the connection between fog nodes has been arranged in the smart city. Finally, the fog node where a given service should be executed varies according to the use case: an edge device may send data to the physically closest fog node or may decide to send data to a distant fog node with enough computing capabilities, in case the former is overloaded, for example (BUYYA; SRIRAMA, 2019; HABIBI et al., 2020).

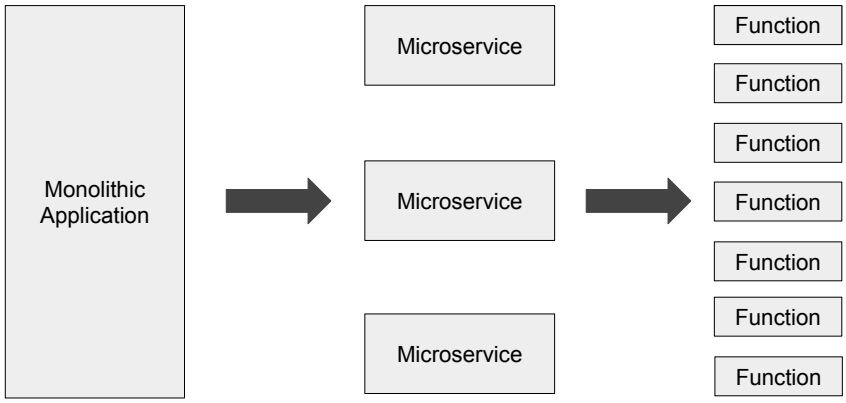
2.2 Function as a Service

Function as a Service (FaaS), sometimes known as serverless computing, represents a trend where applications are built as a composition of stateless functions. This does not mean that services run without servers, though. In turn, it means that software engineers can focus on

business logic instead of spending time managing servers, which is automatically done by the cloud vendor. We emphasize that each function needs to be deployed to a serverless platform, which is responsible for spawning and deactivating function replicas according to the number of requests that are arriving. When executing function requests, the underlying infrastructure managed by the serverless platform of the cloud vendor creates as many instances of the function as needed, to appropriately handle the number of incoming requests. This is done with advanced autoscaling algorithms that are transparent to software engineers focused on deploying and running functions, therefore resulting in increased productivity. In summary, serverless platforms are suitable for running short-term tasks characterized by bursty, intense, sudden, and parallel workloads (CHOWHAN, 2018; KATZER, 2020; KRATZKE, 2018; BALDINI et al., 2017).

A comparison between monolithic applications, microservices, and serverless functions is shown in Figure 2. Monoliths are larger than microservices and functions, as most application logic runs in a single process in the operating system. This characteristic makes it difficult to achieve the desired level of scalability. In addition, monoliths employ a longer initialization time because of the large number of resources that need to be allocated. Monoliths are also usually long-running applications, since they are heavier to load. It would be inappropriate to intermittently start and finish the execution of a monolith in a short time, for example. Microservices, in turn, are located between monolithic applications and serverless functions. They are considerably thinner and smaller than monoliths, making them easier to scale compared to the latter. Microservices are usually long-running applications as well. In turn, serverless functions are short-lived, small pieces of code that can be fired and terminated as much as needed, with a rigid life cycle and typically a single responsibility (CHOWHAN, 2018).

Figure 2: Comparison between monolith, microservices, and serverless functions.



Source: adapted from Chowhan (2018).

Another attractive characteristic of this technology is that serverless computing is suitable for unpredictable workloads and unpredictable usage patterns. As previously stated, serverless

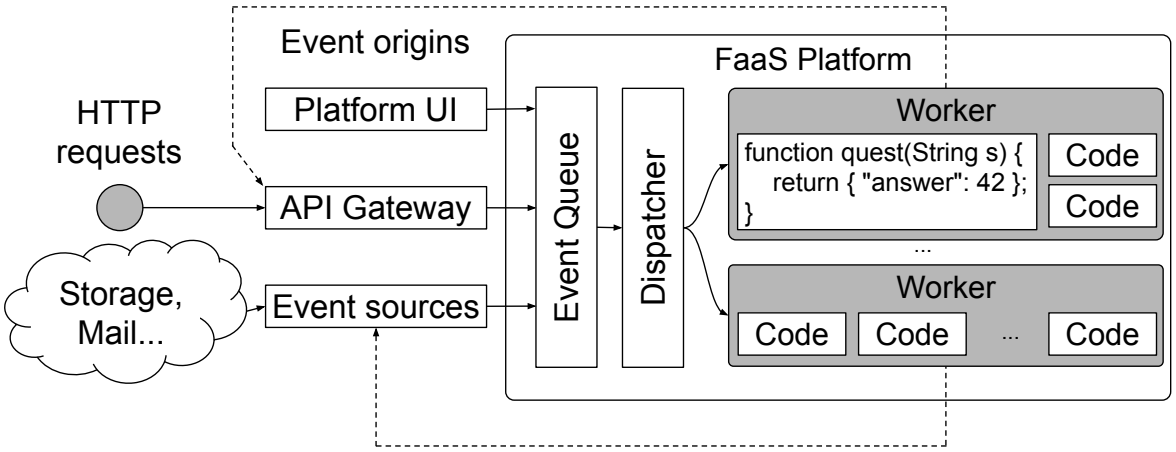
platforms automatically scale resources to meet the increase in incoming requests, which depends on several factors. These factors include the hour of the day, the day of the week, special dates, events, among others. Since serverless functions are typically small and lightweight, they can be instantiated with little effort when compared to monoliths and microservices. This characteristic is an important requirement for serverless computing to achieve the desired level of scalability in a lightweight and automatic fashion. In addition, serverless computing is excellent for event-driven architectures, as the occurrence of an event can trigger the execution of different functions, as well as it is based on principles of Service Oriented Architecture (SOA). This allows applying the same principles of microservices decoupling, as each function is an autonomous, granular, and independent piece of code that can be separately deployed and executed (KATZER, 2020; SBARSKI, 2017).

Several cloud vendors offer solutions for serverless computing, with some of the main vendors presented as follows: Amazon Web Services (AWS) offers AWS Lambda[1], Microsoft offers Azure Functions [2], Google offers Google Cloud Functions[3], and IBM offers IBM Cloud Functions[4]. This list is not exhaustive, though, as other vendors also exist. Each vendor integrates its serverless platform with its cloud ecosystem, as well as charges different prices and provides specific features for developers. In addition, it is suggested that users write single-purpose, stateless functions to comply with the auto-scaling capabilities provided by the serverless platform. It also makes testing easier and can lead to fewer bugs and unexpected side effects. Finally, engineers should not assume that the function will execute on the same sandboxed environment during the next invocations, which is why they should not rely on storing data on global variables to be used on further executions. Sometimes, the same container is used for further executions of the same function, which is known as the warm start, but this is not guaranteed (SBARSKI, 2017).

Figure 3 presents an overview of the main components that exist in serverless platforms. In practice, each platform may have its specific internal components, but the most common are presented as follows. A function request may also be originated from different sources, including the User Interface (UI) provided by the cloud vendor, the API gateway to directly invoke the function via HTTP requests, and different event source connectors. The platform UI can be a web page offered by the cloud vendor, where functions can be manually executed after specifying the payload on the web page, for example. Also, the API gateway is a common approach to executing serverless functions programmatically. Client applications can send HTTP requests to the endpoint provided by the cloud vendor, which is specific to the function. The request payload and additional request parameters (such as headers) are automatically collected by the serverless platform and passed as the argument to the function input during its invocation. Another approach to invoke a serverless function is to connect the serverless platform to a set

[1] https://aws.amazon.com/lambda
[2] https://azure.microsoft.com/services/functions/
[3] https://cloud.google.com/functions
[4] https://ibm.com/cloud/functions

Figure 3: Main internal components of a serverless platform.



Source: Cassel et al. (2022), adapted from Kratzke (2018) and Baldini (2017).

of event sources. This figure presents data being stored in the cloud as a source for triggering a function, as well as the reception of an email could automatically trigger another function. In other words, a function is automatically invoked by the cloud ecosystem whenever data is stored on a database, while passing the stored data as the function input, or information about the email could also be automatically passed as the function input. In summary, event sources are usually integrated with other tools provided by the cloud vendor to offer an easy and positive developer experience (CASSEL et al., 2022; KRATZKE, 2018; BALDINI et al., 2017). Finally, the Event Queue is the module responsible for receiving the execution request that comes from different sources. This queue is integrated with a Dispatcher component, which consumes the queue and effectively creates worker instances to execute the function code. A worker instance may invoke the function as a container, for example, but each serverless platform can employ a specific technology that best fits its needs. It is interesting to notice that workers are also connected to the API gateway and to the event sources, which means that when a function executes it may also trigger the execution of other functions, or even trigger the execution of the same function recursively (CASSEL et al., 2022; KRATZKE, 2018; BALDINI et al., 2017).

Although originally proposed for the cloud, serverless platforms can also be employed on the fog layer. This perfectly fits IoT applications and allows software engineers to benefit from serverless computing while taking advantage of short response times due to the short physical distance between users and the physical machines. An example of a benefit is the ease of creating functions focusing on business code, which in the context of this thesis is processing vital signs collected from smartwatches. Supposing that several health services need to be implemented, several functions can be created, as each function represents a different health service. Functions can also be automatically scaled according to the number of requests; however, they are limited by the computing capabilities provided by resources on the fog. An

example of a serverless platform that can be used on the fog is OpenFaaS[5], but other platforms also exist. We highlight that multiple machines on the fog can work together as a cluster of worker machines responsible for executing function requests, although in this thesis we assume that each fog node is composed of a single machine for simplicity purposes (CASSEL et al., 2022).

2.3 Partial considerations

This chapter presented fundamental concepts necessary to comprehend the SmartVSO model presented in this thesis, while fog and serverless computing are presented as the most important concepts for our hierarchic architecture in the health field. The former results in shorter response times because fog nodes are physically close to end-users and edge devices. Therefore, our architecture will be able to process vital signs and execute health service requests based on this data with short response times. This means that people who are having their vital signs monitored will receive notifications as soon as possible, which is especially important for people in emergencies.

In addition to fog computing, we present concepts about serverless computing because it is the primary technology we employ in our architecture to execute health service requests. Serverless computing provides intrinsic scalability mechanisms, which are abstracted from the viewpoint of the software engineer. This makes it easier to scale the execution of health services for many people in a smart city. In addition, this technology seems quite promising because it provides a sandboxed environment to run services, which makes it difficult for malicious code to mess with other health services being executed. In addition, serverless computing is promising because of its simplicity in creating services. The current chapter is followed by Chapter 3, which will present related works in the field of computational offloading and service placement using the concepts presented in the current chapter.

[5]https://www.openfaas.com/

3 RELATED WORK

This chapter presents existing works addressing computational offloading and Service Placement Problem (SPP). They are very important when proposing the SmartVSO model because it contains a hierarchical architecture that considers fog and cloud in the field of health. Both subjects are grouped into papers that do not address and address serverless computing. Existing techniques to efficiently combine resources on fog and cloud layers are also explored. In addition, we provide information on whether articles address service priorities and how the available resources on the fog layer are monitored. Finally, this analysis also brings existing limitations and gaps that can be addressed in our current work or can be addressed in the future.

3.1 Search methodology

This section details the methodology employed to search for related works. Although this search was not performed systematically, Figure 4 presents the search string used to search for studies. At the same time, different spellings were considered as suggested by Kitchenham and Charters (2007). Both *iot* and *internet of things* represent the same meaning and were grouped with an OR operator. Keywords *computational offloading* and *service placement* were also considered, as both are closely related to the task placement issue we address on the field of health, as well as *cloud*, *fog*, *edge*, *low-latency*, and *low latency*. Google Scholar[1] was mainly used to search for papers as it consolidates papers from different sources in a single place. Only studies from 2018 onwards were considered to better filter results considering recent studies. In addition, interesting papers cited as related works on other papers were also considered in the analysis and were added to our list of related works.

Figure 4: First search string employed in Google Scholar to collect papers. This string addresses computational offloading and service placement for IoT without serverless computing.

```
("iot" OR "internet of things") AND ("edge" OR "fog") AND "cloud" AND
("computational offloading" OR "service placement") AND ("low-latency" OR
                            "low latency")
```

Source: prepared by the author.

Figure 5 presents an additional search string employed to collect papers, which contains the same keywords as Figure 4 but also filters terms related to serverless computing in general. Examples include *serverless computing*, *function as a service*, and *function-as-a-service*, as it is the main technology we address in our work.

[1] https://scholar.google.com

Figure 5: Second search string employed in Google Scholar. This string also addresses computational offloading and service placement for IoT but considers serverless computing as well.

```
("iot" OR "internet of things") AND ("edge" OR "fog") AND "cloud" AND
("computational offloading" OR "service placement") AND ("low-latency" OR
"low latency") AND ("serverless computing" OR "function as a service" OR
                        "function-as-a-service")
```

Source: prepared by the author.

3.2 Work analysis

This section presents an analysis of the selected related works in the field of computational offloading. Results are grouped into two main groups. The first focuses on papers that address computational offloading techniques without serverless computing, while the second also addresses this same subject but focuses on serverless computing. The following subsections highlight the pros and cons of each group, presenting relevant characteristics that serve as a basis for proposing new solutions and filling some of the existing research gaps. The characteristics that were analyzed on each paper are stated as follows:

- Metrics: information that is monitored on fog nodes to make offloading decisions, such as CPU, memory, size, disk, bandwidth, and response time, among others;
- Context: the scenario in which the work is situated (if it is specific to a given field of research, such as health, or if it can be employed regardless of context);
- Goal: main focus addressed by the paper;
- Layers: computational layers (edge, fog, or cloud) where services are executed and the paper is focused on;
- Priority: whether or not the work considers priority while executing requests, such as priority for specific users or for services that require shorter response times.

Some papers present strategies with serverless computing as the primary technology, dealing with offloading and placement of serverless functions on different devices and layers. Other papers, in turn, introduce generic algorithms that can be employed no matter the technology.

3.2.1 Offloading without serverless computing

This subsection, along with Table 1, presents technology-agnostic studies that are not related to serverless computing. These papers typically introduce algorithms and techniques for task offloading operations in the field of IoT. Some studies also focus on energy efficiency by selecting the most appropriate location to run a given service based on how much it will cost,

even though it may not result in the best performance. There is an interesting trade-off between cost and performance.

Table 1: Related work on computational offloading without serverless computing.

Paper	Metrics	Context	Goal	Layers	Priority?
(ALZAILAA et al., 2021)	CPU	Health	Improve latency for critical tasks	Fog, Cloud	✓
(REZAZADEH; REZAEI; NICKRAY, 2019)	CPU, memory, bandwidth	Health	Select the most appropriate fog node based on its available resources and latency	Fog, Cloud	-
(ARORA; SINGH, 2021)	CPU, memory, size, bandwidth	General	Distribute heterogeneous services on heterogeneous fog nodes	Fog	✓
(HASSAN; AZIZI; SHOJAFAR, 2020)	CPU, memory, deadline, size	General	Maximize QoS and minimize energy consumption	Fog, Cloud	✓
(BUKHARI et al., 2022)	CPU, memory, size, bandwidth, burst time	General	Maximize both QoS and resource usage with logistic regression	Fog, Cloud	✓

Source: prepared by the author.

The work proposed by Alzailaa et al. (2021) introduces an algorithm to schedule tasks in the field of e-health across fog and cloud nodes. Tasks are classified into critical or normal priorities according to semantic characteristics identified on the payload while also estimating CPU time based on this analyzed information. Authors suggest that critical tasks are executed on the fog layer due to the lower latency it employs, while normal tasks could be processed on the cloud. Estimated CPU time is compared to thresholds that indicate whether executions should be forwarded to fog or cloud. It is important to highlight that the scheduler considers multiple fog nodes, which chooses the most appropriate fog node to execute incoming requests instead of simply running them on arbitrary nodes. This work is focused on CPU time and does not consider other metrics such as disk, memory, network, and general I/O usage. Evaluation is performed with four virtual machines representing fog nodes and a powerful virtual machine representing the cloud node. Almost all tasks execute with a waiting time of less than 0.2ms with the proposed algorithm, and 50% execute with a waiting time of less than 0.15ms. A very interesting point is that this scheduler favors critical tasks by reducing their waiting time, at the same time it does not hurt non-critical tasks, which maintain a similar waiting time when compared to traditional First-Come-First-Serve (FCFS) scheduling.

In its turn, the work proposed by Rezazadeh, Rezaei, and Nickray (2019) introduces a latency-aware algorithm called LAMP that selects the most appropriate fog node according to the available resources and the delay the fog node employs. In practice, the delay is directly related to the physical distance between the client and fog nodes, while a lower delay is better and makes a given node more feasible for selection. Only fog nodes with enough available resources are selected to execute services. Authors use iFogSim to analyze results and evaluate them by simulating the behavior of resources on the fog layer (GUPTA et al., 2016). Although their work is focused on fog nodes, LAMP can also place services on the cloud when no fog node is suitable for placement. Regarding resource usage, the paper does not clearly define all

metrics used, although CPU, memory, and bandwidth are stated.

The work proposed by Arora and Singh (2021) is focused on heterogeneity in the fog layer. Authors assume that resources on the fog nodes and computing capabilities required by services are heterogeneous. Specific services may require more CPU while others may require less CPU but more disk, for example. Their work proposes an algorithm that sorts heterogeneous services and fog nodes based on the resources these services need (RAM, MIPS, size, and bandwidth) and the resources the fog nodes provide. Their work also organizes services into critical or normal priority. Evaluation is performed with iFogSim (GUPTA et al., 2016), while all settings are manually specified in the simulator and the latency the fog nodes employ to each other. In summary, their work is focused on the heterogeneity of both services and fog nodes and tries to maximize the usage of resources. The work proposed by these authors does not include strategies to offload executions to the cloud when fog is overloaded.

The work proposed by Hassan, Azizi, and Shojafar (2020) has the main goal of maximizing Quality of Service (QoS) and minimizing energy consumption, with two algorithms respectively introduced to this end. This work considers priorities, network latency, and power efficiency of fog nodes. Authors organize services into critical or normal priorities while placing critical services on fog nodes that provide more computing capabilities (to maximize QoS) and normal services on the most energy-efficient nodes (to reduce energy consumption). Their work considers the response time of both local and neighbor fog clusters and the response time of machines on the cloud, while the latter results in higher communication latency but is capable of providing virtually infinite computing resources.

Finally, the work proposed by Bukhari et al. (2022) introduces a task-offloading algorithm to decide between fog and cloud layers to improve QoS and resource utilization. Their work is not limited to horizontal offloading between fog nodes but also considers vertical offloading when fog is overloaded and unable to process incoming requests. A decision-maker module located on the fog is responsible for interacting with a master node, which periodically takes snapshots regarding the status of the fog layer. The proposed algorithm is based on logistic regression on a dataset with 34 characteristics and goes through training, testing, and validation. A pre-processing phase removes outliers and rows with missing values from the dataset and performs specific transformations to ensure all features are numeric. The output of this pre-processing phase is forwarded to the learning process.

3.2.2 Offloading with serverless computing

This subsection presents papers that address computational offloading with serverless computing as the main technology, which is also the focus of the current work. Table 2 presents the most relevant information about these papers. It is possible to see papers focusing on specific computational layers during the decision-making process: some focus on horizontal offloading across devices on the edge, while others focus on horizontal offloading but on the fog layer in-

stead. Others, in turn, focus on vertical offloading from lower to upper layers, such as fog-cloud interaction. Regarding serverless computing, specific characteristics need to be addressed, such as cold start and the amount of time a given node takes to download function images from a remote registry. This subsection will also present how related studies deal with these complexities.

Table 2: Related work on computational offloading with serverless computing.

Paper	Metrics	Context	Goal	Layers	Priority?
(CHENG et al., 2019)	Not specified	General / Smart parking	Reduce response time and latency	Fog, Cloud	✓
(PINTO; DIAS; SERENO FERREIRA, 2018)	Historical duration	General	Reduce response time	Fog, Cloud	-
(PELLE et al., 2021)	Network latency, CPU, memory, execution time, application latency, invocation rates	General / Robotics	Optimize deployment of latency-sensitive applications with 5G network telemetry	Edge, Cloud	-
(BERMBACH et al., 2020)	CPU, disk	General	Scalability	Edge, Fog, Cloud	-
(GEORGE et al., 2020)	Latency, historical duration	General	Reduce response time, reduce energy consumption	Edge, Cloud	-
(DEHURY et al., 2021)	Latency, function size, required resources, number of users, available resources, bandwidth, etc.	General	Optimal function deployment	Fog, Cloud	✓
(RAUSCH; RASHED; DUSTDAR, 2021)	Execution time, edge resources, network usage, cost	General	Efficient usage of edge infrastructure	Edge	-
(CICCONETTI; CONTI; PASSARELLA, 2021)	Response time	General	Efficient and decentralized distribution of serverless functions	Edge	-

Source: prepared by the author.

The work proposed by Cheng et al. (2019) introduces a model for running IoT services with serverless functions on the fog, called Fog Function, which can also dispatch requests to the cloud when fog resources are overloaded. Details on how vertical offloading to the cloud takes place were not presented in depth. The key difference between this model and other serverless solutions for edge and fog is that this solution proposes a data-centric programming model instead of a topic-based, publish/subscribe so that functions are executed on nodes closer to where data is available. Their model has the following main category of components: *i)* workers; *ii)* brokers; *iii)* orchestrators; and *iv)* discovery. This work is considered proactive rather than reactive because it analyzes fog resources to dispatch requests to the cloud when fog is getting overloaded. Still, it does not specify which metrics are exactly analyzed on the fog.

The work proposed by Pinto, Dias, and Sereno Ferreira (2018) introduces a proxy component that decides whether to offload incoming service requests to the fog or the cloud based on durations collected from previous executions. This work does not consider only network latency but also considers processing time, as functions can be processed faster depending on capabilities provided by cloud nodes, even though cloud computing employs higher latency

when compared to fog. Priorities are not considered in the decision-making process.

The work proposed by Pelle et al. (2021) introduces a model for selecting the most appropriate layer (edge or cloud) to run a serverless function, by performing detailed telemetry of several 5G network characteristics and the status of edge nodes. In addition, alarms are automatically triggered when resource usage is out of acceptable margins to redeploy or reconfigure the serverless function with different settings when different resources are required. This work also considers the specifics of the 5G protocol during telemetry, such as network congestion and other characteristics. This work uses AWS tools such as AWS Lambda and AWS IoT Edge so that both are connected. Functions can run without timeout constraints on edge devices, while functions employ timeout in the cloud, which is an AWS Lambda requirement. Three main components exist in the proposed model: Layout and Placement Optimizer (LPO), Serverless Deployment Engine (SDE), and Telemetry Service (TS).

The work proposed by Bermach et al. (2020) introduces a scheduling algorithm that chooses the most appropriate layer to execute serverless functions with an auction-based decision process. Nodes do their best to run serverless functions and bid according to their available resources, such as storage capacity and CPU. Edge and fog nodes typically make both storage and processing bets, while the cloud is known for always being able to run functions, as this layer provides virtually infinite storage and processing capabilities. Another feature that makes this approach interesting is that it does not require a centralized component to distribute decisions.

The work proposed by George et al. (2020) introduces a Python runtime system called IoTPy for running serverless functions on both edge and IoT devices. This work proposes a single and generic API that allows executing the same function on different devices instead of using specific APIs for each device which could result in vendor lock-in and extra complexity. In addition, two schedulers are presented to choose between edge devices and edge nodes to execute functions. Both schedulers benefit from the latency of previous executions. The first, a naive scheduler, is re-executed iteratively to ensure that decisions do not become outdated, as changes to the network or modifications to the application's source code can affect the results. The second, a more advanced scheduler, computes a score considering a window of the last three latency samples and exponentiation by a particular decay to make the transition between devices smoother.

The work proposed by Dehury et al. (2021) introduces a Deep Reinforcement Learning (DLR) approach to decide where (fog or cloud) a given serverless function should be placed based on a list of different parameters. Their goal is to deploy a minimum number of functions on a fog node that minimizes the communication and computation latency without compromising the QoS parameters. Interestingly, this work considers two types of priority: user priority, where users logged into the system could have higher or lower privileges than others, and function priority, where a given function could be executed before or after the others. Some of the parameters considered for the placement decision are: *i)* latency between fog and cloud; *ii)* size

of the serverless function; *iii)* required resources to run a function, *iv)* number of users logged into the system; *v)* available resources; *vi)* bandwidth, among others. Users can manually define how critical their deployed functions are, which is important when deciding which layer (fog or cloud) these functions should be placed on. This value of how critical a function is ranges from 1 to 5, where 1 represents the lowest priority, and 5 represents the highest priority. Functions with priority 5 should be deployed on the fog layer, while priority 1 should be deployed on the cloud. In addition, functions that are smaller in size and receive smaller inputs are typically deployed on the fog instead of the cloud. This work does not analyze functions independently; instead, users deploy a group of functions known as Serverless Service Request (SSR). The decision process focuses on the group of functions instead of each function independently. Even if a given function has high priority, a small size, and receives small input, it can be treated as having lower priority if other functions in the same SSR have low priority, are large, and receive larger inputs, for example.

The work proposed by Rausch, Rashed, and Dustdar (2021) introduces a Kubernetes scheduler named Skippy, to optimize the placement of serverless functions across edge devices. This scheduler considers hard constraints, such as nodes having less than the required capabilities to execute a function, and soft constraints. The latter represents a score indicating which node is the most suitable when considering latency and distance to the data source, among others. OpenFaaS was used along with Kubernetes. Skippy daemon (also deployed as a container) runs on each node and periodically collects information about them to consider pluggable features like connecting GPU devices to USB ports. This work also introduces an API to annotate the data required by each function, which Skippy uses to select the most suitable node according to the external resources used by the function. The time taken to download the container image from the Docker registry is also considered in the decision, as running a function on a device that already has the image previously downloaded is more efficient.

Finally, the work proposed by Cicconetti, Conti, and Passarella (2021) introduces an approach to perform horizontal offloading of serverless functions at the edge in a distributed and decentralized manner. This layer has no centralized component, so different edge devices run a complete serverless platform instance. When a group of devices cannot process requests, they are offloaded to different devices in the same layer rather than vertically offloading them to the cloud, aiming to get the most out of the edge. This paper differs from others because the authors propose a decentralized approach with no main component, such as a load balancer responsible for receiving and forwarding requests.

3.3 Research opportunities

This section presents research opportunities identified during the analysis of related works. A large number of opportunities was identified, such as priorities, proactive decisions, computational offloading, and adaptive timeout, which are deeply explained throughout this section.

Finally, a summary of key research opportunities is presented, although not all are deeply analyzed.

Priorities: most solutions that address serverless computing do not consider priorities when choosing the most suitable layer (fog or cloud) to run functions, representing an interesting opportunity to explore. Related studies without serverless computing, though, address specific techniques to deal with priorities so that critical services are favored. An example is placing requests on a separate queue to be executed before other requests. An interesting improvement is to combine serverless computing with priorities on the decision between running services on the current fog node, on a different fog node, or on the cloud. Especially, priorities that make sense in the field of health and consider the semantics of this area, such as favoring vital signs of users with comorbidities. This way, requests for serverless functions with higher priority have higher chances to execute on the fog layer, which results in lower response time, although fog has limited capacity. On the other hand, low-priority functions do not need to compete with requests running on the same fog node and could be directly forwarded to distant fog nodes or to the cloud to benefit from virtually infinite computing capabilities. However, response time will be higher and different trade-offs are involved in the decisions. Cloud vendors also bill according to the number of resources used, which makes running functions with short response times on the fog layer an interesting approach.

Proactive decisions: most solutions already employ proactive rather than reactive strategies to make decisions, which is very interesting because the former often yields better results. However, different proactive strategies can be further explored. Proactivity can be achieved by anticipating future needs from historical data so that statistical methods can be applied to forecast requirements and make intelligent decisions to improve overall functionalities. In addition, studies address different metrics to make such decisions, such as the amount of available CPU, RAM, bandwidth, and disk. Other metrics could be introduced or combined to make even more accurate placement decisions. An interesting metric would be the number of services running concurrently because it impacts how long a given service could run with high resource usage on the fog layer without harming other executions. In case prediction tells that no service will arise in the next few minutes, executions on the fog layer could run longer, even when using more resources.

Computational offloading: regarding offloading, papers address horizontal, vertical, or both offloading techniques. It is interesting to deeply investigate the latter by collaboratively using horizontal and vertical approaches to solve low-latency and scalability problems. As serverless computing suggests functions to be stateless, it perfectly matches the offloading concept so that functions can be easily placed on both fog and cloud according to current requirements. In addition, focusing on specific areas such as health can be positive, as solutions can perform semantic analysis in the field of research. Some studies address peculiarities of robotics or health, which can be deeply explored. As our work focuses on health, specific health service characteristics could be addressed to further optimize placement.

Adaptive timeout techniques: existing studies primarily focus on workload distribution across machines on edge and fog layers to maximize resource usage. Still, they do not focus on strategies to deal with the maximum duration that executions should last on these layers. This would prevent services from monopolizing shared resources for an extended period. Some other studies allow users to employ timeout constraints manually, but these constraints are possibly not informed with optimal values, as finding optimal timeout values for each service is challenging. Services must not monopolize computing resources for an extended period, or other services would be unable to run, especially when resources are constrained and a large number of heterogeneous services are running concurrently. Timeout constraints are an attempt to reduce concurrency and allow only short-term, critical services to run in the fog layer. This way, services that last longer could be offloaded to the cloud, as this layer provides virtually infinite resources. Multiple pros and cons can be stated regarding the subject of timeout constraints. Timeout is beneficial because no service will monopolize the fog layer for an extended period. Still, it can also be disadvantageous because long-running computations would not be able to execute on the fog layer. In addition, if the time limit is exceeded, the execution is abruptly finished, and all processing efforts are wasted. The challenge is finding the most appropriate timeout for a given service, as it cannot be so small that the service would never be complete, but it should not be so large that the service could monopolize computing resources for an extended period. In addition, timeout values should vary according to the service because those using a small amount of computing resources could run for a longer time and would still not compromise resources. In contrast, services that use many resources should run during smaller periods because it could affect other services. To the best of our knowledge, existing solutions do not address situations where a service unfairly uses resources and compromises the execution of other services. As a last thought, we believe that no solution introduces algorithms to calculate the maximum duration that service executions could last in the fog layer with serverless computing. These values are often specified by users and are possibly not set to optimal values. Finding optimal values is challenging, as it depends on a set of variables that change in a timely manner, such as available resources and the number of services trying to use these resources in a given period. Optimal values could not be static nor manually specified; in turn, they could be automatically defined by a decision-maker module that analyzes several conditions and makes intelligent decisions.

Finally, many research opportunities can be identified from related work, while the authors suggested most throughout the papers. The following list summarizes key research opportunities that were identified:

- Consider stateful scenarios while running services, as serverless functions are traditionally stateless;

- Improve scheduling algorithms to consider resource availability, power usage, battery life, and other factors regarding function placement decisions;

- Consider priorities to give preferences to critical services and users in urgent situations;
- Address strategies to deal with the trade-off between data and computational movement.

3.4 Partial considerations

This chapter presents an analysis of the literature regarding computational offloading and service placement, which are closely related to the subject of this thesis. As we propose a computational model to process health services hierarchically, it is important to understand how related studies propose solutions to optimize the usage of resources in this scenario. First, generic techniques not related to serverless computing were introduced, such as workload distribution, a common subject in distributed computing. Some of the works are focused on the field of health, which is also the subject we are interested in. We also identified the metrics that related studies address to perform their decisions, while CPU is one of the most common and is used by multiple studies. In addition, most studies integrate fog with cloud computing to benefit from virtually infinite computing capabilities.

On the other hand, the second group of studies encompasses serverless computing as the main technology to run services in their fields of interest. This group is also very important and gives us insights into our model, as serverless computing is the main technology we address. The fact that multiple works address this technology, along with offloading mechanisms, indicates a promising field to study. Regarding computational models for smart cities in the field of health, none of the related studies has yet addressed serverless computing. This is an interesting opportunity to evaluate how this technology will behave in this field of research.

Finally, several research opportunities were identified from the related studies. These opportunities were strongly considered and had a strong influence on SmartVSO, which will be presented in Chapter 4. Considering priorities and giving preference to critical vital signs is one of the most interesting research opportunities we identify, which is also important in the field of health. In addition, optimizing throughput is another good research opportunity. SmartVSO focuses on maximizing throughput for better utilization of resources when vital signs share the same priority. We have also been inspired by the possibility of employing multi-fog environments because our work is situated in the context of smart cities, with several fog nodes distributed across neighborhoods. Finally, we are also inspired by the presented scheduling algorithms, as we propose algorithms to select the appropriate location to process vital signs.

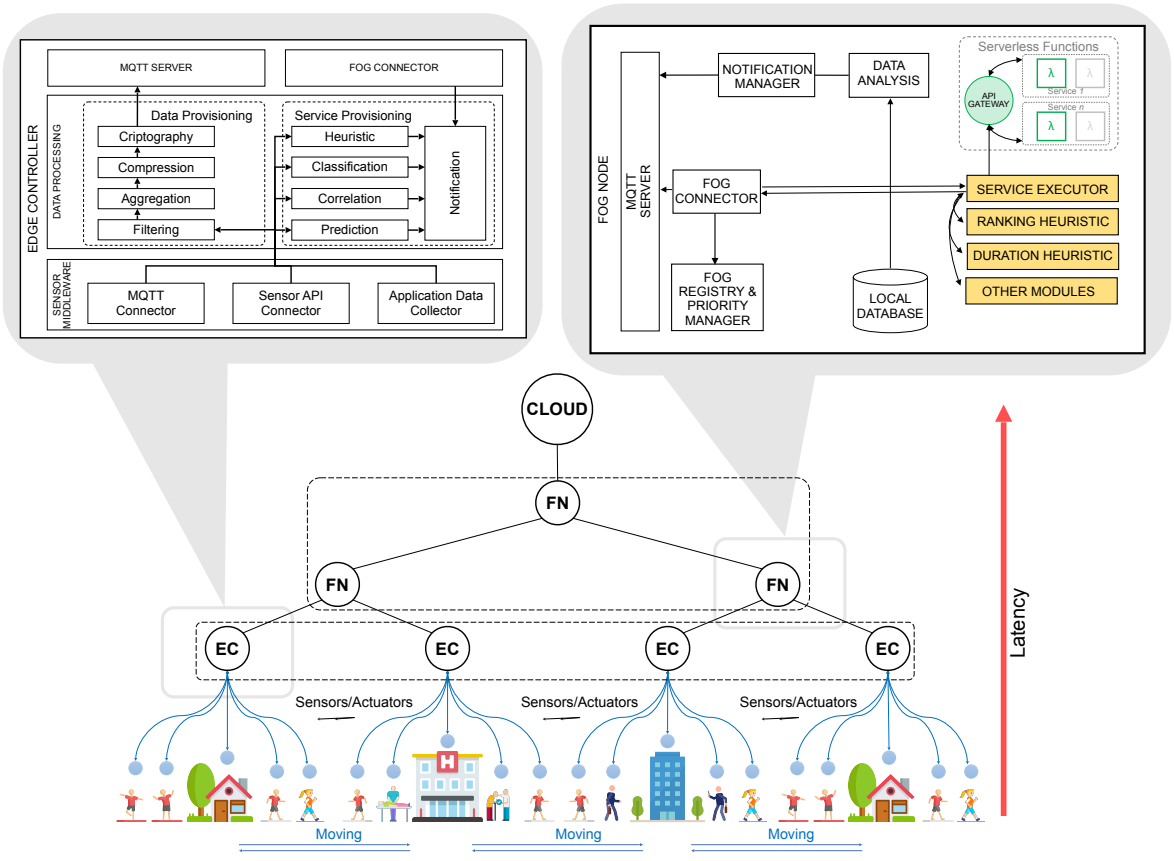
4 SMARTVSO MODEL

This chapter introduces the SmartVSO (Smart Vital Sign Offloading) model for running health services in a hierarchical manner considering fog and cloud with user priority, service priority, and duration prediction. Design decisions, architecture details, modules, algorithms, and forecasting techniques are presented in this chapter, as well as the impact that priorities have on service placement decisions and how internal components interact with each other.

4.1 Contextualization

This thesis is part of a research called MinhaHistoriaDigital[1], which aims to process health services in real-time in smart cities with vital signs collected with smartwatches (RODRIGUES; RIGHI, 2022). The modules proposed in this thesis are presented in Figure 6 and are colored in yellow, which will be deeply detailed in this chapter, while other modules are out of the scope of this thesis and are only presented for contextualization purposes.

Figure 6: Overview of MinhaHistoriaDigital fog-based architecture. The modules proposed in this thesis are colored in yellow on this figure and are part of a bigger architecture.



Source: adapted from Rodrigues and Righi (2022).

[1]FAPERGS/MS/CNPq 08/2020 – PPSUS, Grant number 21/2551-0000118-6, Coord. Rodrigo da Rosa Righi

The social context to remember while reading this thesis is explained as follows. People move between neighborhoods using smartwatches that periodically collect vital signs from the human body, such as the temperature, oxygen level in the blood, and heartbeat level in a given time interval. Services that identify potential health problems should process these vital signs in real time. In other words, the health service receives vital signs, detects a possible heart failure, and automatically calls an ambulance or sends a notification to a close relative reporting this situation. This has the social benefit of improving the quality of life in smart cities. There are some challenges to achieving this, such as processing a large number of vital signs in real time but having limited computing resources available. With this in mind, in summary, this thesis introduces mechanisms to favor the processing of vital signs for people in critical conditions. The effect is that people in critical health conditions will get fast responses most of the time, while people in healthy conditions may get responses with more extended time intervals.

The MinhaHistoriaDigital project encompasses a fog-based architecture to process vital signs in smart cities, acting on edge, fog, and cloud layers. The edge controllers are responsible for aggregating vital signs collected from sensors and wearable devices, such as smartwatches, and performing lightweight pre-processing on the data, such as filtering and compressing. The fog nodes are responsible for receiving vital signs from the edge controllers and executing health services based on the received data. Finally, the cloud provides virtually infinite computing capabilities when fog nodes cannot handle large workloads. Users move through the city and therefore the vital signs are sent to a fog node that is physically close to the person, resulting in lower response time due to the shorter physical distance. For the sake of simplicity, this thesis will assume that smartwatches directly send vital signs to the fog nodes. At the same time, in MinhaHistoriaDigital project, additional middlewares may be located at the edge of the network that are responsible for filtering and aggregating data. Also, the algorithms responsible for discovering which fog node is the physically closest to the person are out of scope for this thesis. We assume that smartwatches will always send vital signs to the closest fog node. Finally, a wide range of scenarios may exist in a smart city, but we assume specific conditions where SmartVSO model is expected to be situated and to work correctly:

- Fog nodes are geographically distributed in neighborhoods of the city, but are static and cannot be moved among neighborhoods once they are configured;
- Each fog node is connected to its parent fog node on the hierarchy and none of them is connected to multiple parents at the same time;
- We work with multiple layers of fog computing to introduce an architecture in the form of a tree, but we consider a single cloud provider which is used to process vital signs asynchronously when fog nodes are overloaded;
- Response time increases as the physical distance from users also increases, so the cloud results in high response times because of its longer physical distance from users;

- Health services are available in all fog nodes and the cloud, so there is no subset of fog nodes responsible for running specific health services unavailable in other fog nodes.

4.2 Design decisions

This section describes the key design decisions considered when proposing SmartVSO and highlights why these decisions were made. The approach introduced in this thesis takes the most of fog and cloud by proposing a middleware that hierarchically chooses which computing layer is more appropriate for running health services with vital signs sent by smartwatches. This middleware considers both user and health service priorities to make offloading decisions. It also monitors the percentage of used CPU and predicts the duration of health services based on historical data to increase throughput, by favoring health services that execute faster. Finally, our approach does not increase complexity in smartwatches, which can run lightweight software and do not need to make complex decisions. Smartwatches simply submit vital signs to a physically close fog node and the offloading decisions to execute the health service on the local fog node or to offload the vital sign to a different machine are made by the proposed middleware. We provide a generic and pluggable mechanism to allow the execution of the most varied health services as serverless functions, without the need to change the source code of the middleware a posteriori. These health services can be implemented and deployed by authorized third parties after undergoing a medical validation process. We emphasize that the implementation of health services is out of the scope of this thesis, and some specific health services were only implemented for evaluation. In summary, we propose a computational model of an architecture with the following main characteristics:

- We employ user and service priorities to calculate a ranking that indicates how important a given vital sign is, while user priority is dynamic (changes over time and depends on the person) and service priority is static (is the same for all users);

- We propose a recursive solution that vertically offloads vital signs to the parent fog node when needed, composing a hierarchical architecture, while the last fog node in the hierarchy is able to offload vital signs to the cloud;

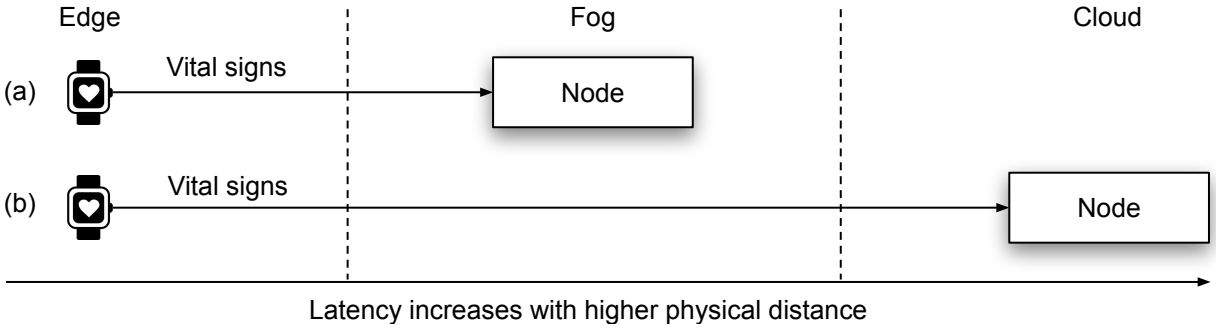
- Algorithms predict how long a health service will take to complete to maximize throughput, by favoring health services that execute faster when multiple vital signs collide with the same calculated ranking;

- Serverless computing is employed as the main technology to execute health services in a sandboxed environment, which provides a pluggable and straightforward approach for authorized third parties to create and deploy their own health services.

It is important to remember that SmartVSO is situated in the context of health and aims to run health services after collecting vital signs with wearable devices, such as smartwatches

and medical equipment. The following paragraphs should be read having this health context in mind. The first design decision is to employ fog computing to achieve responses with a short response time since the fog layer is physically closer to the user when compared to the cloud. Response time is one of our primary concerns when proposing a model in the field of health, especially because SmartVSO may run different time-sensitive, critical health services simultaneously. These services should complete with the minimum possible response time, and one of our goals is to increase throughput, especially for critical vital signs and critical health services. Figure 7 demonstrates the physical difference between fog and cloud by showing smartwatches sending vital signs to both layers, respectively. Figure 7 (a) elucidates that fog is physically close to the smartwatch, resulting in a shorter response time and responses reaching their destinations faster. On the other hand, Figure 7 (b) shows a smartwatch sending data to a computing node on the cloud, which employs a higher response time due to its longer physical distance.

Figure 7: (a) Fog resulting in a shorter response time; (b) Cloud resulting in a higher response time.



Source: prepared by the author.

Scalability is also an important aspect of the proposed model because it allows vital signs to continue being ingested even during high usage peaks, when resources on the fog are overloaded and unable to execute health services. Although fog computing results in shorter response times, it has limited computing capabilities and cannot process huge workloads. SmartVSO model leverages cloud computing to benefit from virtually infinite computing resources when the fog cannot process vital signs. In summary, the architecture first tries to ingest vital signs on the fog node that is physically close to the user to benefit from responses with short response times, but offloads vital signs to another fog node or to the cloud when the current fog node is overloaded. Also, we emphasize that offloading heuristics are proposed with performance in mind: we need to make fast decisions regarding where to process the vital sign, but this decision should not be computational-heavy and time-consuming, in a way that it would be faster to always offload the vital sign or to simply process the vital sign locally. Several variables could influence the person's health, as well as the duration of running a health service, but we do not consider all possible variables in favor of making fast decisions.

Finally, we designed SmartVSO to be transparent, effortless, and automatic, in a way that a minimum number of parameters needs to be configured. Examples include predicting how long the health services will take to complete instead of manually specifying this information, which is used in the offloading decision, and calculating the vital sign ranking based on user and service priorities, also for the offloading decision. Most complexities are also hidden from the public Application Programming Interface (API) so that smartwatches do not need to worry or deal with the internal complexities of running health services in a scalable manner. In short, smartwatches simply submit vital signs to the closest fog node, which takes care of the rest and consumes the vital sign in the most appropriate computing layer (fog or cloud). Smartwatches do not even know what happened internally and how the health services were executed; they simply know that vital signs have been ingested and a health service will automatically process them in the near future, sending a notification to the user or performing an intelligent action according to the person's health status, such as calling an ambulance.

4.3 Architecture

This section dives deep into how the proposed architecture works and explains the core ideas in detail. Important terminology is also described in this section, such as the tree structure's definition and how the connection between fog nodes and the cloud takes place. The proposed decision-making modules and their respective roles in the decision-making process are also introduced, which are responsible for ingesting vital signs and processing them in a local serverless platform or offloading them to the parent layer of the architecture.

4.3.1 Terminology

It is worth mentioning a set of nomenclatures to better understand how this model is designed. These definitions are specific to SmartVSO model and may not reflect what different authors use in their works, as these concepts are specifically contextualized on our model. Table 3 summarizes the basic nomenclatures, while the following paragraphs explain each in depth.

Health service: from a technical perspective of software engineers, it is a serverless function that identifies if the person has a specific health problem and performs an action when the health problem is detected. Each function receives a single vital sign as its input and needs to be deployed to a serverless platform running on each fog node. Also, the health service (serverless function) must be deployed to a cloud serverless platform, which will process vital signs offloaded by the last fog node in the hierarchy. From the perspective of people in the smart city, it is a service that sends notifications to their smartphones in case they have a specific health problem or perform particular actions in critical conditions, such as calling an ambulance.

Fog node: the fog node represents a machine located in a neighborhood of a smart city. This machine consumes vital signs generated by devices such as smartwatches and invokes

Table 3: Main concepts and definitions used on this document, which are relevant to comprehend how the proposed hierarchical fog-cloud architecture works.

Nomenclature	Description
Health service	Serverless function that receives a single vital sign as input and identifies if the person has a specific health problem.
Fog node	Machine located in the neighborhood of a smart city, receiving vital signs and executing health services (serverless functions) with vital signs as the input.
Leaf node	Fog node situated on the lowest level of the hierarchy. This node receives vital signs directly from smartwatches and never from other fog nodes.
Parent node	Fog node to which the vital sign is sent during offloading operations, which is the cloud when it comes to the parent of the fog node on the tree's root.

Source: prepared by the author.

health services (serverless functions) with vital signs as the input. These health services should have been previously deployed to the local serverless platform on the fog node. Although other works in the literature may consider a fog node as a logical entity composed of a cluster of machines, the terminology for the current thesis considers the fog node as a single machine.

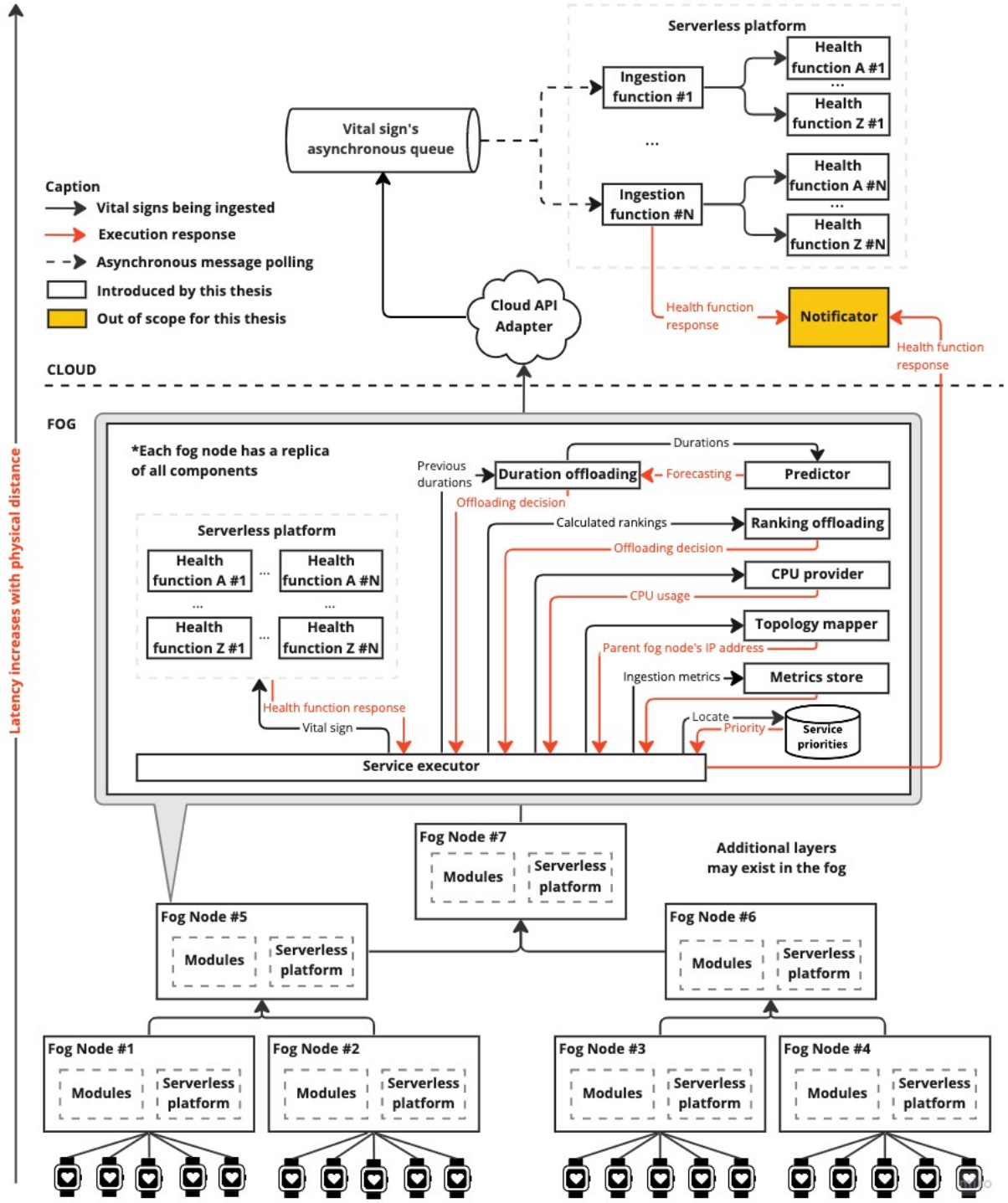
Leaf node: a leaf node is a fog node located on the lowest level of the tree and is physically closer to the end-user. All vital signs collected by smartwatches should be sent to these fog nodes to benefit from shorter response times due to the short physical distance from the user. When CPU usage on the leaf fog node is overloaded, vital signs are sent from this node to its parent fog node of the hierarchy, according to a set of rules that will be detailed in this chapter. Leaf nodes should never receive vital signs from other fog nodes and only from smartwatches.

Parent node: this thesis proposes a model of a hierarchical architecture in the form of a tree to ingest vital signs. This means that fog nodes are distributed among neighborhoods of a smart city in the shape of a tree, where each fog node is connected to a single fog node, which is considered its parent. When the fog node is getting overloaded and cannot execute health services locally, or the vital sign is not critical, incoming vital signs are offloaded to the parent node, which is the only one the current fog node is connected to. The last fog node in the hierarchy (the root of the tree) does not have a parent fog node but is directly connected to the cloud instead.

4.3.2 Data Processing and Data Communication Strategies

This model is designed to work with hierarchical fog nodes in the format of a tree and is not limited to a single computing layer. Figure 8 presents this behavior by illustrating the interaction between different nodes, as well as the modules that exist inside each fog node and how the connection with the cloud takes place. Each fog node runs the modules proposed in this thesis and also runs a serverless platform where health services are executed as serverless functions.

Figure 8: Interaction between hierarchical fog nodes and the cloud, while each node runs a replica of the proposed modules. Each node is directly connected to its parent and the last node on the hierarchy is connected to the cloud, providing virtually infinite computing resources.



Source: prepared by the author.

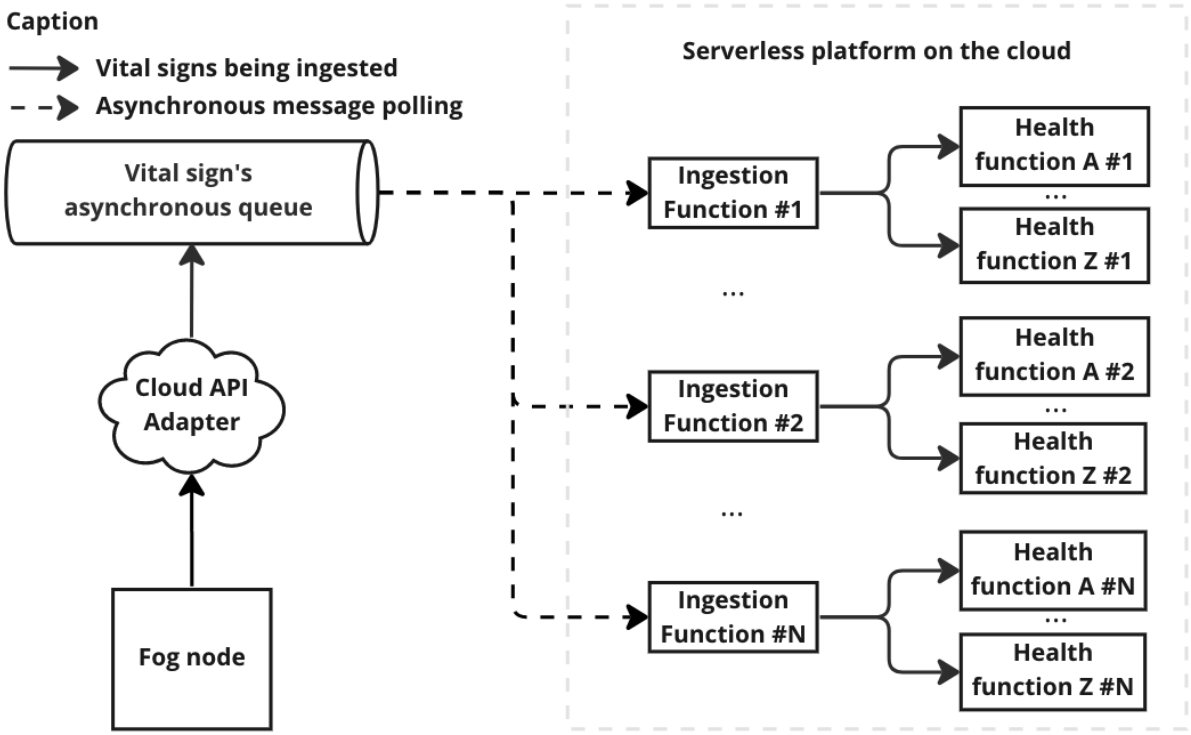
Computing resources on each fog node may differ, so nodes at higher levels of the tree may have more computing resources than nodes on the lower levels, for example. In practice, each node is connected to its parent, which is located on the next level of the hierarchy. The only

exception comes to the fog node located at the root level of the tree, which is not connected to any fog node but is directly connected to the cloud instead. We emphasize that response time increases when sending vital signs to upper layers of the architecture because of the long physical distance. Also, this figure indicates that fog nodes are not connected to more than a single fog node. Therefore, there is no communication between siblings or with multiple parents. The architecture is designed so that smartwatches always send vital signs to a fog node at the first level of the hierarchy because they are physically close to the user, resulting in short response times. Intermediate fog nodes should not receive vital signs directly from smartwatches but only receive vital signs offloaded from nodes located on lower levels of the tree. This figure also indicates the existence of asynchronous processing in the cloud, as vital signs are stored in a queue and are processed at a speed that consumers can deal with, as a large number of vital signs can arise at a given moment, and the computing resources in the cloud might not be ready to process everything simultaneously. This figure also elucidates that the result of the health service, e.g. indicating if the person has a fever, does not navigate back to lower layers of the architecture in a way they arrive in the smartwatches again. Instead, as soon as a health service processes the vital sign, regardless of whether it happening in the fog or in the cloud, the result is sent to a module that we name *notificator*. This is colored in yellow because its implementation and decisions are out of the scope of this thesis, so we assume that future work will focus on it. However, we consider this module because it is directly related to the architecture, with the goal of forwarding the response to the most appropriate destination to improve the user experience. Examples include forwarding results to push notifications on the user's smartphone, notifying a close relative, and notifying the hospital to call an ambulance.

Regarding the execution of health services, Figure 8 also indicates that the architecture processes vital signs synchronously when they are in the fog nodes but asynchronously when they arrive in the cloud, as cloud providers typically impose a limit on how many serverless functions can be invoked at the same time. Let us suppose that the architecture processed vital signs in a synchronous manner in the cloud by invoking a replica of the serverless function whenever a vital sign arises. In that case, the limit of concurrent functions could be easily exceeded, and the cloud vendor would reject additional serverless functions invocations. The practical impact is that vital signs would be discarded if error conditions were not handled appropriately. Implementing logic in the fog nodes to handle such scenarios is possible, although it incurs extra complexity in the fog nodes. They would need to retry the offloading operations or store vital signs in a local queue in the fog node to reprocess later. With this context in mind, to avoid extra complexity in the fog nodes and to ensure that vital signs will not be lost due to limits imposed by the serverless platform of the cloud provider, the *Cloud API Adapter* always stores the received vital signs in a queue and never invokes serverless functions directly. Functions further consume this queue at speed they can deal with, according to the limit of replicas allowed by the cloud provider. In other words, this strategy works as a producer-consumer model, where the *Cloud API Adapter* is the producer of vital signs for the queue, and functions in the cloud

are the consumers.

Figure 9: An overview of the asynchronous processing in the cloud. Multiple replicas of the serverless function named *ingestion function* consume messages as a producer-consumer model, and invoke the respective health function associated with the message.



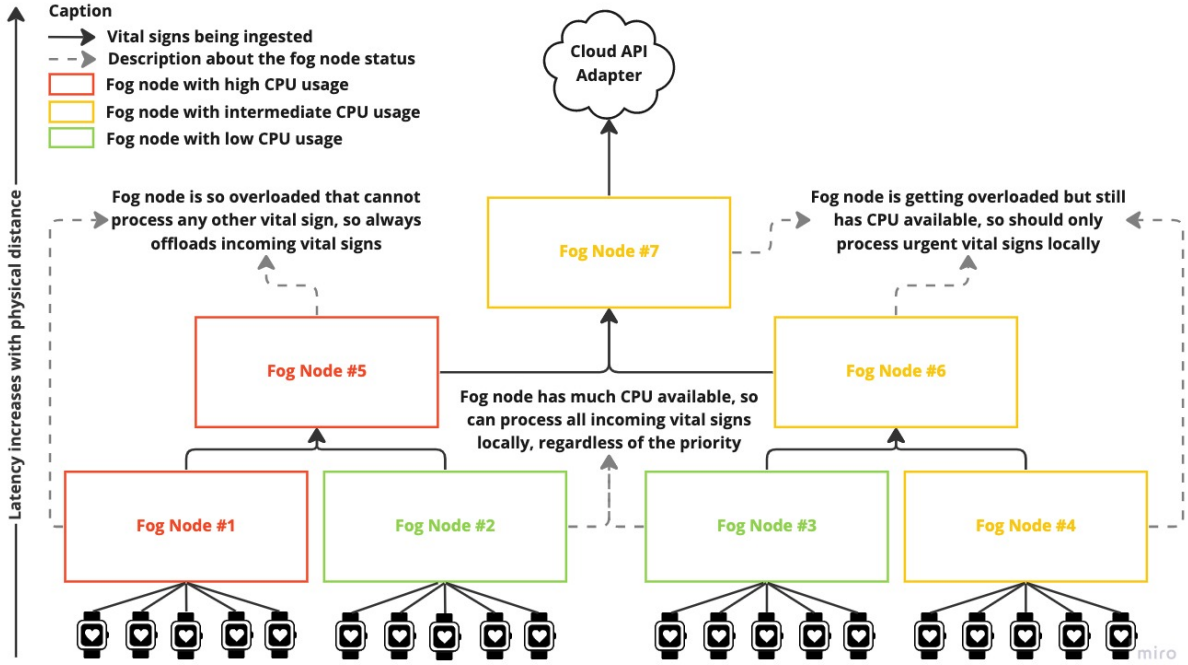
Source: prepared by the author.

Figure 9 complements the details presented in the previous paragraphs and elucidates how SmartVSO consumes vital signs from the queue in the cloud. The work proposed by Chen, Wang, and Liu (2022) indicates that resource-limited devices are unable to handle high usage peaks, so storing additional requests in a queue is a good strategy to deal with this kind of scenario. Their work served as inspiration for us to use a queue in the cloud to deal with high usage peaks, as computing resources in the fog are constrained and may not be able to handle vital signs in a neighborhood with a large concentration of people. This process happens asynchronously, while only a single serverless function named *ingestion function* is directly connected to the queue. However, there are multiple replicas of this function to consume messages faster. For each message, this function extracts the name of the health service that should be executed with the vital sign as its input, and invokes the appropriate health service (which is also a serverless function) that matches the given name. In summary, only a single type of function is connected to the queue, while health services are not directly connected to the queue but are invoked by the main ingestion function instead. The invocation of the health service from the *ingestion function* happens synchronously: the ingestion function remains blocked until the health service completes its execution. The limit of how many serverless function replicas can

be executed simultaneously can be modified by requesting an increase in quotas to the cloud provider support. However, increasing this limit may still not be sufficient for real-world usage in smart cities. The number of vital signs sent to fog nodes in a given neighborhood (and indirectly offloaded to the cloud) may fluctuate because of several reasons, such as the number of people getting sick, the number of people coming from rural zones to urban areas, daily work routines that result in people moving between neighborhoods, social events, and the periodicity in which smartwatches send vital signs. Increasing the limit for serverless functions' concurrency in the cloud may be a simple and tempting solution. However, in case of high usage spikes, vital signs may still surpass the maximum limit and be discarded when rejected by the serverless cloud platform. In other words, increasing the number of concurrent replicas mitigates this problem but does not solve its root: vital signs may still be lost when rejected by a spike in requests. This is why we employ a strategy to always store vital signs in the queue when arriving at the cloud, which allows health services (serverless functions) to consume vital signs at a speed they can handle. In summary, the queue can grow much faster than the vital signs are consumed - which can lead to waiting for a longer time to be processed - but the most important part is that vital signs will not be discarded. Sooner or later, they will be processed by an available serverless function replica.

In this architecture, the idea is that each fog node at the first level of the tree is placed in a different neighborhood of the smart city. Therefore, if the city has x neighborhoods, there would be at least x fog nodes available to process vital signs by receiving data directly from smartwatches. These fog nodes would be represented at the tree's lowest level and receive vital signs directly from smartwatches. The other fog nodes, located at the second level of the tree, would be placed in strategic locations of the city that have a high concentration of people, so when the fog nodes of the first layer become overloaded, they offload vital signs to a fog node on the second layer. The number of levels of the tree depends on the size of the smart city. As a general idea, the larger the smart city, the larger the number of fog nodes. It is possible that large neighborhoods also have more than a single fog node. Typically, the higher the level of the tree, the higher should be the computational power of the fog nodes because they will receive vital signs offloaded from a wide range of lower fog nodes simultaneously. The smartwatch first sends vital signs to a leaf fog node, representing a node physically close to the user. The vital sign can only be consumed on this fog node when the machine is not overloaded (has CPU available). Otherwise, the vital sign must be offloaded to the parent fog node. The vital sign is recursively offloaded to parent fog nodes until any of the following conditions is satisfied: *i)* available CPU on the current machine is enough to execute the health service, which means the machine is not overloaded, or *ii)* the vital sign was offloaded to the cloud by the last fog node on the hierarchy, as the cloud can always ingest vital signs because it provides virtually infinite computing resources. We highlight that the offloading process happens automatically and transparently, so smartwatches do not need to know how this process works internally and do not even need to know that it happened.

Figure 10: Example with fog nodes having different CPU usages, which causes incoming vital signs to be processed locally or to be offloaded if they are not critical. Finally, some fog nodes are so overloaded that cannot even process critical vital signs, and for this reason, they are also offloaded to the parent fog node.



Source: prepared by the author.

There are two main reasons to perform offloading operations on this architecture. Details regarding the offloading strategy will be deeply discussed in Section 4.4, but in summary, it can happen because the incoming vital sign is not critical (was collected from a healthy person that does not have strong requirements for short response times) or because the fog node where the vital sign is currently being ingested is overloaded. Figure 10 presents fog nodes with different colors: green, yellow, and red. Fog nodes in green have much CPU available, meaning they receive few vital signs because there are few people concentrated in the neighborhood where the fog node is located. Since the fog node has much CPU available, it can process all incoming vital signs on the local serverless platform (regardless of how critical the vital sign is) to generate results within a short time interval. When smartwatches send vital signs to fog nodes #2 and #3, for example, they will be processed locally for people in critical conditions and also for healthy people, as they have available computing resources. Moving forward to fog nodes in yellow, they have a considerable percentage of CPU being used, which means the fog node is getting overloaded but is not entirely overloaded yet. In other words, vital signs that arise on fog nodes #4, #6, and #7 will only be processed locally if they are critical, or will be offloaded to the parent layer of the architecture otherwise. Specifically talking about fog node #7, non-critical vital signs will be offloaded to the cloud because this fog node is located in the tree's root. Finally, fog nodes in red (#1 and #5) are much overloaded because there is a concentration of

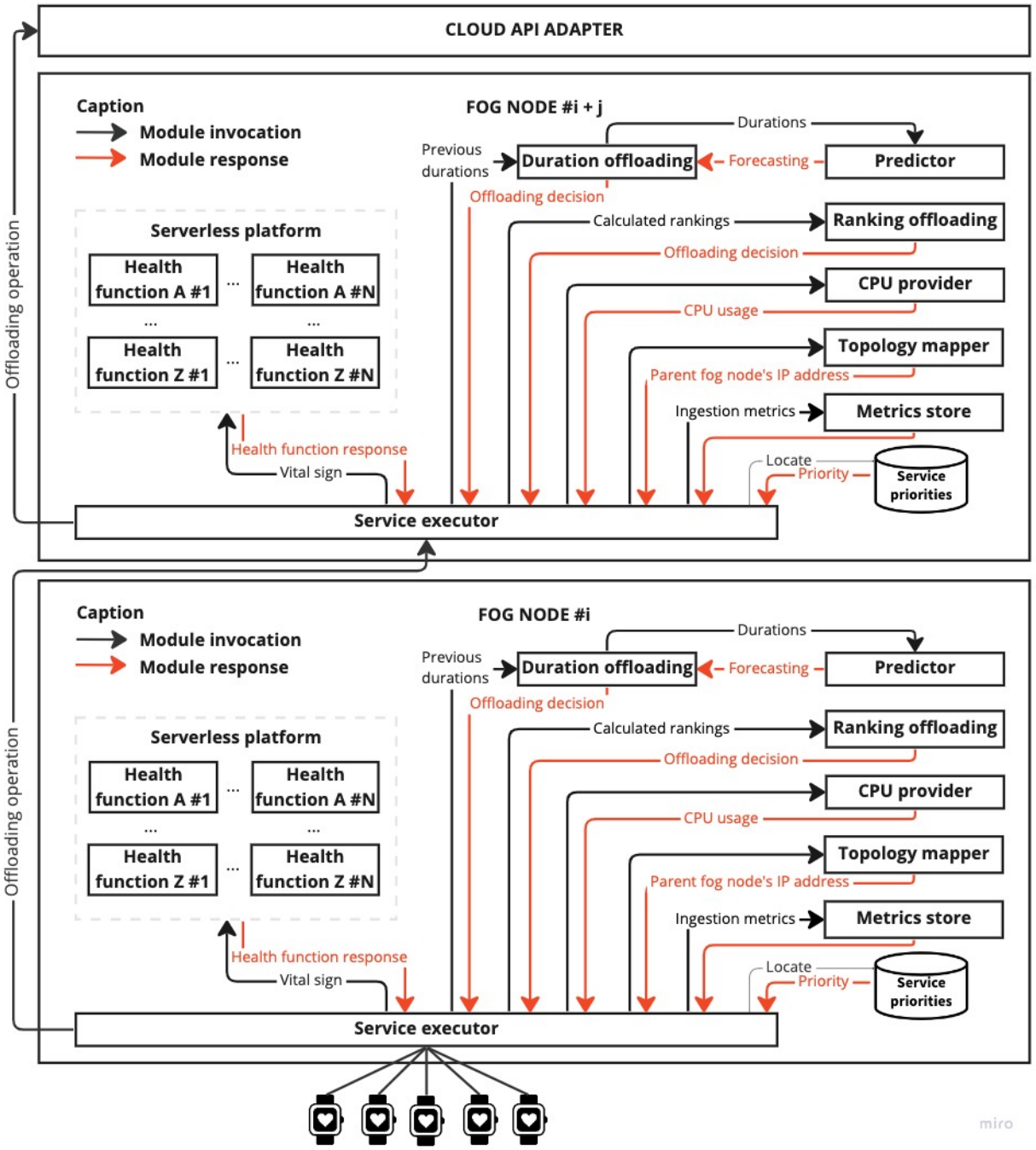
people in the neighborhood where these nodes are located. These fog nodes cannot process any vital signs that arise, regardless of the priority. Keeping in mind that vital signs collected from smartwatches are always sent to fog nodes on the lower level of the hierarchy, this paragraph gives concrete examples to make it easier to understand how vital signs navigate through fog nodes by explaining the possibilities indicated in Table 4, which is introduced in the context of Figure 10. Fog nodes #1, #2, #3, and #4 are in the lower level of the hierarchy and are the only ones to receive vital signs directly from smartwatches. In this example, when a smartwatch sends a vital sign to fog node #1, it will always be processed in fog node #7 or in the cloud because all of the other nodes in the path (#1 and #5) are entirely overloaded and colored in red. On the other hand, when smartwatches send vital signs to fog nodes #2 or #3, they can always be processed locally because they have much CPU available (colored in green). Regarding smartwatches sending vital signs to fog node #4, they will only be processed locally in a given fog node in the path when the vital sign has high priority compared to other vital signs being processed on the same fog node, otherwise, it will be recursively offloaded until a fog node can process it or is offloaded to the cloud. We emphasize that the definition of how urgent a given vital sign is depends on the priority of other vital signs being currently processed on the fog node, which will be further discussed in Subsection 4.4.2.

Table 4: Possibilities where the vital sign can navigate in the context of Figure 10. Vital signs are always sent from smartwatches to a fog node on the first layer of the hierarchy, while the last element on the path represents the location where the vital sign could be processed.

Navigation paths for vital signs
Smartwatch $\longrightarrow$ Node #1 $\longrightarrow$ Node #5 $\longrightarrow$ **Node #7**
Smartwatch $\longrightarrow$ Node #1 $\longrightarrow$ Node #5 $\longrightarrow$ Node #7 $\longrightarrow$ **Cloud**
Smartwatch $\longrightarrow$ **Node #2**
Smartwatch $\longrightarrow$ **Node #3**
Smartwatch $\longrightarrow$ **Node #4**
Smartwatch $\longrightarrow$ Node #4 $\longrightarrow$ **Node #6**
Smartwatch $\longrightarrow$ Node #4 $\longrightarrow$ Node #6 $\longrightarrow$ **Node #7**
Smartwatch $\longrightarrow$ Node #4 $\longrightarrow$ Node #6 $\longrightarrow$ Node #7 $\longrightarrow$ **Cloud**

Finally, Figure 11 provides a global view of SmartVSO and elucidates that smartwatches send vital signs directly to the fog node located on the lowest part of the hierarchy, while Subsection 4.3.3 will further complement this figure by giving details about each module that exists inside a fog node. To configure the interaction between fog nodes, each fog node should reference the IP address of its parent. In contrast, the replica located in the last (root) node of the hierarchy finally references the IP address of the cloud provider. The system administrator manually defines the connection between nodes. Therefore, he or she needs to manually identify the most interesting and beneficial connections according to the physical locations of the fog nodes, which is subject to automation in future work. We reiterate that each fog node connects with, at most, a single parent and does not support connecting with more than a single fog node.

Figure 11: Modules running on fog nodes and their interactions. Each fog node is connected to its parent, and the last is connected to the cloud provider. The *service executor* module is the entry-point for vital signs ingestion and decides between processing locally or offloading.



Source: prepared by the author.

This also makes the model scalable because adding more computing resources when needed is easy. In addition, each fog node is independent of the other to some extent. Adding a fog node to the hierarchy does not employ tight coupling with existing ones, except for changing the destination IP address on specific fog nodes at the lower level of the hierarchy, as they will now be connected to the new node. This is similar to inserting an element into a data structure

of a tree, which also requires exchanging pointers for the connected elements. Future work can automate this manual process to automatically update IP addresses when a fog node is added.

4.3.3 Proposed modules

This subsection will give details about each module that exists and can be found on the fog nodes, except the *Cloud API adapter*, which is presented but does not run on fog nodes. Their input, output, and expected behavior are also presented, and the interactions between modules will be introduced. We emphasize that incoming examples are presented with the JSON format for exchanging messages between components, but any other format could be used, such as XML. Also, the format of the messages depends on how the modules are implemented. Let us suppose they are implemented inside a single process of the operating system. In that case, the invocation can be done by calling methods or functions implemented in an arbitrary programming language. Still, in case each module is implemented as a different service on the operating system, defining a format for exchanging data is important.

4.3.3.1 Service executor

The *service executor* module is the entry point for external communication with the architecture. It is responsible for receiving and processing vital signs with previously deployed health services, or offloading vital signs to the upper layer when needed. The upper layer is the parent fog node, in case of a leaf fog or intermediate fog node, or the cloud, in case the fog node is located on the tree's root. Different factors contribute to the decision of where to process the vital sign and will be further d in Section 4.4. Still, the main factors include the ranking calculated for the vital sign and the CPU percentage available in the fog node. When there is a ranking clash, the *service executor* also considers how long the health service takes to execute to favor health services that execute quickly. The input for the *service executor* module is a message composed of *i)* the vital sign to be ingested, *ii)* the user priority, *iii)* a unique identifier for the message, and *iv)* an optional field indicating the name of the health service that will process the vital sign. Only a single health service will be executed if its name is specified in the body of the message. Otherwise, all existing health services will be automatically triggered and executed with the same vital sign as their input. A sample input message containing the name of an arbitrary health service is presented as follows. This message indicates a vital sign with a heartbeat value of *97*, SpO2 level (oxygen level) of *99*, and temperature of *40*, which were collected from the human body by the smartwatch. It also indicates the *body-temperature-monitor* as the only health service to be triggered when this vital sign arises in an available fog node or in the cloud. The user priority of this message is *3*, which represents a vital sign coming from a person that is not completely healthy but is not in such a critical condition, as the values for user priority range between 1 and 5. Finally, it also contains a unique identifier for the message, which is

required for traceability purposes and to calculate the waiting time by comparing the moment it was registered and the moment it was processed. We emphasize that messages are ingested and processed asynchronously when arriving in the cloud, so having a unique identifier to calculate the waiting time is essential.

```
{
    "service_name": "body-temperature-monitor",
    "vital_sign": "{\"heartbeat\": 97, \"spo2\": 99, \"temperature\": 40}"
    "user_priority": 3,
    "id": "3df4f9e3-f9e8-4cc1-9b18-7a8893a14838"
}
```

The following example shows another message received by an arbitrary fog node, but this time the name of the health service is omitted. This means that all health services previously deployed to the serverless platform will be executed with this same vital sign. For example, let us suppose that there are three different health services deployed to the serverless platform as functions: *body-temperature-monitor*, *heart-failure-predictor*, and *breath-rate-monitor*. These three health services will be executed by receiving the same vital sign as their input. Regarding data received by the health services, fields such as the user priority, the service name, and the unique identifier will not be included in the input message. The main goal of health services is to process vital signs, while the other fields are only useful for internal decisions of the architecture. For this reason, only the *vital sign* structure will be sent as the input when invoking the health service as a serverless function.

```
{
    "vital_sign": "{\"heartbeat\": 97, \"spo2\": 99, \"temperature\": 40}"
    "user_priority": 3,
    "id": "3df4f9e3-f9e8-4cc1-9b18-7a8893a14838"
}
```

Giving more information about the *id* field on the request message, this field is responsible for uniquely identifying a message. At the beginning of the execution, a *pending* state regarding this message should be stored to indicate that this vital sign is being ingested but has not been processed by any health service yet. After the vital sign is processed with an arbitrary health service, this identifier indicates that this specific vital sign has already been consumed. This is important to allow real-time monitoring of how many vital signs are being ingested and how many have already been processed, as well as allows calculating how long it took to process the vital sign. It is expected that the message already arises on the fog node with this *id* field and the vital sign has already been registered in the *pending* state. However, future work can improve this by not indicating the message *id* on the request payload, but enriching the message instead, by automatically including a random identifier when it arises on the leaf fog node.

An important topic to address is how the smartwatch would send the aforementioned fields to the *service executor* in addition to the vital sign. In practice, we assume that many smartwatches simply send the vital signs and are agnostic to any unique identifier or user priority.

The fact that the service name is optional is good because it is one less piece of information that smartwatches need to send. On the other hand, smartwatches are produced by different vendors, and it is not very easy for them to send the user priority. Some solutions for this problem are explained as follows: *i)* vital signs could be sent to an intermediate middleware sitting between the service executor and the smartwatch, responsible for enriching the vital sign before sending the message to the *service executor* module, or *ii)* the *service executor* module itself enriching the message with the user priority. Regardless of the approach, the user priority is mandatory information for the offloading decisions. Finally, the ranking calculation is also the responsibility of the *service executor* module. The user priority is received in the input along with the vital sign, but the service priority needs to be fetched from the database. As will be further explained in Section 4.4.1, the user priority is dynamic because it depends on the person, while the service priority is not related to any specific person and is manually specified by specialists after the health service has been deployed to the serverless platform as a serverless function. Another interesting behavior of the *service executor* module is to send a notification to the *notificator* module indicating that the vital sign with the given *id* has been processed, which moves the state of the vital sign from *pending* to *completed*.

4.3.3.2 Ranking offloading

This module is responsible for running the ranking heuristic presented in Subsection 4.4.2. This module receives two pieces of information: *i)* the rankings calculated for the vital signs that are already being processed by health services at the moment on the current fog node, and *ii)* the ranking calculated for the vital sign that has just been received. The *ranking offoading* module is always invoked by *service executor*. Other modules in SmartVSO never trigger it, as well as it does not invoke any of the other modules. Formally, this module receives a list with the rankings calculated for the vital signs currently processed in the fog node and the ranking calculated for the vital sign being ingested. The following message is an example where 7 vital signs are being processed simultaneously in the current fog node, since the *all_rankings* field is composed of 7 numerical values, representing the calculated ranking for each vital sign. Still, the ranking calculated for the current vital sign is *15*.

```
{"all_rankings": [1, 3, 6, 3, 7, 8, 2], "calculated_ranking": 15}
```

The output of this module is a message indicating if the vital sign should be offloaded to the upper layer of the architecture. This module does not perform any offloading operation, e.g., effectively sending the data to the parent fog node, as this is the responsibility for *service-executor*. Also, as explained in Subsection 4.4.2, there is a possibility of the heuristic being unable to determine if the vital sign should be offloaded or not, so a third value may be returned indicating so. The following message is an example of a response for this module telling the *service-executor* to execute the health service locally with the vital sign as its input instead of performing an offloading operation to the upper layer. Possible values for the

offloading_decision field include *RUN_LOCALLY*, *OFFLOAD*, or *UNKNOWN*.

```
{"offloading_decision": "RUN_LOCALLY"}
```

4.3.3.3 Duration offloading

This module is responsible for detecting whether the vital sign should be offloaded or not based on the forecasting of how long the health service will take to complete, with the logic explained for the duration heuristic presented in Subsection 4.4.3. The forecasting is calculated with the durations of previous executions, for each health service running in the fog node. This module is only invoked by the *service executor* and is never called by the other modules. Also, the *predictor* module is invoked *n* times from the *duration offloading* module to calculate the predicted duration of each health service using statistic techniques, while *n* is the number of health services running at the current moment. The output of this service tells whether the vital sign should be offloaded or not. If this module cannot make a decision, the response will be a third value reporting this situation. In summary, the contract of this module is similar to *ranking offloading* in the sense that information about vital signs being currently processed is received in the module's input, as well as information about the vital sign that has just been received by *service executor*. The following message indicates an example of the input for this module. The *durations_running_services* field indicates the last few durations for the health services that are currently being executed at the moment on the current fog node, and the *durations_target_service* field has the previous few durations for the health service that is supposed to execute the vital sign received. In case multiple replicas of the same health service are running simultaneously in the fog node, duplicate values should be removed, as well as values for the target health service should only be included in *durations_target_service* and not in the *durations_running_services*.

```
{
    "durations_running_services": [[528, 745, 821], [1120, 1187, 1238]],
    "durations_target_service": [123, 175, 158],
}
```

The following message indicates a response with the offloading decision made by this module, similar to the response generated by *ranking offloading*. Finally, the response can also contain a third value representing that the heuristic could not determine whether the offloading operation should be performed. The latter situation seldom happens for this module because it is difficult for two health services to have the same forecasted duration. On the other hand, for the *ranking offloading* function, it is not so uncommon because there is a limited range of priorities for users and services, and it is easier to have a ranking clash when processing a huge number of vital signs concurrently.

```
{"offloading_decision": "RUN_LOCALLY"}
```

4.3.3.4 Predictor

This module is responsible for forecasting a set of numerical data received on the input and returning the predicted value as the output. This module is generic enough to be invoked with arbitrary values, regardless of what they semantically represent. However, in practice, this module is only invoked by the *duration offloading* module with data regarding previous durations of a given health service. This module is stateless because it receives all the necessary information on the input and does not locate anything from any database. We suggest using the *double exponential smoothing* to make predictions based on the received values because this statistical technique gives exponentially higher weight to the recent values and also considers trends, differently from the *simple exponential smoothing*, which does not consider trends. Also, we do not employ *triple exponential smoothing* because we have not yet seen the need to consider seasonality. However, this module is generic enough to be implemented with different forecasting techniques in the future. Multiple implementations of this module could be used automatically depending on the scenario to achieve better results. The input for this module is as follows:

```
{
  "data": [17, 21, 23, 26, 26, 28, 30, 30, 30, 31, 32, 33, 39, 41, 41],
  "smoothing_level": 0.8,
  "smoothing_trend": 0.2,
  "future_data_points": 1
}
```

The *data* parameter represents the numerical values to be considered in the forecasting. In other words, based on this set of values, the prediction will be made. It is expected that values are sorted by chronological order, in the sense that the last values of the list are more recent and have more weight when compared to the first values of the list, which are older and have less weight. This behavior is the essence of the *double exponential smoothing*. Both the *smoothing_level* and the *smoothing_trend* fields are required for the double exponential smoothing and calibrate how much the data will impact the forecasting. Finally, the *future_data_points* field indicates how many future values should be forecasted. For example, if this field is equal to 1, then a single forecast will be made, and the result will be a list with a single value. If this value is x, being x greater than 1, then x future forecasting values will be returned in the list. Finally, although this function is generic, it is always invoked by the *duration offloading* module with *future_data_points* equal to 1 because we are only interested in a single forecasted value. The following example indicates the response for the input with a single future data point:

```
{"forecast": [43.08]}
```

The above response means that, based on the double exponential smoothing, the next value according to this pattern and the inclination parameters will be 43.08. As another example, let us suppose that the *future_data_points* is 2, then the response would be the forecasting for the next 2 values.

```
{"forecast": [43.08, 44.8]}
```

4.3.3.5 CPU provider

Each fog node also runs a module named *CPU provider* that gives the percentage of CPU currently used in the local machine. As mentioned at the beginning of this section, the percentage of used CPU is important because it allows the *service executor* to know how overloaded (or underloaded) the fog node is when a vital sign arises. Depending on how overloaded the fog node is, and also depending on additional heuristics, the *service executor* may decide to offload the incoming vital sign to the upper layer of the hierarchy, as explained in Section 4.4. Each *CPU provider* instance keeps running a loop in the background that periodically collects the percentage of used CPU on the local machine, in a way that it can be promptly provided when the *service executor* asks for this information. This module does not expect any parameter when invoked because it should simply give the current CPU percentage collected by the loop running in the background. An example of a response message returned by this module is as follows. Future work may extend this component to collect additional metrics, such as memory, disk, and network, but the initial version proposed in this thesis makes decisions considering only CPU observations. We emphasize that the contract of this module is to return immediately whenever invoked by the *service executor*, to reduce the possibility of compromising the performance of the decision-making process. In other words, when the *service executor* asks for this information, the *cpu-provider* should immediately return with the available CPU information. This way, the process of collecting CPU usage percentage (which can range from milliseconds to seconds, depending on the employed strategy) must be done asynchronously in the background.

```
{"cpu": 14.4296875}
```

4.3.3.6 Topology mapping

This module is responsible for resolving the mapping between the current fog node and the upper layer, which can be the parent fog node or the cloud. The main challenge is how the current fog node knows the destination to which it should send vital signs. Different approaches exist to solve this, such as *i)* using a fixed IP address on each fog node, *ii)* relying on a DNS to translate the name of the fog node to its IP address, or *iii)* having a mapping of the whole topology containing the connection between fog nodes. Each approach has pros and cons. The pros of the first and second approaches include simplicity, especially when using a DNS server because one does not need to update the connections whenever the IP for a given node changes. On the other hand, the cons include being challenging to maintain the destination for offloading operations in case it is hardcoded on the source code or located in the properties file

located on the fog node itself. When directly referencing the IP address, the module would need to be redeployed whenever the destination for offloading operations changes, which is time-consuming. With this problem in mind, we introduce the *topology mapping* module to return the IP address of the upper layer based on a global mapping with connections between fog nodes and the cloud. This mapping file is stored in the cloud so that fog nodes can fetch the new mapping whenever it is updated. We emphasize that this *topology mapping* module is triggered by the *service executor* only once, during its initialization phase, and stores the destination in the memory to avoid the overhead of constantly triggering this module to resolve the mapping from the file located in the cloud. The following message indicates an example of a response of this module, containing the destination of the upper layer (which can be another fog node or the cloud) to which the current fog node is connected to. This module does not consider any input during invocation because it is expected to be deployed with the name of fog node, which is used to locate the connection for the current fog node in the mappings file. The following message indicates an example of a response for this module.

```
{"mapped_destination":"18.124.112.159"}
```

The following listing indicates an example of a mapping file with three fog nodes. Both *fog_node_a* and *fog_node_b* are connected to *fog_node_c*, which in turn is connected to the *cloud_api_adapter*. Let us suppose that we are interested in discovering the IP address of the parent fog node of *fog_node_a*. The module first looks for the name of its connect fog node, which in this case is *fog_node_c*. Given this information, the module looks for the IP address of this fog node, which in this case is 18.124.112.159. This IP address is the result of the module because it indicates the IP address of the parent fog node, to which vital signs will be sent during offloading operations.

```
# Connection between each fog node
connection.fog_node_a=fog_node_c
connection.fog_node_b=fog_node_c
connection.fog_node_c=cloud_api_adapter

# IP Address defined for each fog node or Cloud API Adapter
ip_address.fog_node_a=19.212.193.202
ip_address.fog_node_b=18.204.199.123
ip_address.fog_node_c=18.124.112.159
ip_address.cloud_api_adapter=https://adapter-lambda-url.eu-west-2.on.aws
```

4.3.3.7 Metrics store

This module is responsible for storing and providing the metrics during the execution of health services and the ingestion of vital signs. These metrics include information such as whether the vital sign was offloaded or not, which offloading heuristics were triggered, the amount of used CPU when ingesting the vital sign, the timestamp of CPU collection, the name

of the health service, the calculated ranking, among several pieces of information that can be used for further analysis. In this thesis, we use this module to generate charts after running experiments. Still, in future work (or in production), this module can monitor in real-time how fog nodes are behaving and how many vital signs are being received during a given period. This helps visualize which city neighborhoods are receiving more data, for example. Also, this module is only invoked by the *service executor* to store metrics about the ingestion but does not return any information on the response body. The following message indicates an example of what pieces of information are sent when invoking this module.

```
{
    "user_priority": 3,
    "ranking": 8,
    "used_cpu": 96,
    "last_cpu_observation": 98.23,
    "cpu_collection_timestamp": 1663527788128,
    "function": "foo-function",
    "offloading": true,
    "running_locally": false,
    "exceeded_critical_cpu_threshold": true,
    "triggered_heuristic_by_ranking": false,
    "result_for_heuristic_by_ranking": false,
    "triggered_heuristic_by_duration": false,
    "result_for_heuristic_by_duration": false,
    "assuming_fallback_for_heuristics": false
}
```

4.3.3.8 Cloud API adapter

Incoming vital signs can be offloaded to the upper layer of the hierarchy according to certain rules, while the upper layer can be another fog node or can be the cloud if the current fog node is located on the tree's root. The *Cloud API Adapter* will receive vital signs offloaded from the last fog node on the hierarchy and will store the message in a queue, to be processed asynchronously when serverless function replicas are available in the serverless platform located in the cloud. Cloud providers restrict the number of parallel execution of serverless functions, so for this reason, the messages need to be stored in a queue to be processed when serverless functions are available. This module implements on purpose the same contract (API) as the *service executor* module that runs on every fog node. This makes the architecture generic and transparent: once the fog node has the IP address of its parent, it does not need to know if this parent is another fog node or the cloud itself. They share the same API, so the last fog node in the hierarchy can offload the vital sign to its parent (the *Cloud API Adapter*) and automatically benefit from virtually infinite computing resources without even knowing that it was offloaded to the cloud. Although this module is analogous to the *service executor* in the sense that both implement the

same API contract, there is a subtle difference in the implementation because this module is quite simpler and does not employ any offloading heuristic since the cloud provides virtually infinite computing resources. This module simply receives vital signs and stores them in a queue. With this in mind, the field *user_priority* presented in the Subsection 4.3.3.1 is also received by the *Cloud API adapter* but is ignored because there is no heuristic using it in the cloud. Future work may exercise this ability to generalize the API contract and create additional modules implementing the same contract, such as a module that offloads vital signs to multiple cloud providers according to a set of rules. This could prevent problems caused by a specific cloud provider becoming a single point of failure and compromising the execution of health services.

4.3.4 Health services

Health services are implemented as serverless functions and can be plugged into the architecture anytime without needing to recompile or redeploy the proposed modules. Once the health service is deployed to the serverless platform, it can promptly process incoming vital signs. The suggestion is that only authorized third parties can deploy health services in the serverless platform, as they are critical and deal with sensitive information about the users. Health services only receive the vital sign as the input and do not receive additional information such as the user priority and the message identifier. The following example indicates the message passed as the input when the *service executor* invokes a health service as a serverless function. We emphasize that different smartwatches may collect different information about the human body, so the health service should not expect that a given information is always available.

```
{"heartbeat": 97, "spo2": 99, "temperature": 40}
```

There are some requirements that software engineers should consider when developing health services (serverless functions) to be executed with this computational model. First, as the execution of the serverless function is feasible to exceed timeout constraints on the serverless platform, services must perform atomic operations to avoid inconsistent data in case the serverless platform abruptly terminates them. Supposing that a service performs a set of interactions with an external entity and the execution is suddenly finished in the middle of these interactions, the service should be able to clean up resources on further executions or only perform transactional, atomic, and idempotent interactions with external entities. As a concrete example, a service that calls an ambulance when the patient is close to having a heart attack should not call the ambulance twice if the service exceeds the timeout on the fog and is re-executed. A second requirement is that services do not persist information on the local machine because further executions of the health service might be executed on different fog nodes or even on the cloud. Let us suppose the service needs to persist some information across different executions. In that case, it is suggested to keep them in a remote database in the cloud that is accessible from any location, regardless of where the vital sign is processed. Also, future work may address strate-

gies to share data between fog nodes transparently so that health services can store information locally, and they will be automatically replicated to all of the fog nodes in an asynchronous manner, as well as will also be stored in the cloud.

4.4 Offloading strategy

This thesis introduces an offloading strategy that favors the processing of critical vital signs collected from unhealthy people or people in urgent situations. Before diving deeper into the strategy, we explain what an offloading operation represents and indicate its benefits for the proposed model. In the context of this thesis, an offloading operation means forwarding a vital sign from the current fog node to its parent fog node or to the cloud, in a way that the health service running in a remote machine will further process this vital sign. We emphasize that smartwatches send vital signs to their physically closer fog node to achieve low-latency communications due to their physical proximity, as presented in Subsection 4.1. Also, computing resources on the current fog node are limited, so it may not be possible to process all vital signs locally and we need to decide which subset of vital signs should be offloaded. Finally, as offloading operations incur extra network latency, it should be avoided as much as possible for critical vital signs because people in critical conditions should be monitored in real-time. With this in mind, our offloading strategy increases the quality of experience by processing critical vital signs locally most of the time and forwarding non-critical vital signs to remote machines. We summarize the following main benefits of our offloading strategy:

- Critical vital signs will achieve short response times most of the time because non-critical vital signs will be offloaded to a remote machine, resulting in more computing resources available on the current fog node, although non-critical vital signs will be harmed by the extra network latency;
- Scalability is achieved by allowing more machines to extend the computing capabilities of the architecture, which helps to deal with large workloads when the local machine cannot process such a large number of incoming vital signs.

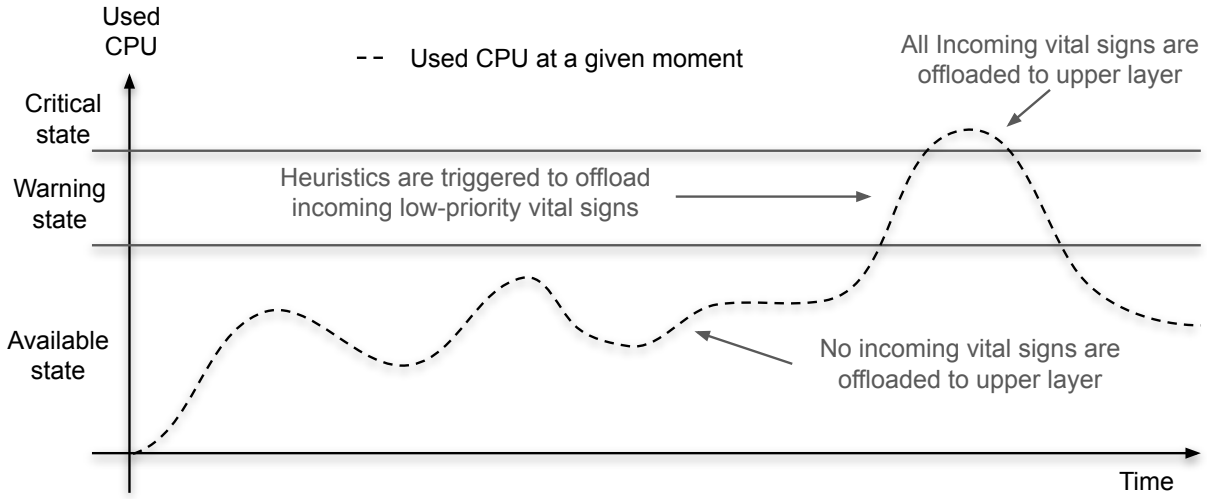
To better comprehend the need for priorities, the following paragraphs and examples will be presented in the context of a smart city, where most people have agreed to have their vital signs daily monitored by computers to detect health problems. This scenario includes people of different ages located in different neighborhoods of the city, such as people inside hospitals who are facing health problems and need to be closely monitored, or people without health problems but still willing to have their vital signs monitored, so problems might be detected in the future. The smart city needs a scalable computing infrastructure to support the processing of a huge number of vital signs, which varies according to the number of users in the neighborhood and can fluctuate throughout the day. This infrastructure has several interconnected fog nodes distributed across the neighborhoods, so wearable devices and medical equipment can send vital

signs to the physically closest fog node. The idea is to achieve a short response time by reducing the physical distance between the user and the computing infrastructure whenever possible. In addition, computing resources on each node are finite. When the person's closest fog node receives the vital sign, but its resources are overloaded, the vital sign needs to be offloaded to the parent fog node of the hierarchy, so the processing can be done on a remote, different physical location. The challenge is how to efficiently use the computing infrastructure in the smart city and, especially, how to use it when computing resources are getting overloaded while still taking response time into account to meet needs in the field of health. Users with high priority should receive notifications with the lowest possible response time by sending vital signs to the person's nearest computing infrastructure, at the same time it is acceptable that users with lower priority wait for additional seconds or minutes until responses are received, to the detriment of vital signs with higher priority.

In practice, the extra latency is not a big problem for non-critical vital signs because the person is healthy and has fewer chances of having a health problem. Delaying the health service response for healthy people would typically not cause trouble. On the other hand, the extra latency would be a real problem for critical vital signs because each second matters for a person with serious health problems or in emergency situations. This is the reason why we favor critical vital signs and try to process them locally whenever possible. However, there are situations where the current fog node is so overloaded that it cannot process any other incoming vital sign, regardless of being critical or not, which leads us to offload any incoming vital sign when the machine is completely running out of resources. Diving deeper into how the offloading strategy works, each fog node can be situated in one of the three following states according to the percentage of CPU currently used: *available*, *warning*, or *critical*. The rules for deciding if the incoming vital sign should be executed locally or offloaded to the parent fog node are directly related to the percentage of used CPU. Our decision was inspired by other works in the literature, such as the one introduced by Righi et al. (2016) with a model called AutoElastic. Their work monitors the CPU load and considers a minimum and a maximum threshold to perform elasticity operations and initiate or destroy virtual machines, in a way that VMs are instantiated when the CPU load is greater than the maximum threshold, and VMs are destroyed when the CPU usage is below the minimum CPU threshold. We are inspired by this idea but we use it to offload vital signs instead of allocating or deallocating cloud resources.

Figure 12 elucidates how the percentage of used CPU affects the decision of where the vital sign should be processed. This figure indicates that all vital signs can be processed on the current fog node when most CPU is available, meaning the fog node is in the Available state, and the CPU is below the warning threshold. In addition, this figure shows that all vital signs are offloaded to the parent fog node when the used CPU is extremely high and exceeds the critical threshold, regardless of the vital sign priority because the fog node is so overloaded that execution of health services would be compromised. Finally, only high-priority vital signs are processed locally when the CPU usage in the current fog node is between warning and critical

Figure 12: Chart presenting the threshold for offloading low-priority vital signs.



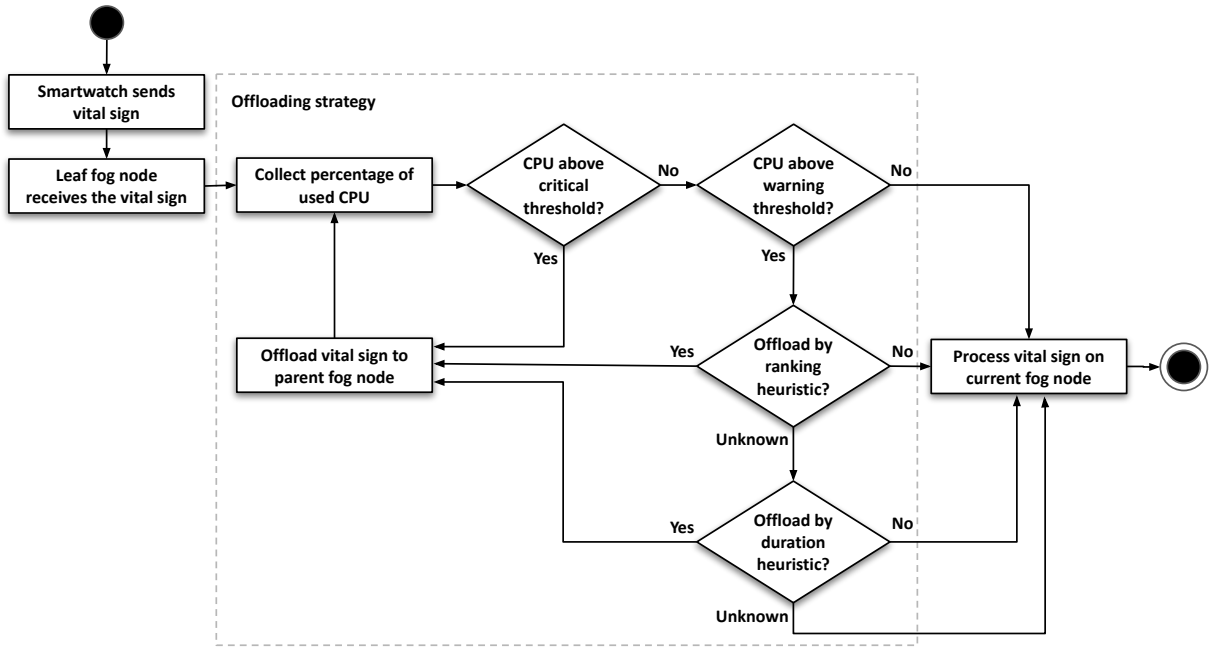
Source: adapted from Righi et al. (2016).

thresholds, while low-priority vital signs are offloaded to the parent fog node and processed remotely. We highlight that offloading heuristics are only triggered when the fog node is in the *warning* state. It is worth mentioning that threshold values for this thesis are static. The fog node is in the *critical* state when the CPU usage is greater than x, and is in the *warning* state when the CPU usage is greater than y, being $0 > y > x$. In its turn, when the CPU usage is equal to or lower than y, the fog node is in the *available* state. The main goal of the *warning* state is to process high-priority vital signs locally and avoid offloading them to parent fog nodes as much as possible. Processing high-priority vital signs locally results in lower response time, which is desirable, as these vital signs should be processed as soon as possible. The following items describe the possible states of the fog nodes:

- **Critical:** no vital sign should be processed locally when the percentage of used CPU is above the critical threshold, but should be offloaded to the parent fog node instead, as health services could be compromised when running on an overloaded fog node;
- **Warning:** only high-priority vital signs should be processed locally when the percentage of used CPU is above a warning threshold but still less than the critical threshold, while low-priority vital signs should be offloaded to the parent fog node;
- **Available:** all incoming vital signs can be processed locally when a large percentage of CPU is available (less than the warning threshold), regardless of the vital sign's priority.

Figure 13 presents a flow diagram with the main steps involved in this decision-making process, highlighting that vital signs can be offloaded to the parent fog node recursively until a fog node with enough processing capabilities is found. This diagram is focused on the fog layer for simplicity reasons, but the vital sign can always be ingested when the upper layer is the

Figure 13: Flow diagram elucidating how the service execution happens on the fog layer. This diagram elucidates that the fog node processes the vital sign locally when it has enough computing capabilities or the vital sign has high priority, or offloads to the parent fog node otherwise.



Source: prepared by the author.

cloud, which employs a higher response time but has virtually infinite processing capabilities. In other words, if the vital sign cannot be processed on the local fog node, it is offloaded to a *service executor* instance of the parent fog node or to the *Cloud API adapter* if the parent layer is the cloud.

Regarding reasons why there is a fluctuation in the percentage of used CPU, the fog node to which the smartwatch sends vital signs may differ as people move through neighborhoods, with the goal of always sending vital signs to a physically close fog node to achieve low-latency communications. A fog node of a given neighborhood may also receive more or less vital signs according to the number of people physically located around it. Also, we assume that smartwatches used by a healthy and young population can send vital signs with a low periodicity because they are less likely to have critical health problems. In turn, smartwatches used by older people, or younger people with health problems, could send data with a high periodicity. The more people with diseases or more older people, the higher the number of vital signs the fog node in this neighborhood would receive. External conditions may also impact the decision of how long the periodicity to send vital signs should be, which may cause fog nodes to be more or less overloaded. Examples include seasons of the year and pandemic or endemic situations. For example, some diseases are more likely to happen in the winter, even in young people. Smartwatches could send vital signs with a higher periodicity in scenarios like this because this allows people to be closely monitored by health services. In endemic or pandemic situations, where there is a higher risk of contamination and proliferation of diseases, it is also important

that vital signs are sent with a higher periodicity. Finally, another example that can impact how overloaded fog nodes become is the number of people coming from rural zones to urban areas. Although out of the scope of this thesis, smart modules can be implemented to calibrate how often the smartwatch should send data to the fog node to reduce the chances of the node becoming overloaded. The calibration could make smartwatches used by young people send fewer data at the expense of helping the nearest fog node have resources available for high-priority people. After some time, when resources on the closest fog node become available again, devices could be automatically recalibrated to send data with normal periodicity, which is suggested as future work.

4.4.1 User and service priority

As previously explained in the offloading strategy, offloading decisions are based on the percentage of used CPU and also based on how critical the vital sign is. This thesis considers two types of priorities that we combine into a single numerical value called *ranking*, which is an indicator of how critical the vital sign is. The SmartVSO model considers user and service priorities to calculate the ranking, as presented in Table 5 and detailed in the following paragraphs. In summary, the user priority comes from the request payload because it is specific to the user whose vital sign is being ingested, while the service priority comes from the database because it is specific to the health service that will be executed and not related to the user.

Table 5: Categories of priorities considered by the SmartVSO model.

Priority	**Origin**	**Description**
User	Request	Indicates how critical the vital sign collected from this user is, being directly related to the person's health condition, such as being sick or healthy.
Service	Database	Indicates how important a given health service is, regardless of the person being monitored. Specific services are arguably more critical than others.

Source: prepared by the author.

User priority: people in critical condition in a smart hospital's emergency room should receive closer attention than people who do not have comorbidities or who have already recovered from health problems. Each second matters when monitoring people in critical situations so that doctors can perform agile decisions, which are decisive in saving a person's life (PAREEK; TIWARI; BHATNAGAR, 2021). Health services should continuously monitor the person's status and proactively send notifications about problems that may arise or worsen in the next few moments. At the same time, other people may also need their health status monitored at home because they are not feeling well but are not in such a critical condition. We emphasize that, although everybody needs attention, the latter may not be in such a critical situation as people in the emergency room. Therefore, it is acceptable that health services result in a higher delay to

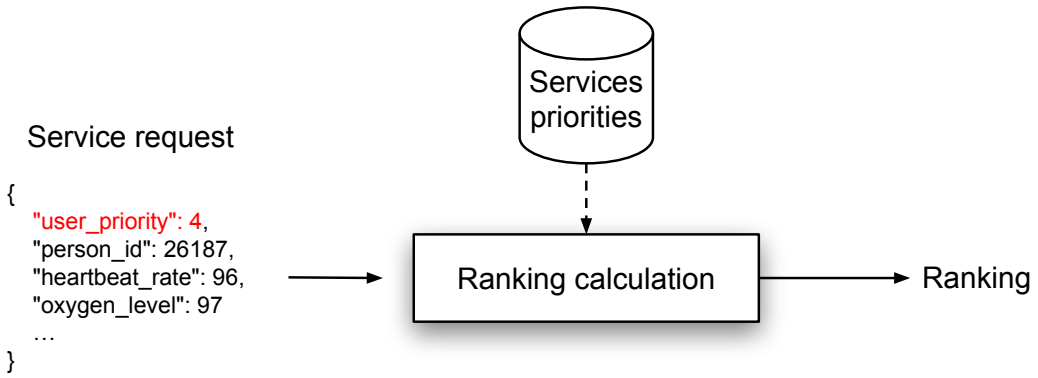
process vital signs of people in less critical situations. Finally, there may be completely healthy groups of people and it is acceptable that services take even more extra time to process their vital signs, especially during high usage peaks, when computing resources are getting overloaded. This elucidates that there are different acceptable limits on the response time according to the user being monitored, which directly impacts the quality of experience.

Service priority: medical analysis may indicate that specific health services are more relevant and valuable than others. A wide range of health services can be implemented in the context of a smart city so that different problems are detected, and medical assistance can be quickly provided. The SmartVSO model should do its best to give results for critical services when compared to less critical ones, especially when the infrastructure's finite computing resources are overloaded because of high usage peaks. For example, a health service may employ statistics to foresee if the person will have heart failure in the next few minutes and automatically call an ambulance in this situation, while another service may monitor the body temperature to send notifications when the person has a fever. Medical analysis may indicate that the former is more important than the latter. Therefore, when the current fog node runs out of resources, SmartVSO should give higher importance to executions regarding the former service.

This concludes that SmartVSO needs to consider two types of priorities when deciding which vital signs should be processed on the current fog node to avoid extra communication latency caused by the offloading operation: *i)* user priorities, and *ii)* service priorities. Technically speaking, priorities impact the decision of the layer (current fog node or parent fog node of the hierarchy) where vital signs should be processed. Our goal is to favor the execution of critical health services for users in critical conditions whenever possible, which results in shorter response times, by processing vital signs locally. This is where priorities make a difference: when the architecture is experiencing high usage peaks, it should favor high-priority vital signs and give them greater chances of running on the small and precious amount of resources that are still available on the current fog node, to benefit from short response time. Less important vital signs should be offloaded to parent fog nodes on the architecture and will experience higher response times. Therefore, priorities are important to optimize the processing of critical vital signs when finite resources on the current fog node are getting overloaded. Without priorities, though, the model could incorrectly offload critical vital signs, which would be a problem for people in urgent conditions.

Figure 14 complements the previous explanation and presents how the model combines user priority with service priority. The former is important when specific users should receive more attention than others, such as older adults or groups in risky situations in the field of health, that should have their vital signs processed with the lowest possible response time. The latter is related to the priority of the service itself, as specific health services may be more critical than others. We emphasize that user priority is dynamic and is received on the request payload, as a person with health problems may become healthier after some time, and further vital signs should indicate lower priorities. The service priority is typically static and does not change

Figure 14: User priority originated from the payload and service priority from the database.



Source: prepared by the author.

over time unless the service is re-deployed or the priority is globally reconfigured, and for this reason, it is fetched from the database. The ranking calculation combines both priorities in a single numeric value, while the combination of each priority is detailed in Figure 15. The reason why we considered five user priorities and five service priorities instead of a different number of priorities is that leading emergency triage systems also consider this number, such as the Canadian Triage and Acuity Scale (CTAS), the Emergency Severity Index (ESI), the Australasian Triage Scale (ATS), and the Manchester Triage System (MTS) (SIMON JUNIOR et al., 2022). This indicates that five levels of priorities are already used globally and inspired us to introduce the calculation of the ranking based on this number of priorities as well.

Figure 15: Ranking combining service and user priorities, with intense colors representing more important vital signs. The higher the ranking, the smaller the chance of being offloaded.

Source: prepared by the author.

We give higher weight to user priority than service priority when calculating the ranking, as we argue that the priority associated with a given user should have more impact on the final offloading decision. Let us give an example where a healthy person (user priority 1) is running

a critical service to detect heart failures, which has service priority 5, at the same time a person in critical conditions (user priority 5) is running a non-critical service to detect fever, which has service priority 1. We could simply sum the priorities without giving higher weight to the user priority as a naive approach, but this would cause a ranking collision because both calculated rankings would be 6. We understand that a person in critical condition requires higher attention than a healthy person, regardless of the priority of the health service. Also, we argue that running a health service with priority 1 for a person in critical conditions, such as identifying fever, is more important than identifying a heart attack in a person who is healthy and has a really low probability of having a heart attack. With this in mind, we give twice the weight for the user priority when calculating the ranking. We also understand that this is still arguable, so medical validation is needed before using this approach in production. Equation 4.1 indicates how the ranking is calculated on SmartVSO and makes it clear that user priority has double weight.

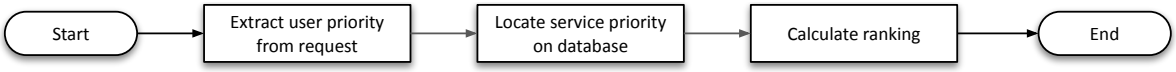
$$ranking = 2 * user\_priority + service\_priority \tag{4.1}$$

There is another benefit in multiplying the user priority by 2 instead of simply summing both priorities or considering the service priority as a tiebreaker when the user priority is the same for multiple vital signs. Let us give another example, but this time regarding a user with priority 5 and a user with priority 4. The former is running a low-priority service to identify fever, which has priority 1, but the latter is running a high-priority service to identify heart failures, which has priority 5. In this scenario, when giving double weight to the user priority, the ranking calculated for the user with priority 4 will be 13 and the ranking for the user with priority 5 will be 11. This is arguably correct to calculate a higher ranking for a user with priority 4 in this situation. Both user priorities (4 and 5) are really close to each other, so both persons are in a condition that requires much attention. Given this context that both persons are unhealthy and that the importance of health services is completely different (service priority 1 against 5), we believe it makes sense to calculate a higher ranking to the user with priority 4 because it is running a critical health service, differently from the user with priority 5 that is running a non-critical service.

On the other hand, there is a scenario where giving double weight to the user priority is more controversial and subject to discussion, as follows. A person with user priority 3 running a critical health service (with priority 5) will have a calculated ranking of 11, but a user with priority 5 running a non-critical health service (with priority 1) will also have a calculated ranking of 11. This will result in a ranking collision and the heuristic will be unable to determine the offloading operation, therefore giving the final decision to the duration heuristic that will be further presented in this section. Finally, Figure 16 presents a flow diagram with the main steps involved in calculating the ranking. The first step is to collect the user priority from the request payload. After the user priority is extracted from the payload, the algorithm must collect the service priority from the database, as it is static. As previously explained, the service priority is

not related to the user and therefore is globally defined for the service and stored in a database. Finally, once the heuristic has both priority values, the ranking is calculated with Equation 4.1.

Figure 16: Flow diagram summarizing the main steps for ranking calculation.



Source: prepared by the author.

This thesis originally proposed a naive calculation of the ranking by simply summing the user priority with the service priority, without giving double weight to the user priority. This means that both priorities had the same impact on the ranking, but we identified undesired results with that approach. As explained in the previous paragraph, a critical service for a very healthy person should not have the same importance as a non-critical service for a person in urgent conditions. This has led to undesired offloading behaviors, as the system could favor the vital sign regarding the healthy user depending on the duration heuristic. Given this context, we decided to improve the ranking calculation to give double weight to the user priority to avoid this aforementioned problem. This leads to more accurate decisions for offloading operations, which helps process vital signs for people in critical conditions with shorter response times.

The calculation of the user priority is out of scope for this thesis, and we expect this value to be received from the request payload, along with the vital sign. Future work may study the pros and cons of bringing the calculation of user priority to the fog nodes instead of receiving it on the request payload. One possible approach is that smartwatches send vital signs to a middleware running in single-board computers (SBC) available in the person's house, responsible for enriching the vital signs by including the user priority in the request. A suggestion is to consider the user's age to initialize the user priority, followed by an instruction that multiplies this priority by a given *alpha* value in case the person has any comorbidity. The advantages of calculating the user priority on the edge of the network include reducing the responsibility on the fog nodes, which can focus as much as possible on service execution and offloading operations, and reducing the processing efforts in the fog node. The disadvantage of calculating the user priority on the edge is, in case the algorithm to calculate the priority needs to be improved, the algorithm must be updated on every edge device. This increases complexity on the edge. Also, during a period there may be edge devices calculating user priority with the previous version of the algorithm and other edge devices calculating with the newer version. This may cause the architecture to treat users differently until all edge devices use the latest version of the algorithm.

Future work can improve this model and also provide an API to associate a priority with a given service. This API would allow a given service to have its priority changed anytime, as long as this decision is aligned with medical specialists who agree with it. In addition, this specific API should not be public to prevent non-authorized clients from changing these values

on purpose, so this should be technically exposed behind authorization flows. This API could receive the following parameters: *i)* name of the service that will have its priority modified, and *ii)* priority level to be associated with the service. Invoking this API for every service would not be mandatory, so services default to priority 3 when this API is not invoked. Finally, a real-world example of a flow to define the service priorities is suggested as follows, but other possibilities may be employed. A user-friendly interface, such as a Web page or a smartphone app, can be developed to provide a global view of existing services. Doctors and specialists in the field of health could log in to this management page with their credentials, so they can quickly review and change the priority for existing services according to the needs of a smart city. For example, supposing that cases of a specific disease are increasing in the city and need to be monitored with greater attention, authorized doctors may increase the priority for this service so that it executes with lower response time during high usage peaks. Future work may also allow specifying priorities for a service in a subset of neighborhoods, such as neighborhoods with hospitals.

4.4.2 Ranking heuristic

Considering the details of how the calculation of the ranking works based on user and service priority, Algorithm 1 is responsible for detecting if the vital sign should be offloaded based on rankings calculated for vital signs. To be more specific, this heuristic considers: *i)* the ranking that has just been calculated for the vital sign being ingested, and *ii)* rankings that were previously calculated for other vital signs being processed in the fog node at the current moment. The semantics of what a low-priority vital sign means depends on the rankings of other vital signs being processed at the moment. For example, let us suppose a situation where there are n vital signs being processed simultaneously, and all of them have a ranking value of 5. If the incoming vital sign has a ranking value of 6, it is considered a high-priority vital sign because this ranking is higher than vital signs already running. On the other hand, if all vital signs have a ranking value of 7 but the incoming vital sign has a ranking of 6, the incoming vital sign is assumed to have low priority. The algorithm's input considers the ranking calculated for the incoming vital sign and the ranking calculated for other vital signs currently being processed. The output is the offloading decision made by this heuristic, which can be *offload*, *run_locally*, and *unknown*. The first is returned when the calculated ranking is smaller than the median of the rankings for services that are already running. In comparison, the second is returned when the calculated ranking is greater than the median. Finally, a ranking collision may occur, and the calculated ranking can be the same as the median of the other rankings. In this situation, the heuristic is unable to detect if the ranking is semantically more or less important than the others, and for this reason, returns the *unknown* value. This will further lead to the execution of the second heuristic, called *duration heuristic*, which will try to decide based on how long the health services take to complete. In summary, the heuristic presented in Algorithm 1 is

quite fast to execute because it does not employ any computational-intensive decision. This is important because the heuristic should not employ a heavy overhead so that it would be faster to execute the health service locally instead of running the heuristic to decide the offloading.

Algorithm 1 Ranking heuristic to determine the decision for the offloading operation.

Require: $calculated\_ranking, all\_rankings$
Ensure: $decision$
1: **function** $ranking\_heuristic(calculated\_ranking, all\_rankings)$
2: $sorted\_rankings \leftarrow sort\_removing\_duplicates(all\_rankings)$
3: $middle\_ranking \leftarrow median(sorted\_rankings)$
4: $decision \leftarrow$ *RUN_LOCALLY*
5: **if** $calculated\_ranking < middle\_ranking$ **then**
6: $decision \leftarrow$ *OFFLOAD*
7: **end if**
8: **if** $calculated\_ranking == middle\_ranking$ **then**
9: $decision \leftarrow$ *UNKNOWN*
10: **end if**
11: **return** $decision$
12: **end function**

4.4.3 Duration heuristic

The duration heuristic aims to determine the offloading operation based on forecasting how long each health service will take to complete. This subsection uses the term *forecasting* as a synonym for *prediction*. Offloading operations employ extra overhead because of the network latency, so it is more efficient to offload vital signs when the health service that will process the vital sign takes a considerably long time to finish when compared to other health services running on the current fog node. This heuristic helps improve the architecture's throughput when the ranking heuristic could not determine the offloading operation. Algorithm 2 describes this heuristic, which receives two parameters: *i)* the list of previous durations for the health service that will process the incoming vital sign, and *ii)* the list of previous durations for the other health services that are being executed on the fog node. By a previous duration, we mean how long (in milliseconds) a previous execution of the health service took to complete. Let us suppose that the health service *foo* will process the incoming vital sign, and this health service has already been executed three times in the current fog node with different vital signs. It took 527 milliseconds to complete on the first execution, while it took 630 milliseconds in the second execution and 820 milliseconds in the last one. The parameter *durations_target_service* will receive a list with values 527, 630, and 820, respectively. The second parameter is a list of lists, meaning it is a list with the previous durations for each of the other health services, except *foo*, since its durations were already received on the first parameter. The algorithm makes one prediction for each health service to calculate how long its execution will take to finish. Each prediction can be understood as the following transformation: given a list of *n* numerical

durations of an arbitrary health service, this list is transformed into a single value that indicates the forecasting of how long the service will take to complete on its next execution. In other words, the parameter *durations_target_service* will be transformed from a list of values to a single value, and the parameter *durations_running_services* will be converted from a list of lists of values to a list of predictions.

Algorithm 2 Duration heuristic to determine the decision for the offloading operation.

Require: $durations\_target\_service, durations\_running\_services$
Ensure: $decision$
1: **function** $duration\_heuristic(durations\_target\_service, durations\_running\_services)$
2: $N \leftarrow length(durations\_running\_services)$
3: $predictions\_running\_services \leftarrow [N]$
4: **for** $i \leftarrow 0$ to $N - 1$ **do**
5: $durations\_current\_service \leftarrow durations\_running\_services_i$
6: $predictions\_running\_services_i \leftarrow HES(durations\_current\_service)$
7: **end for**
8: $sorted\_predictions \leftarrow sort\_removing\_duplicates(predictions\_running\_services)$
9: $middle\_prediction \leftarrow median(sorted\_predictions)$
10: $prediction\_target\_service \leftarrow HES(durations\_target\_service)$
11: $decision \leftarrow$ ***RUN_LOCALLY***
12: **if** $prediction\_target\_service > middle\_prediction$ **then**
13: $decision \leftarrow$ ***OFFLOAD***
14: **end if**
15: **if** $prediction\_target\_service == middle\_prediction$ **then**
16: $decision \leftarrow$ ***UNKNOWN***
17: **end if**
18: **return** $decision$
19: **end function**

Still, on Algorithm 2, the decision-making is made based on the forecasting values instead of the original values regarding the previous durations. Duplicate predictions are removed, and the median value is collected from the list of predictions. The offloading operation happens when the prediction for the target service is greater than the median prediction for the other services. For this algorithm, the name of the health service and any additional information about it are irrelevant - the heuristic is only interested in the numerical durations and nothing else. We emphasize that this historical information only considers previous executions on the current fog node and does not consider the durations in other fog nodes of the architecture. Otherwise, it would incur extra overhead to share the duration of a health service across multiple fog nodes. Since each fog node can have different computing capabilities, they could also execute the same health service faster or slower, which would not be appropriate for the prediction. We also highlight a significant difference from the ranking heuristic: the ranking heuristic decides to offload the vital sign when the calculated ranking is lower than the median, but the duration heuristic is the opposite, as it offloads when the predicted value is greater than the median. Finally, this algorithm leverages Holt Exponential Smoothing (HES), also known as Double

Exponential Smoothing, to forecast trends to foresee the duration of future executions.

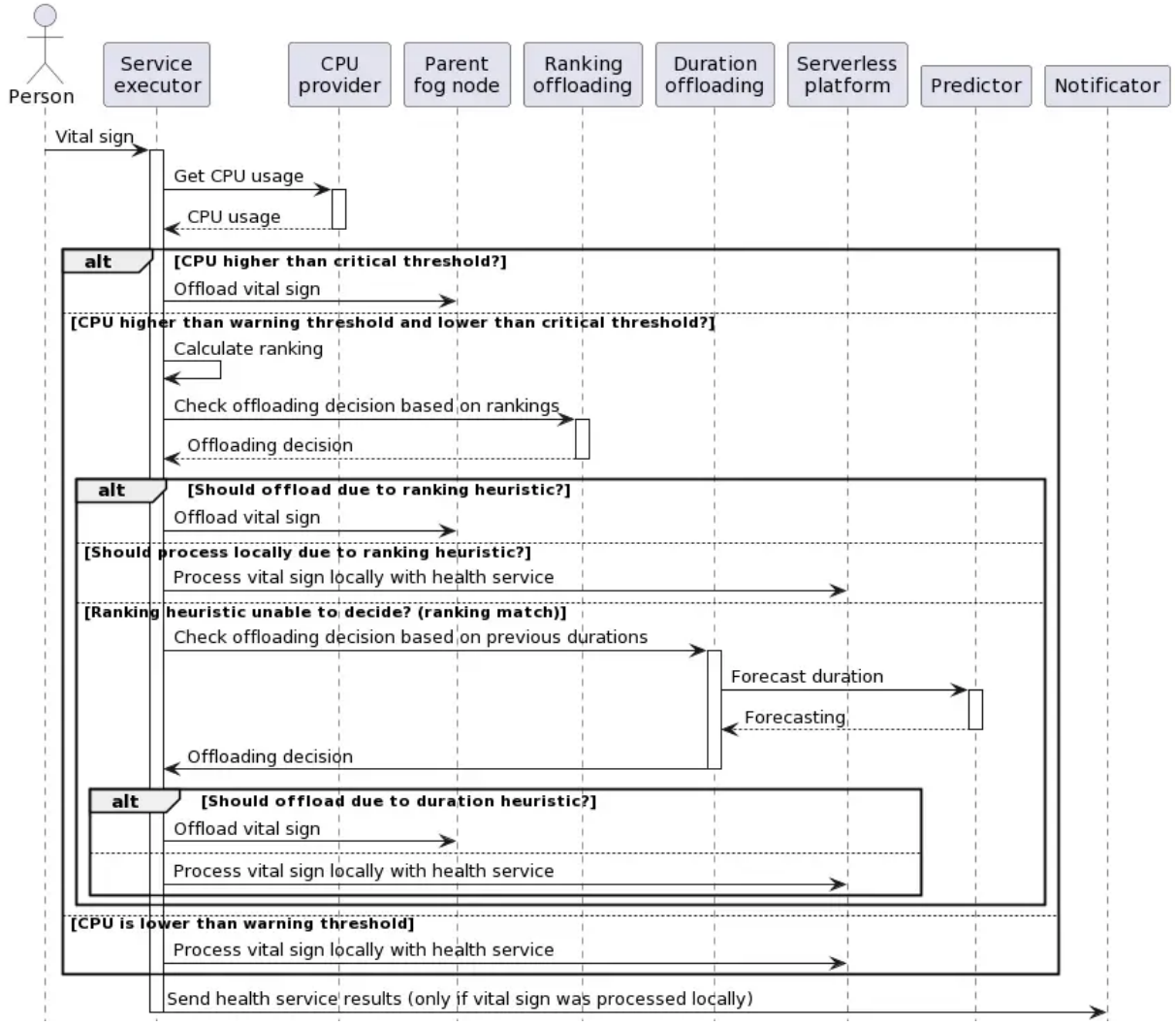
We emphasize that this duration heuristic only considers the duration of the health service executing on the current fog node. In other words, it does not consider any other information about the health service, such as how much memory and disk it uses. Also, this duration does not consider the network latency of offloading operations because we consider the duration as the difference between the final and initial timestamp when the health service was executed on the local fog node. Also, the duration is not shared among different fog nodes; each fog node has its own set of historical durations to make predictions. This heuristic is only executed as a fallback for the ranking heuristic, which means it is only invoked when the ranking heuristic cannot make an offloading decision because there was a collision in the calculated rankings. Only in this situation the duration heuristic is triggered to make the final decision, and it is never triggered in any other situation.

Finally, future work can address different strategies for offloading vital signs based on the duration of a health service. For example, instead of comparing how long each health service will take to complete, the heuristic could compare if the forecasted duration is higher than the communication latency with the fog node on the upper layer of the hierarchy. On the other hand, doing so also imposes some challenges because the offloading operation can be made recursively across a path of several fog nodes. Only considering the network latency for a single offloading operation may not be enough. Additionally, we highlight that making predictions may incur extra computational overhead, even though it can be little. Future work can compare how the architecture behaves with a simpler heuristic that only makes decisions based on the duration of the last execution of the health service and does not make any prediction at all, instead of making predictions with a set of previous durations. The decision could be less accurate, but the benefit is that it would make decisions faster. Future work can compare the trade-offs between approaches to understanding which fits better in which scenario.

4.4.4 Algorithms combined

Since the previous subsections indicated details about each offloading heuristic, this subsection details how the heuristics work together by providing a complete view of the workflow. Figure 17 complements the previous explanations by presenting a sequence diagram highlighting the main interactions between modules and additional players. Specific details were omitted for the sake of simplicity. First, the smartwatch sends the vital sign to the *service executor* module, which will be responsible for processing the vital sign locally or offloading it to the upper layer according to a set of rules. The upper layer, in turn, can be another fog node or even the cloud itself, as the SmartVSO model is generic enough to abstract implementation details from the parent layer. Once the *service executor* receives the vital sign, it first calls another module named *CPU provider* to collect the CPU usage on the current fog node. The *service executor* now makes decisions based on the percentage of CPU used on the fog node. In summary, all

Figure 17: Sequence diagram depicting main steps involved during the ingestion of a vital sign. The *service executor* is the entry point for ingestion, while offloading decisions are related to the percentage of used CPU and to offloading heuristics based on ranking and service duration.



Source: prepared by the author.

vital signs should be processed locally when most resources are available, regardless of the priority. Otherwise, the *service executor* needs to filter which vital signs should be processed locally and which should not. Low-priority vital signs should be offloaded when resources on the fog node are getting overloaded, while high-priority vital signs should be executed locally as much as possible to benefit from short response times. With this in mind, when CPU usage is between the warning and the critical thresholds, the *ranking offloading* module is triggered to decide the offloading operation based on rankings calculated for vital signs being processed at the moment. If the heuristic indicates that offloading should happen, then the vital sign is forwarded to the parent fog node, otherwise, it is processed on the local serverless platform with a health service written as a serverless function. Also, if the *ranking offloading* module could not make this decision, the *duration offloading* module is triggered to decide the offloading based

on previous durations of health services. This is done with the support of the *predictor* function, which receives a given set of numerical values and returns a prediction based on them. The result of this heuristic is verified as a final decision for the offloading operation. Finally, if the vital sign was processed locally because of the aforementioned reasons, results are forwarded to the *notificator* module to notify the person in case a health problem is detected.

Finally, Algorithm 3 indicates how a given vital sign is ingested when it arises in the fog node. This algorithm receives a message with the vital sign as the input, along with the health service that should be executed, the user priority, and a unique identifier for the message. This algorithm detects if the vital sign should be offloaded by invoking another algorithm that helps with this decision, and offloads the vital sign to the upper layer if needed. Otherwise, the algorithm executes the health service in the local serverless platform with the vital sign as the input. Another behavior of this algorithm is to register that the vital sign is being processed by the health service, as well as indicate that the vital sign is not being processed anymore after the health service finishes its execution. This is important because offloading heuristics need information about the vital signs that are currently being processed and therefore need to further access this data. Finally, if the health service executes locally, this algorithm sends the service response to the *notificator* module, which is responsible for making an intelligent action with the response. This includes sending a push notification to the user's smartphone or calling an ambulance. Finally, this algorithm does not return any information on the response body.

Algorithm 3 Ingestion of a vital sign by considering the offloading strategy.

Require: $message$
1: **function** $ingest\_vital\_sign\_message(message)$
2: $user\_priority \leftarrow message.user\_priority$
3: $health\_service \leftarrow message.health\_service$
4: $vital\_sign \leftarrow message.vital\_sign$
5: $ranking \leftarrow calculate\_ranking(user\_priority, health\_service)$
6: **if** $should\_offload(ranking, health\_service)$ **then**
7: $offload(message)$
8: **else**
9: $id \leftarrow message.id$
10: $register\_execution\_started(id, health\_service, ranking)$
11: $response \leftarrow execute\_health\_service(health\_service, vital\_sign)$
12: $register\_execution\_finished(id)$
13: $send\_response\_to\_notificator\_module(id, response)$
14: **end if**
15: **end function**

Algorithm 4 is invoked by algorithm 3 and is responsible for telling whether the vital sign should be offloaded to the upper layer. The input is the calculated ranking and the name of the health service, while the response is a boolean value indicating the decision of the offloading operation. The first step is to collect the current CPU usage on the fog node, which is compared with the critical and warning thresholds and directly impacts the offloading operation. When

the CPU usage is greater than the critical threshold, the algorithm understands that offloading the vital sign is required and does not even trigger the ranking heuristic or the duration heuristic to favor critical vital signs. Next, in the situation where the CPU usage is not greater than the critical threshold, the algorithm checks if this value is greater than the warning threshold. In this scenario, specific heuristics are triggered to determine the offloading operation, while each heuristic was presented in the previous subsections. The algorithm first invokes the ranking heuristic to determine the offloading based on the calculated ranking. However, when the ranking heuristic cannot make such decision, there is a fallback to the duration heuristic to decide according to how long the health service will take to complete. If both heuristics are unable to decide, or the CPU usage is not greater than the warning threshold, the algorithm assumes that the vital sign should be processed locally.

Algorithm 4 Offloading decision according to CPU usage and heuristics.

Require: $calculated\_ranking, health\_service$
Ensure: $offload$
1: **function** $should\_offload(calculated\_ranking, health\_service)$
2: $offload \leftarrow false$
3: $cpu\_usage \leftarrow get\_used\_cpu\_percentage()$
4: **if** $cpu\_usage >$ *CRITICAL_THRESHOLD* **then**
5: $offload \leftarrow true$
6: **else**
7: **if** $cpu\_usage >$ *WARNING_THRESHOLD* **then**
8: $all\_rankings \leftarrow get\_rankings\_running\_services()$
9: $decision \leftarrow ranking\_heuristic(calculated\_ranking, all\_rankings)$
10: **if** $decision ==$ *UNKNOWN* **then**
11: $durations\_service \leftarrow get\_durations(health\_service)$
12: $durations\_others \leftarrow get\_durations\_ignoring(health\_service)$
13: $decision \leftarrow duration\_heuristic(durations\_service, durations\_others)$
14: **end if**
15: **if** $decision ==$ *OFFLOAD* **then**
16: $offload \leftarrow true$
17: **end if**
18: **end if**
19: **end if**
20: **return** $offload$
21: **end function**

4.5 Partial considerations

This chapter presented SmartVSO, a computational model of a hierarchical fog-cloud architecture for executing health services while taking user and service priorities into account. The main goal of this model is to favor high-priority vital signs, in a way that they can be processed with short response times, by offloading non-critical vital signs to upper fog nodes on

the hierarchy. This model combines fog computing and cloud computing to benefit from short response times and virtually infinite computing capabilities, respectively, so vital signs continue being processed even when the architecture faces a sudden and large workload. Also, strategic offloading decisions are made when computing resources on the current fog node are getting overloaded. This includes favoring critical vital signs and optimizing throughput with health services that complete faster, according to forecasting based on previous service durations.

This chapter first presented design decisions that guided the proposition of this computational model. We then presented an overview of the architecture, along with how the connection between fog nodes and the cloud takes place. Several modules are presented in the meantime, as well as the expected behavior for each module and the interactions they have with each other. The *service executor* is the entry point for ingesting vital signs and is responsible for triggering most other modules. This is followed by a deep explanation of how our offloading strategy works, which considers the CPU load on the current fog node and triggers heuristics based on rankings and health service durations to determine the offloading decision. SmartVSO is designed to work with serverless functions on both fog and cloud layers because of the increasing usage of this technology in the field of Internet of Things, along with the ease of creating and deploying health services with this technology. Serverless functions can be easily implemented and run in an isolated fashion in a way that a function cannot mess with the underlying operating system or access data of different health services running on the same infrastructure. This increases the security of SmartVSO model. Also, serverless computing allows configuring timeout limits to prevent a certain function from running indefinitely and monopolizing computing resources for a long time.

In addition, we present a detailed view of how SmartVSO model works with a hierarchical approach so that several fog nodes are distributed and interconnected in neighborhoods of a smart city in the form of a tree. We also present the three states where the platform can be situated, according to the percentage of CPU used on the node. This includes *available*, *warning*, and *critical* states. Each one impacts the decision to process the vital sign on the local fog node or automatically offload it to the parent fog nodes on the hierarchy. The *available* state indicates that all vital signs can be processed on the local node, regardless of the calculated ranking. The *critical* state indicates that no vital sign can be processed on the local node because there are almost no computing resources available. In turn, the *warning* state is the most interesting one and is the moment when most of our decision-making algorithms are executed to optimize the usage of resources while still considering priorities in the field of health.

Finally, we also present how the vital sign ranking is calculated based on the user priority, which comes from the request payload, and the service priority, which is globally defined for each service regardless of the user sending vital signs. Both priority values are combined into a single value called *ranking*. The higher the ranking, the higher the chances of the vital sign being processed on the current fog node when computing resources are getting overloaded, resulting in shorter response times for people in critical conditions. However, when the ranking

for the incoming vital sign collides with the rankings of vital signs already being processed, the duration heuristic is triggered as the second criterion for the offloading decision, so only short-term health services execute on the current fog node. The prediction is made with statistical methods based on historical execution data. As a last thought, we hope this model serves as an inspiration for different areas of knowledge, including those not directly related to health, but could also benefit from scalable workload distribution in a hierarchical manner.

5 EVALUATION METHODOLOGY

This chapter details how SmartVSO is evaluated with the developed prototype. Metrics, evaluation scenarios, thresholds, and additional characteristics of each experiment are also presented throughout this chapter. We give detailed information about the implementation details, such as the programming languages and technologies employed when building the solution, and present different architectures used in the evaluation. We also explain why we considered different architectures and give detailed information about each scenario that is evaluated, along with the number of vital signs ingested, how vital signs were artificially generated, the CPU thresholds considered for the experiments, and strategies to collect CPU load.

5.1 Implementation details

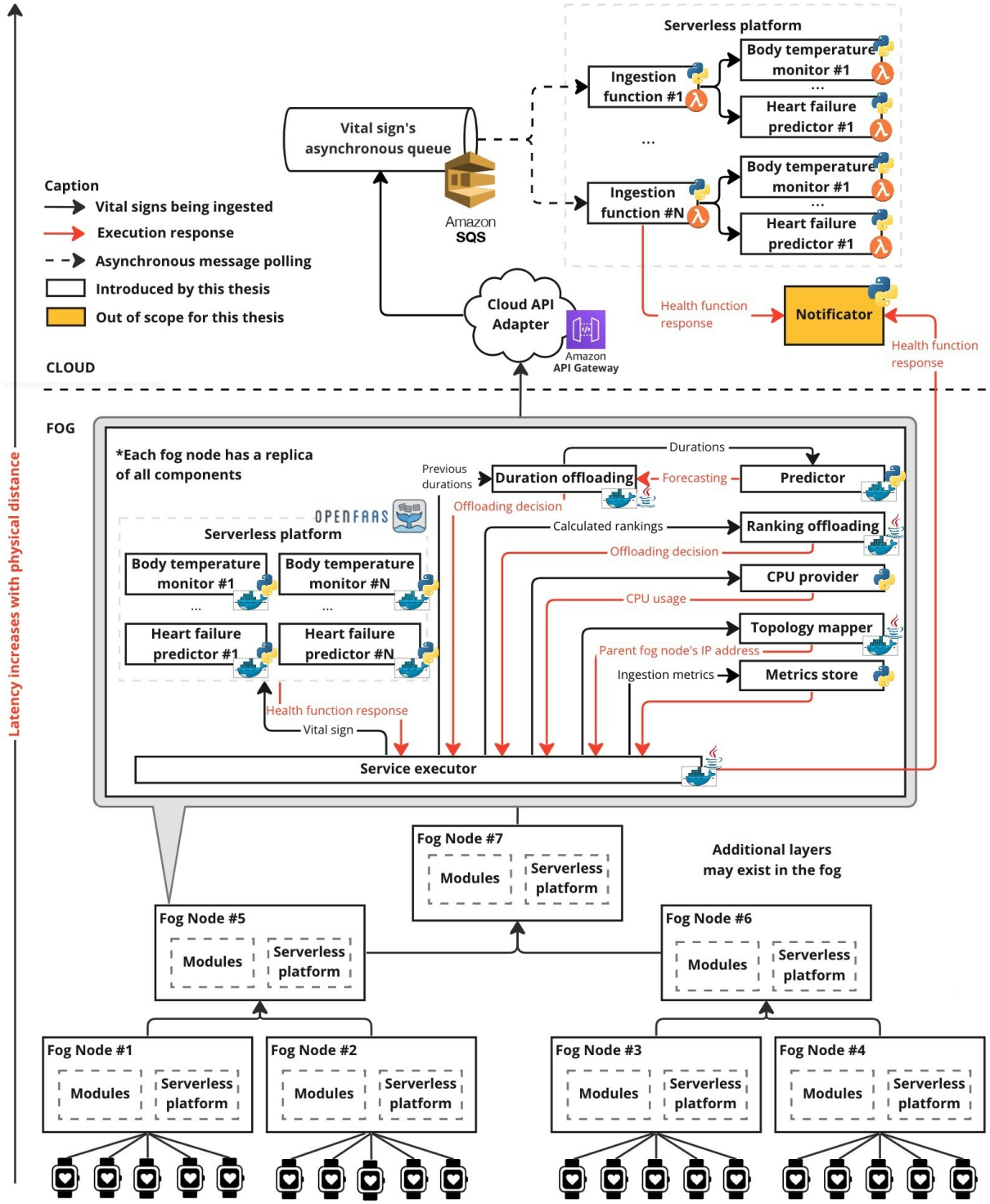
The modules implemented for this thesis are publicly available under vital-signs-ingestion[1] repository on GitHub. This repository encompasses the decision-making modules and the health services, as well as evaluation scripts and JMeter test plans used for the evaluation. Since this thesis aims to use serverless functions because of their trending usage in the field of Internet of Things, almost every module was written as serverless functions, including the decision-making modules. Therefore, not only the health services were written as serverless functions. The only exceptions are the *results*, *metrics*, and *cpu-provider* modules. The first two hold, in the memory, information about the vital signs ingested and for this reason needed to maintain a state during the experiment. The third one, *cpu-provider*, needs to run directly on the operating system without any container or virtualization layer that abstracts the visualization of computing resources, as the CPU should be collected considering the fog node and not considering a sandboxed environment that would provide a limited view of the system. Table 6 presents the modules implemented for this thesis and the language in which they were developed.

Table 6: Modules implemented for running experiments of this thesis.

Name	Language	Function?	Description
service-executor	Java	✓	Ingestion of vital signs
ranking-offloading	Java	✓	Offloading heuristic based on calculated rankings
duration-offloading	Java	✓	Offloading heuristic based on services durations
topology-mapping	Java	✓	Returns the IP address to offload vital signs
ingestion-function	Python	✓	Ingests vital signs from the queue on the cloud
predictor	Python	✓	Makes prediction with double exponential smoothing
results	Python	-	Stores and retrieves results after processing vital signs
metrics	Python	-	Stores and retrieves metrics during the experiment
cpu-provider	Python	-	Returns information about the CPU on the fog node

[1] https://github.com/GustavoASC/vital-signs-ingestion

Figure 18: Interaction between hierarchical fog nodes and the cloud. This figure also elucidates the technologies employed to build each component of the architecture.



Source: prepared by the author.

Figure 18 brings the same architecture previously presented in Subsection 8, but this time with the main technologies used to implement each proposed module. The reason why some functions were implemented in Java 11 is because of the familiarity of the authors with this lan-

guage. These functions were implemented using the Quarkus[2] framework with native GraalVM[3] builds enabled, resulting in faster boot and execution times when compared to running the modules with traditional Java bytecode. On the other hand, Python has broader support for prediction and forecasting libraries, which is why the predictor function was written in this language. Functions must also be wrapped into a Docker container before being deployed to OpenFaaS. Also, Python provides easy built-in functionality to expose HTTP web servers, which is why some modules were also written in this language. Regardless of the language, modules communicate with each other through HTTP requests by exchanging messages in JSON format.

Functions overview: several functions and modules were implemented for this thesis. Some of them were implemented exactly as indicated in Chapter 4. However, some modules have interesting details to emphasize and, for this reason, have separate paragraphs in this chapter. The *service executor* module was written as a Java serverless function and was one of the most challenging modules to implement because it is the entry point of the fog node, responsible for orchestrating and invoking most of the other modules. Regarding offloading heuristics, a specific serverless function was developed for each offloading heuristic, which is, *ranking-offloading* and *duration-offloading*. Both were written in Java and implemented the algorithms presented in Subsections 4.4.2 and 4.4.3, respectively. The *predictor* function is written in Python and uses the *statsmodels*[4] library version 0.13.2, which provides several off-the-shelf methods for making forecasting and predictions. Also, the documentation for some of these methods references the Forecasting: Principles and Practices book, which has inspired us when choosing the most appropriate forecasting technique. This *predictor* function is generic in a way that it is not directly related to the health field but acts as a wrapper for the *statsmodel* lib, by exposing some of its features via HTTP. The *topology-mapping* function, in turn, is written in Java and is responsible for returning the IP address of the target destination for offloading operations. Since the idea is to avoid re-deploying the function on each fog node whenever a mapping between fog nodes changes, or the IP address of a given fog node changes, this function locates the mappings from a *properties* file located inside an Amazon S3[5] bucket. This way, once this file is updated, all fog nodes have an updated view of the topology and connections between fog nodes. Finally, the *ingestion-function* is written in Python and is only deployed to AWS Lambda, since its goal is to consume messages originating from the queue in the cloud, and there is no need to build and deploy this function to OpenFaaS and run them on fog nodes. We also emphasize how the *Cloud API Adapter* was introduced for the experiments: in practice, since we use AWS self-managed services, we did not have to write any piece of code to receive a vital sign by HTTP on the cloud and store the message on the queue. We only needed to create and configure an instance of AWS API Gateway[6] and connect a route to the queue created

[2]https://quarkus.io
[3]https://graalvm.org
[4]https://statsmodels.org
[5]https://aws.amazon.com/s3
[6]https://aws.amazon.com/api-gateway

using Amazon SQS[7], which worked out of the box. Another benefit is that API Gateway can handle many requests simultaneously and also works with high usage spikes. In practice, we accessed AWS Console, collected the URL regarding the API Gateway, and included this URL on the mappings file for the *topology-mapping* function. This allowed the last fog node of the hierarchy to offload vital signs to the URL of the AWS API Gateway. Following this same rationale, we did not implement any particular source code to store messages received from the API Gateway in a given database or in a queue, as the data in the queue is automatically persisted and works in a self-managed manner.

Metrics store: this module is written in Python and exposes an endpoint that will only be invoked by the *service executor* module with information about the execution. During the experiment, several pieces of information, such as the user priority, the calculated ranking, the CPU observations, and the CPU collection timestamp, among others, will be sent to this module. For the experiments presented in this thesis, there is no need to store the received data in a database, as results are collected from the memory right after the experiments. This module provides different endpoints to perform the following operations: collect CPU metrics, collect memory metrics, collect metrics summary, clear metrics, and send metrics. The endpoint to collect CPU metrics returns an array with all CPU collections for the fog node, combining the CPU observation with the timestamp when the observation was collected. This allows the generation of time-series charts for a given fog node. Given that this module runs on port 9001 by default, an example of sending metrics to this module is by making a POST request to the */metrics* endpoint with the given payload. For example, POST `http://{{REMOTE_HOST}}:9001/metrics` with the given request body:

```
{
    "user_priority": 3,
    "ranking": 8,
    "used_cpu": 96,
    "last_cpu_observation": 98.23,
    "cpu_collection_timestamp": 1663527788128,
    "function": "foo-function",
    "offloading": true,
    "running_locally": false,
    "exceeded_critical_cpu_threshold": true,
    "triggered_heuristic_by_ranking": false,
    "result_for_heuristic_by_ranking": false,
    "triggered_heuristic_by_duration": false,
    "result_for_heuristic_by_duration": false,
    "assuming_fallback_for_heuristics": false
}
```

There are endpoints to collect specific types of information, which are useful depending on the type of chart that will be generated. To collect data regarding the usage of CPU, for

[7]https://aws.amazon.com/sqs

example, the contract is to send a GET request to *metrics/cpu*, such as GET `http://{{REMOTE_HOST}}:9001/metrics/cpu`. An example response will be presented as follows. The *cpu* field represents the actual CPU percentage when the vital sign was ingested, compared with the thresholds to invoke offloading heuristics. The *last_cpu_observation* field returns the last CPU observation, but this is not the value compared with thresholds, as the approach of smoothing the CPU based on the last few observations requires different observations. This field is used to plot charts that compare the difference between the smoothed CPU and the last CPU observation. Since this endpoint returns an array, each JSON entry in this array represents a vital sign that was ingested. In practice, this endpoint may return thousands of entries depending on the number of vital signs generated by the experiment. Finally, the *collection_timestamp* field represents the moment when the CPU was collected and therefore allows generating time-series charts based on this information.


```
[
    {
        "cpu": 7.878125,
        "last_cpu_observation": 1.0,
        "collection_timestamp": 1671389293714
    },
    {
        "cpu": 7.918273,
        "last_cpu_observation": 7.20,
        "collection_timestamp": 1671389293743
    }
]
```


This module also exposes an endpoint to collect a summary of the metrics, which is especially useful when generating charts. The contract is to send a GET request to */metrics/summary* with the user priority as a query parameter. For example, GET `http://{{REMOTE_HOST}}:9001/metrics/summary?user_priority=5` will retrieve a summary for the metrics collected for vital signs regarding users with *very critical* priority, while an example of the result is presented as follows:


```
{
  "response": {
    "total_offloading": 15,
    "total_local_execution": 0,
    "total_exceeded_critical_cpu_threshold": 15,
    "total_exceeded_critical_mem_threshold": 0,
    "total_triggered_heuristic_by_rankings": 0,
    "total_result_for_heuristic_by_ranking": 0,
    "total_triggered_heuristic_by_duration": 0,
    "total_result_for_heuristic_by_duration": 0,
    "total_assuming_fallback_for_heuristics": 0
  }
}
```

Finally, it is also possible to clear all previously collected metrics before running the experiments without restarting the virtual machines by sending a POST request to the */clear* endpoint. For example, POST `http://{{REMOTE_HOST}}:9001/clear` will release all data in the memory. The return of this endpoint is a simple empty JSON, as there is no such information to be returned, and there is no information to be sent on the request body when invoking this endpoint. We emphasize that the aforementioned endpoints were not specified in Chapter 4 because they are specifically used for our experiments, since we are willing to generate charts based on this data. The only requirements for this module are to receive metrics and allow to retrieve these metrics afterward, but the design of the operations and the endpoints are specific to our implementation. This is why they are only detailed in the current chapter.

Results module: this module is analogous to the *notificator* module and is written in Python, running on a specific machine exclusively employed to receive execution results. In other words, this module does not run on a machine that is part of the hierarchical architecture, although it is physically located on the fog layer. The *results* module is different from the *metrics* module because the goal of the *results* is not to store metrics regarding how the vital signs were ingested. Instead, the goal is to store information when the vital sign is finally processed by a health service, which allows us to have knowledge of how long each vital sign took to be processed. For example, our implementation of this module provides a */results/summary* endpoint that returns a JSON object summarizing how many vital signs have already been ingested and how many are being processed. By *running* requests, it means a vital sign that has already been artificially generated and sent to a fog node on the first level of the hierarchy, but no health service was executed with this vital sign yet. This means that the vital sign is navigating on fog nodes or is waiting to be processed in the queue on the cloud, for example. One can also check how many vital signs have already been artificially generated by edge nodes by subtracting the expected amount of vital signs for the experiment from the sum of the total running requests and the total finished requests. This endpoint does not differentiate vital signs that are navigating on the fog nodes or waiting to be processed on the queue in the cloud, as it considers both as pending requests. Finally, when *total_running_requests* is zero and *total_finished_requests* is equal to the expected number of vital signs for the experiment, we know that the execution of the experiment has completed.

```
{
  "total_running_requests": 7590,
  "total_finished_requests": 72410
}
```

When the edge node (analogous to a smartwatch) is close to sending a vital sign to the leaf fog node, it first sends a notification to the *results* module indicating that a given vital sign has been registered and is pending to be processed. This means that the vital sign is in *running* state. Also, each vital sign has a unique identifier, which is further used to associate the final timestamp when the vital sign is finally processed. The contract to register a vital sign in *running*

state is to send a PUT request to */results/{{ID}}*, such as PUT `http://{{RESULTS_NODE}}:9095/results/{{ID}}` with a payload such as the following:

```
{
  "service_name": "heart-failure-predictor",
  "user_priority": 5,
  "start_timestamp": 1663527788128,
  "vital_sign": "{\"heartbeat\": 97,\"spo2\": 99,\"temperature\": 40}"
}
```

When the vital sign is finally processed, either by a fog node o by a serverless function on the cloud, a PATCH request is sent to update the state of the given vital sign and indicate that the vital sign was already processed. The payload of this request should contain a field indicating the origin of the request (*fog* or *cloud*) to generate charts of where each vital sign was processed, and also contains the timestamp regarding the end of the execution. An example of a request is to send PATCH `http://{{RESULTS_NODE}}:9095/results/{{ID}}` with the following payload:

```
{
  "origin": "cloud",
  "end_timestamp": 1669635936775
}
```

It is also possible to clear previous results so that the virtual machines do not need to be restarted (or the process needs to be finished on the operating system) to clean up results and start a new experiment. The contract is similar to the *metrics* module, where a POST request is sent to the */clear* endpoint without any request body. Finally, it is also possible to retrieve the complete dataset with details of all vital signs being ingested or already processed. The contract is to send a *GET /results*, and the response will be something such as the following:

```
{
  "running_requests": {
    "9928987bf-9330-4355-8fe2-00ab1f7420bf": {
      "service_name": "heart-failure-predictor",
      "user_priority": 5,
      "start_timestamp": 1663527788128,
      "vital_sign": "{\"heartbeat\": 97,\"spo2\": 99,\"temperature\": 40}"
    },
    "9928987bf1-9330-4355-8fe2-00ab1f7420bf": {
      "service_name": "heart-failure-predictor",
      "user_priority": 4,
      "start_timestamp": 1663527913148,
      "vital_sign": "{\"heartbeat\": 97,\"spo2\": 99,\"temperature\": 40}"
    }
  },
  "finished_requests": {
    "c3754270-c173-4be0-b01e-f2f3b290d57c": {
      "service_name": "heart-failure-predictor",
```

```
      "user_priority": 4,
      "start_timestamp": 1663527913148,
      "end_timestamp": 1669635936775,
      "result_received_at": 1673828223258,
      "origin": "fog",
      "vital_sign": "{\"heartbeat\": 97,\"spo2\": 99,\"temperature\": 40}"
    }

  }
}
```


CPU provider: this module is written in Python and is responsible for providing the CPU percentage on the fog node. The contract of this module is to return immediately so that the processing of vital signs does not need to wait until results are available. This way, the process of collecting CPU usage percentage (which can range from milliseconds or seconds, depending on the approach) must be done in the background in a way that this module returns the most recent value. It exposes a single GET */machine-resources* endpoint, which returns the information required for decision-making on the fog node.


```
{"cpu": 14.4296875, "last_observation": 15.3, "memory": 68.3}
```


We highlight the existence of the *memory* field on the response for this module. For troubleshooting purposes, we needed to collect and monitor the amount of memory available in the fog nodes during the experiments, although no offloading decision was made based on this information. This was only used for visualization and debugging purposes. For this reason, this field is included in the response of the *cpu provider* we implemented but is not expected on the definition of this module in Subsection 4.3.3.5. Still, future work may consider this field as a second parameter for triggering the heuristics instead of always relying on CPU, e.g., triggering offloading heuristics when the used memory is above 90%, regardless of the CPU usage being low. The value of 90% was simply used as an example in this paragraph, and future studies could determine a more appropriate value. The response of this module also indicates an additional field named *last_observation*. As will be explained in Subsection 5.4, different approaches for CPU collection are evaluated, while one of them uses a smoothing technique based on the last few CPU observations. To this end, the *cpu* field is the one which *service executor* uses for the offloading decisions, and the *last_observation* field indicates the last CPU observation collected when using the smoothing approach but is not used for making decisions. In practice, for our experiments, this field is only used for plotting charts by comparing the difference between the smoothing CPU collection and the naive approach, which only considers the last CPU observation. For the SmartVSO model, it is not mandatory that the *cpu-provider* module returns any other field in addition to the *cpu* field.

Health services: two health services were developed for this prototype. Both were written in Python because of the ease of handling JSON data compared to other languages and because developing serverless functions in Python requires less code for the OpenFaaS platform

and also for AWS Lambda. The *body-temperature-monitor* health service is responsible for receiving a vital sign and indicating whether the person has a fever, hypothermia, or none of them. This function is lightweight because it does not make any prediction based on historical values and simply makes an *if* condition on the temperature field extracted from the vital sign JSON structure. The *heart-failure-predictor* health service is responsible for telling whether the person will have heart failure in the next few minutes based on the previous heartbeats and SPo2 levels. For the prototype evaluation, this health service considers a hardcoded list of previous values collected for both heartbeat and SPo2 and forecasts both considering the actual values from the vital sign received on the function's input. If the predicted values are within a certain range, the person may have heart failure in the next few minutes. We emphasize that both services have only been developed for evaluation purposes and require medical validation before being used in production. Whenever a health service completes its execution, the result telling if the person has a health problem is returned as the response of the serverless function, which is collected by the *service executor* and forwarded to the *notificator* module. Since the proposed computational model runs health services on fog and cloud environments, we needed to implement the functions for both OpenFaaS and AWS Lambda platforms. The core logic of the health service remains the same, but there are slight differences regarding the serverless function's entry point. Even though both receive an *event* and a *context* parameters, the types of the objects are different, and the vital sign needs to be collected from different fields within the input. Another difference between OpenFaaS and AWS Lambda is the deployment approach, mainly because of the size of the serverless functions on AWS Lambda. Machine learning models that use external dependencies may become large and exceed the size limits of AWS. The library *statsmodel* on version 0.13.2, for example, depends on additional packages to work. Together, these packages result in 308,4 MB in size, while the limit for a serverless function on AWS is 262,144 MB when decompressed from a .zip file. This is a challenge because we needed alternatives to deploy machine learning models on the cloud, such as using layers or deploying functions as container images. The first approach was to directly upload a zip file on the AWS Lambda through the AWS console, but since the .zip file with function logic and dependencies is 75,7 MB in size, it was not possible to upload the zip file directly on the AWS console for Lambda. We needed to upload this zip file to the Amazon S3 bucket and then reference the bucket during the lambda deployment. However, since the decompressed size exceeded the above limit, we needed to use a different approach. This approach deployed the container image to Amazon ECS[8] (Elastic Container Registry) and further referenced this image to deploy the AWS Lambda function.

Evaluation tools: in addition to developing the modules and heuristics mentioned above, a big challenge faced in this thesis was to develop tools that analyze the behavior of these modules. Once an arbitrary experiment was executed, we needed a tool that checks if the right amount of vital signs had been really processed, the amount of offloading operations, the re-

[8]https://aws.amazon.com/ecr

sponse time to ingest and process a vital sign, the number of local executions, the CPU load when the vital sign was being consumed, among other metrics. The developed modules needed additional code to send these metrics to an external data source, which in this thesis we introduced as the *metrics store*, so these metrics can be further collected and evaluated to validate the system's behavior. Also, as we are dealing with a distributed system composed of several fog nodes, these tools automate the retrieval of this information on the *metrics store* located on each machine. Manually accessing each VM through SSH (or equivalent protocol) was time-consuming and became even more difficult as experiments were done with more fog nodes. With this in mind, we also developed Python scripts to automate the deployment of services, execute the experiments, and plot charts based on the metrics collected after the experiment was completed. Another challenge we faced during the experiment is that each VM has a dynamic IP address that is renewed whenever the VMs are rebooted. This would require changes to different parts of the system that need communication with the fog nodes, including the *service-executor* module for offloading operations and the evaluation scripts to submit vital signs to the fog nodes, as well as remotely re-deploy functions with different thresholds, depending on the experiment. The developed evaluation tool uses the AWS SDK to mitigate this problem, which allows us to list all existing virtual machines (fog nodes) associated with the Amazon account and get their current IP addresses. We also attached a *name* Tag to each VM to filter on the requests made by the SDK. All fog nodes were configured to have a name starting with the *fog_node* prefix, so we can get the required IPs by using the mask *fog_node_** to collect information about all fog nodes and ignore any other existing virtual machine that is not a fog node. When plotting charts with the execution metrics, our first approach was to manually copy results and paste them to Google Sheets, where charts could be plotted. However, some charts required data pre-processing and data aggregation, which was difficult and time consuming to be done manually. To automate this process, we decided to use a Python library called Matplotlib[9] to plot charts based on the pre-processed data. The evaluation tools were segregated into Python modules, presented in Table 7 along with their descriptions. Every execution starts with the *evaluation* module, which in turn uses the other modules to generate analysis based on the results. Also, the evaluation tool performs the following main steps to initiate the execution and collect the results. Finally, additional and specific steps were also performed but are not described for simplicity reasons. The only requirement is that fog nodes are already up and running:

1. Locate the name, public, and private IPs from fog and edge nodes using the AWS SDK;
2. Update the CPU collection interval;
3. Authenticate on the remote OpenFaaS running on the fog node;
4. Re-deploy the service-executor module with current warning and critical thresholds;

[9]https://matplotlib.org/

5. Re-deploy topology-mapping module with the node alias from the name of the VM based on the AWS console;

6. Warm-up serverless functions to avoid HTTP 500 internal errors from FaasD due to non-running functions.

Table 7: Modules implemented on the evaluation tool.

Name	Description
assertions	Performs assertions on the metrics to ensure that vital signs were processed successfully
aws	Provides functions to collect VMs information and interact with AWS resources
evaluation	The entry-point for execution of experiments
metrics	Initializes metrics on remote fog nodes and collects metrics at the end of the experiment
plot	Plots charts based on the previously collected metrics
properties	Generates the properties file with connections between fog nodes
summary	Summarizes information based on the results collected from the fog nodes
warm	Warms functions to avoid HTTP 500 errors with non-initialized functions on FaasD

Once the steps mentioned above have been performed, all of the VMs are prepared for the execution of the test scenario. The tool we developed for running these tests also helps achieve reproducibility, as the experiments can be re-executed without much manual effort. Finally, this tool helps execute the experiments with several parameters and thresholds, so we can understand how the system behaves under different workloads and settings. The following steps are performed for each test scenario:

1. Clear in-memory metrics on each fog node;

2. Invoke tests by sending vital signs to fog nodes on the first level of the hierarchy;

3. Await until all vital signs have been processed;

4. Collect CPU usage and additional metrics on each fog node;

5. Print a summary of the execution to the log;

6. Plot several charts based on the collected information.

5.2 Infrastructure for tests

We chose AWS to support our infrastructure because of the authors' familiarity with this cloud provider, as well as because other scientific works use this infrastructure with successful results (NATH et al., 2022; SANCHEZ-GALLEGOS et al., 2022). Evaluations are made in an emulated environment composed of virtual machines representing fog nodes because we do not have a real environment with physical machines distributed among neighborhoods to perform

our experiments. With this in mind, this emulated environment must behave as much as possible like a smart city, so we need to consider a virtual infrastructure analogous to what would happen in a real city. Virtual machines representing fog nodes should be physically close to the virtual machines responsible for artificially generating vital signs, to result in short response times, as would happen with people sending vital signs to their closest possible fog node. In contrast, resources on the cloud are physically distant from people in a smart city. This is the reason why we intentionally place resources regarding the fog layer in São Paulo (*sa-east-1*) and resources regarding the cloud layer in London (*eu-west-2*), which will result in higher latency and higher response time. We also emphasize that, in Brazil, AWS only offers computing resources located in São Paulo[10]. Additionally, in this thesis, we use the term *edge nodes* to represent machines that artificially generate vital signs to fog nodes, which in a smart city is analogous to people sending vital signs to a fog node physically located in the same neighborhood.

Different architectures are evaluated with a specific number of fog nodes each and will be detailed in the next paragraphs. For organization and simplicity purposes, evaluations were first done with a simple architecture composed of a single edge node, a single fog node, and serverless functions running in the cloud. Further evaluations evolved to use more robust architectures composed of interconnected fog nodes in order to understand how SmartVSO would behave in a city with a higher concentration of people in specific neighborhoods, as well as with asynchronous processing of vital signs in the cloud. The final version of the architecture is the outcome of several improvements made after running experiments and iteratively analyzing their results, as explained at the beginning of this thesis in Figure 1. We emphasize that three different architectures were evaluated in the experiments. In summary, this thesis originally proposed the architecture *A*, but we noticed during initial evaluations that this simpler architecture could not handle a more significant number of vital signs arising simultaneously. The architecture then evolved until we could introduce an architecture that could handle such workload. This is why different architectures are presented in the following paragraphs and are evaluated throughout the experiments, being architecture *C* the final version presented in Chapter 4.

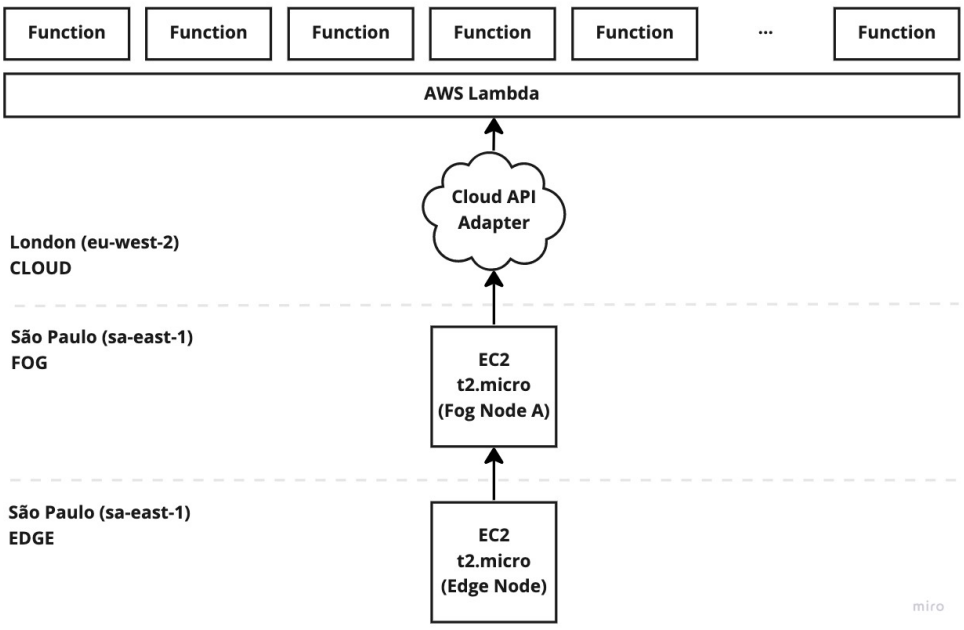
Regardless of the architecture, all edge and fog nodes allocated in AWS are physically located in São Paulo (*sa-east-1*) and are represented by virtual machines with Elastic Compute Cloud (EC2) instances in this region, while the EC2 instances used in experiments are of the type *t2.micro* having 1 vCPU and 1GB of RAM. We chose this type of instance to reduce costs because machines with more CPU cores and RAM are more expensive. Also, this type of instance is part of the Free Tier in Amazon AWS and, to some extent, provides free resources, as will be detailed in Section 5.3. The serverless platform we use for running functions in the cloud is AWS Lambda, and the queue for storing messages in the cloud is AWS Simple Queue Service (SQS). Serverless functions are connected to SQS and are automatically triggered as messages arise in the queue, so we did not need to implement a middleware that polls messages from the queue and triggers the Lambda functions. The self-managed AWS services automatically

[10]https://aws.amazon.com/pt/about-aws/global-infrastructure/regions_az/

provide this integration. Similarly to what happens in a smart city, communication between fog nodes and the cloud (AWS SQS or AWS Lambda) is expected to result in higher latency when compared to smartwatches sending vital signs to fog nodes.

Architecture A: this is the most straightforward architecture evaluated and is presented in Figure 19. Vital signs are generated from a single edge node and are forwarded to a single fog node. This architecture was beneficial during the first experiments because debugging an architecture with few nodes is more manageable. However, it may not represent real-world usage in a smart city. In summary, this architecture was first used to help us build the solution and ensure that the main components were working as expected and the offloading behavior was also working correctly. For this architecture, the Cloud API Adapter has been implemented as a serverless function with AWS Lambda. This adapter is analogous to the *service executor* module running on fog nodes. They implement the same API contract and are responsible for receiving vital signs and spawning serverless functions to process these vital signs. However, the Cloud API Adapter is much simpler than the *service executor* because there are no heuristic or offloading operations involved because (in theory) the cloud benefits from virtually infinite computing resources. This module receives the vital signs and synchronously triggers health services as serverless functions. However, in practice, we were surprised that these virtually infinite computing resources are limited by quotas available for the account on the cloud provider, as will be detailed in the results during the experiments. Pros of this architecture include the ease of debugging and doing the first experiments, but cons are that this architecture cannot handle such a massive workload in vital signs in a smart city. Finally, experiments with this architecture were done with JMeter.

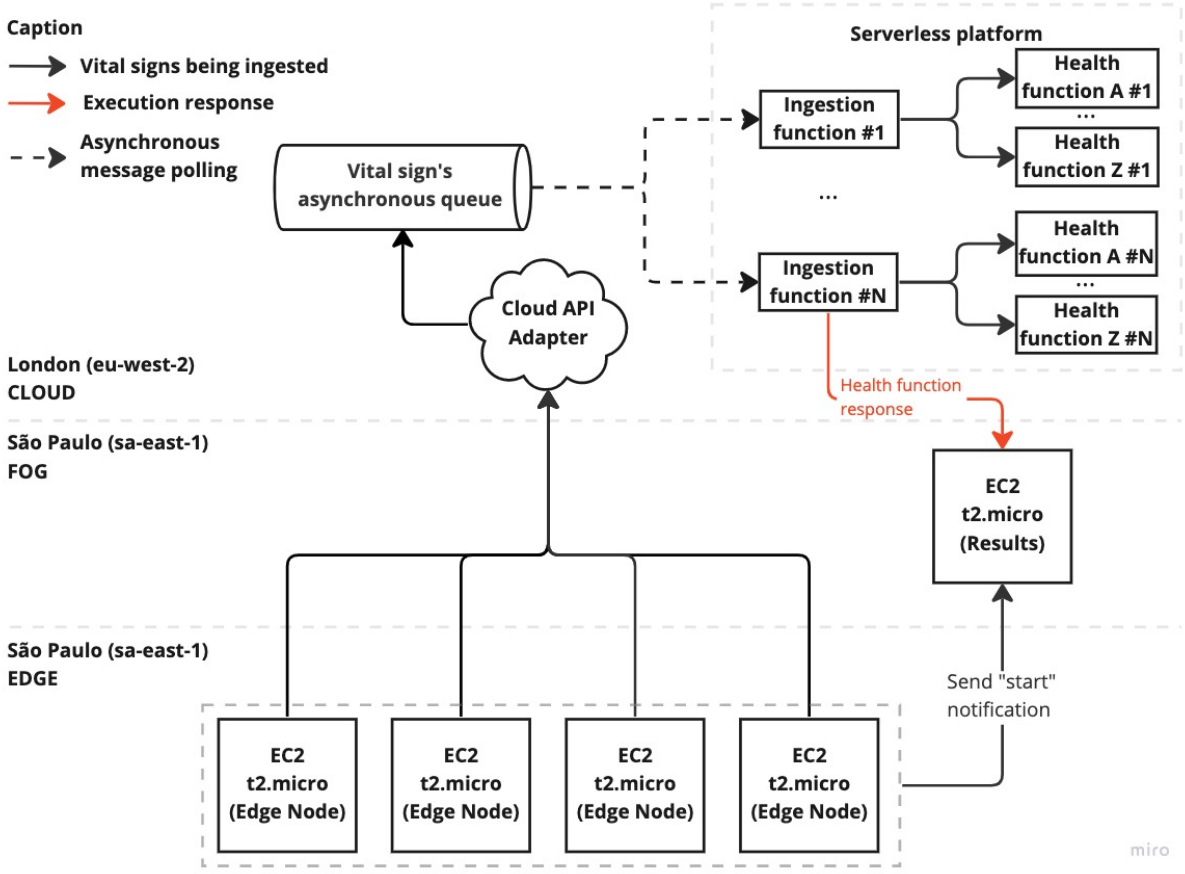
Figure 19: Architecture A with a single fog node and synchronous processing on the cloud.



Source: prepared by the author.

Architecture B: Figure 20 presents a different architecture composed of four edge nodes (representing several people in a smart city) that send vital signs to the queue on the cloud. Compared to architecture A, the main difference is that this architecture processes messages asynchronously and is not limited by the quotas of simultaneous serverless functions running concurrently. The queue increases in size and the messages are consumed at a speed that the lambda functions can ingest. The goal of this architecture is to have a comparison with architecture C, which also processes messages asynchronously on the cloud using a queue and uses fog nodes to process critical messages synchronously. Results for these architectures are further compared to understand how better the results are when fog nodes exist in a smart city. With this architecture, it is expected that all vital signs share similar response times since this architecture does not favor critical user priorities and ingests vital signs as they arrive in the queue. The pros of this architecture include handling a huge amount of vital signs. In contrast, the cons include higher response times due to the physical distance between the edge nodes and the cloud and the fact that no heuristic prioritizes critical vital signs being processed with critical health services.

Figure 20: Architecture B without fog nodes and asynchronous processing on the cloud.



Source: prepared by the author.

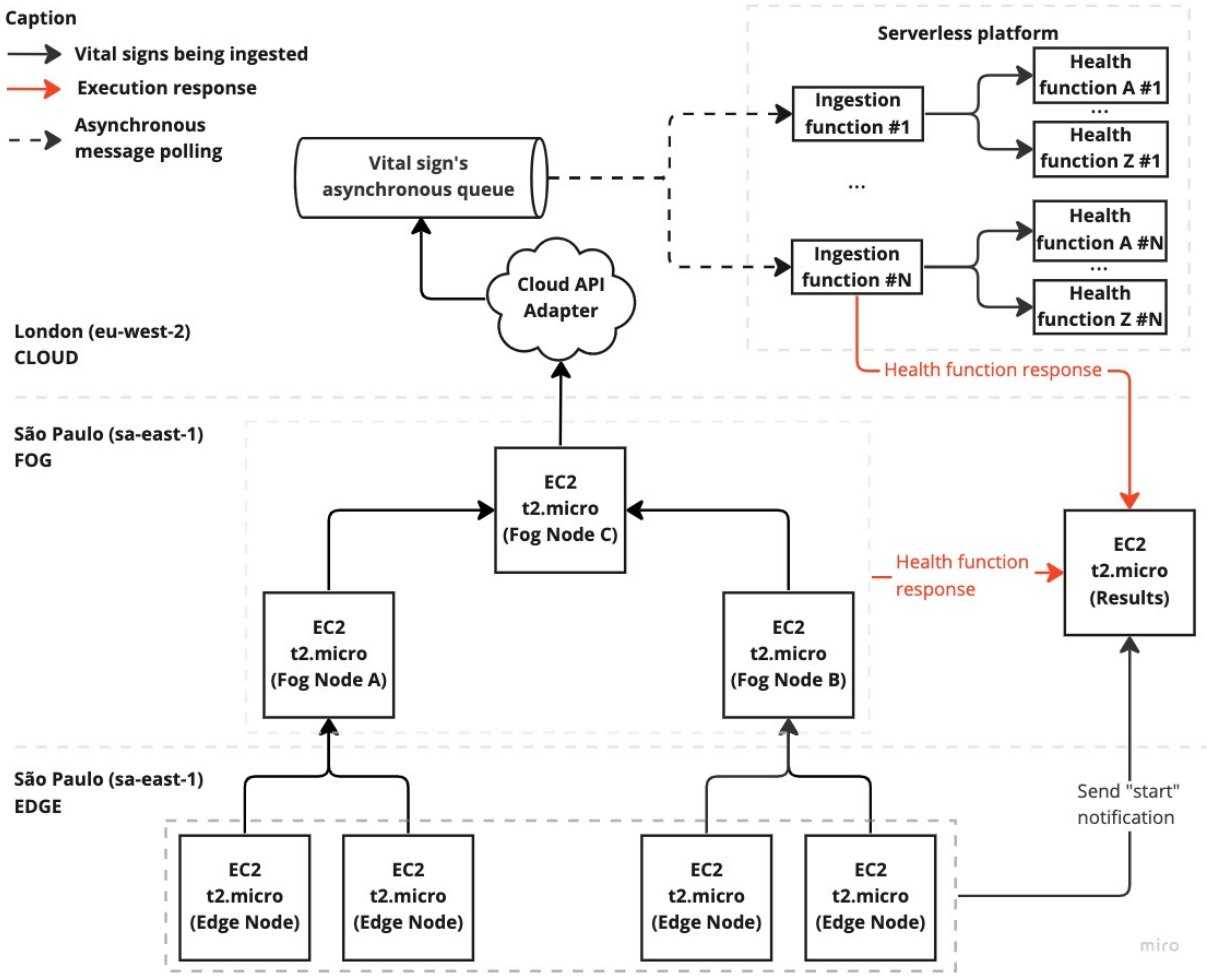
This architecture has an additional component not present in architecture *A*. This component is a virtual machine responsible for receiving the responses after the vital signs are processed. This is important because we need to identify how long each vital sign takes to be processed, as it happens asynchronously. Architecture A is simpler to develop because the response time for synchronous responses would be the time between sending a vital sign (establishing an HTTP connection) and receiving the HTTP response. However, since architecture B allows messages to be added to a queue on the cloud, the moment the HTTP request is completed does not mean that the vital sign has already been processed. To deal with this scenario, we need a virtual machine exposing an endpoint that receives and stores the results. This required a modification on the initial version of the proposed modules, as they also need to send the response to this endpoint. In addition, each message now requires a unique identifier since the execution is asynchronous. This is needed to calculate the total time between the moment the message was sent and when the message was processed. The response is sent to an in-memory data store in the same region (São Paulo) as the fog nodes are located instead of being stored on the cloud, due to the proximity with the user. Supposing that the health service sends a notification to the user's smartphone, the vital sign will need to go back to the physical user's location, and the latency will come into account again. To make this experiment as close as possible to real-world usage, we decided to place this *Results* machine in the same region as the fog nodes. Finally, as we wish to consider the complete response time, including the network latency, we need to collect the timestamp when the response arrives on the *results* module and not when the health service is processed (in the cloud, for example).

As soon as the health serverless function completes its execution, the *ingestion* function sends a notification indicating that the vital sign has been successfully processed. This is important because this information serves as the basis for the analysis that will be presented in Chapter 6. Finally, it is important to notice that a *start* notification is sent to this *results* module before sending the vital sign to the cloud, which contains the timestamp of the moment when the vital sign started to be handled. This is further used to calculate how long it took for the vital sign to be processed by subtracting the finish timestamp from the start timestamp. Finally, experiments made with architecture A were done with JMeter, but we needed to create a specific script for architecture B to deal with asynchronous processing. Although JMeter offers advanced mechanisms to customize requests, we decided to make a Python script to send multiple vital signs because of our familiarity with this language instead of diving deeper into writing plugins for JMeter. Also, since the processing now happens asynchronously, we need to assign a unique identifier to each vital sign to further calculate how long it took to ingest the vital sign. Before artificially generating the vital sign, the evaluation script must register the timestamp at which the execution started. This needs to be done precisely before the vital sign is sent. Finally, as the execution happens asynchronously, the evaluation script does not assume that the response time for processing the vital sign is the HTTP request duration.

Architecture C: this is the most robust architecture because it combines fog computing with

asynchronous processing in the cloud. This is the architecture presented in Chapter 4 of this thesis and is a result of several improvements after analysis made on the previous architectures. At first glance, we were unaware of some limitations regarding concurrent replicas of serverless functions on AWS, which invalidated architecture *A* to some extent. We needed to reformulate our idea around how to ingest vital signs and decided to include a queue in the cloud to process messages asynchronously. This architecture then represents the final version after several improvements made during the experiments. Pros of this architecture include processing critical vital signs with low response time in the fog nodes due to the short physical distance between people and the fog layer, while cons include not having a prioritization of messages in the queue when dealing with high usage peaks and fog nodes need to offload some critical vital signs to the cloud. Figure 21 is similar to the architecture presented in Chapter 4 but highlights different edge nodes sending data to leaf fog nodes, while edge nodes are analogous to smartwatches.

Figure 21: Architecture C combining fog nodes with asynchronous processing on the cloud.



Source: prepared by the author.

The *Cloud API Adapter* module is responsible for receiving the vital signs from the last fog

node and persisting them in a queue on the cloud. We emphasize that this module has the same API that the *service executor* has on the fog, which makes the solution generic. There is no *if* condition in the *service executor* module to introduce a specific behavior when offloading vital signs to the cloud. In practice, the *service executor* module does not even know it is sending vital signs to the cloud: it simply sends the vital sign to the HTTP endpoint of the upper layer, which for the last fog node is the HTTP endpoint of the *Cloud API Adapter*. Regarding the queue details, it is resilient and not volatile, so we do not manually interact with any virtual machine holding data in the memory. Introducing a queue to process additional messages on the cloud is interesting because consumers can work at their own speed, even though it is slower than the speed of the producers. This solves the problems faced by Architecture A, where vital signs were lost because of throttling errors, at the same time the current architecture is able to favor vital signs with high priority.

This architecture has some trade-offs, especially between increasing the number of serverless functions replicas in the cloud and including extra fog nodes in the neighborhoods. The main trade-off is the cost. In case most vital signs generated in a given neighborhood are from healthy people, this means that the vital signs are not so critical. Therefore, it is acceptable to have a longer response time and take a longer time for the queue to be consumed, so sending them to the cloud is an exciting option. On the other hand, a longer response time may not be accepted if the neighborhood concentrates hospitals or has more people with health problems. On this scenario, the most appropriate solution would be to include fog nodes on the hierarchy. The costs can be different for each approach; therefore, each neighborhood needs to be analyzed in particular. Also, when choosing the number of fog nodes, an essential factor to consider is the total hops between machines until the vital signs arrive in the queue located in the cloud. This differs from sending all vital signs directly to the cloud, as each hop imposes extra network latency. When having fog nodes, it takes longer for the vital sign to reach the cloud because it needs to navigate through different fog nodes, as only the last fog node on the hierarchy is configured to send data to the cloud. In other words, if there is a huge spike of vital signs arising to fog nodes, and these machines cannot process all vital signs locally, there is a chance of them taking longer to be processed than directly being offloaded to the cloud without navigating through fog nodes.

This architecture uses all of the available function replicas on the AWS account as consumers of the queue. If needed, one might request an increase of AWS Lambda quota on the service account to have more simultaneous executions and, therefore, more queue consumers. Let us suppose that a large number of vital signs is generated in a given neighborhood, but fog nodes cannot process all vital signs. In that case, the queue can grow faster than the consumers can process the messages. To this end, a higher number of lambda quotas can be requested from the cloud provider support (in our experiments, AWS) so that more lambda replicas are available to process vital signs simultaneously. Another approach to increase throughput is investing in more infrastructure on the fog by placing more physical machines in the neighborhood. Fewer

vital signs will be offloaded to the cloud and processed physically close to the person, reducing response time. Finally, one approach does not eliminate the other, so more fog nodes can be placed in the neighborhood, and the number of serverless function replicas can also be increased by contacting the cloud provider support.

5.3 Cost analysis

All experiments were made using AWS Free Tier[11], which provides free computing resources as long as certain conditions are met. This includes the type of EC2 instances, the amount of allocated storage, the number of serverless functions triggered during the current month, and the number of messages stored in the queue during the current month. However, the Free Tier was not enough for all experiments. It resulted in additional costs, especially regarding storage limits and the time spent with virtual machines turned on. The total cost for running the experiments for this thesis in AWS was R$ 187,36.

5.4 Defining the parameters

The process of choosing the correct parameters for the experiment is a challenge. Multiple parameters are considered in SmartVSO, such as warning and critical CPU thresholds and approaches to collect CPU observations. However, their exhaustive combination results in a large matrix of possibilities that would require much effort to be tested. With that in mind, we based our choices on similar work and experimented with thresholds that make more sense in the context of a smart city, even though they are not entirely based on the literature. We set initial thresholds and iteratively calibrated them based on the generated results: several experiments generated unexpected results that did not favor users with critical priorities, but they guided us to improve SmartVSO until it could converge to better results and handle a larger number of vital signs. The most relevant experiments are presented in the incoming sections, even though the first ones do not present the most appropriate results. We explain why we believe that some results are not appropriate and other results are appropriate. The following subsections will dive deeper into how we chose the approach to collect CPU load and select the warning and critical CPU thresholds. Finally, we also emphasize that specific parameters may fit better than others depending on the workloads and the type of health service, such as services being more CPU bound than others. We suggest future work to address a dynamic calibration of these values to make the system adaptive to workload patterns.

The most relevant computing resource monitored by the proposed heuristics is the CPU load, while the time interval for CPU load collection is critical. Our experiments indicate that the smaller the configured time interval for CPU collection, the faster our system detects changes in the usage of the CPU. For example, suppose that the CPU is mostly idle for an extended

[11] https://aws.amazon.com/free

period, and a sudden, huge workload spike arises in the last second. If the CPU collection interval is set to 1 second, our system will understand the CPU usage being close to 100%, as it only considered the time window of the spike and ignored the idle CPU time. If the CPU interval is set to 2 seconds, our system will understand the CPU load being close to 50% during this time interval, as the CPU was idle during half of the time and busy during the other half. The details above about CPU collection do not mean that a specific interval is better than the other in all scenarios, though. It depends on how fast the system wants to react to changes in CPU and how long the health services take to complete. Finally, it also depends on the number of vital signs received in a time interval because it directly impacts how many instances of health services will be executed. Each scenario needs to be analyzed separately. Identifying a CPU interval that suited the workloads we used in the experiments was a big challenge. We conducted experiments with different CPU collection intervals and analyzed their results to find an appropriate one. We suggest a dynamic calibration of the CPU collection time interval in future work because they do not necessarily fit all the workloads of a smart city. Finally, different approaches exist to consider the CPU load on the operating system. Some methods only consider the last CPU observation; others use statistical techniques to smooth the last observations and prevent fast reactions to the CPU load, and others consider multiple threads to collect the CPU with a time shift between threads. At first glance, we were unsure which approach would best fit our solution. We tested different techniques to understand their impact on the throughput, and the following subsections will give details about them. We emphasize that the collected CPU load is regarding the fog node and is not related to the CPU load of the container where the health service (serverless function) is executed.

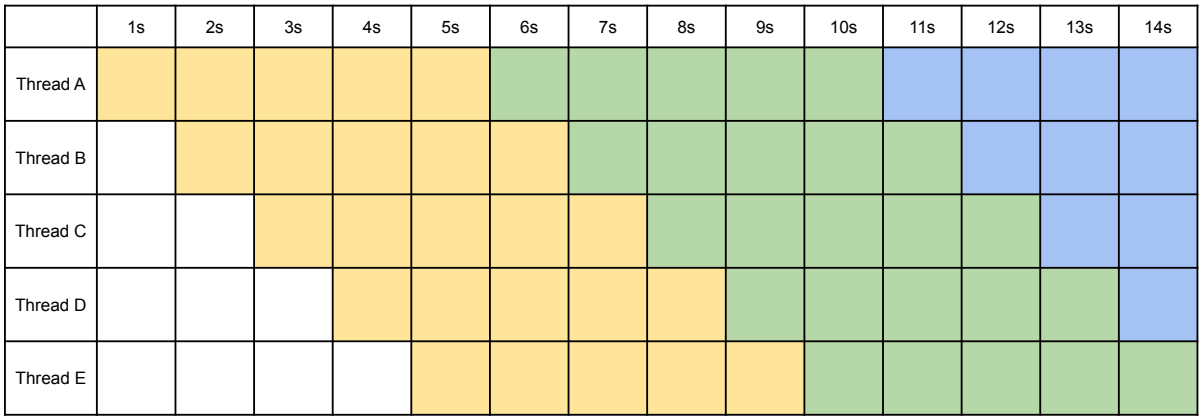
Single thread for CPU collection: this mechanism consists of a single thread responsible for updating, in the background, a variable with the CPU usage during the time interval of x seconds. The value of x depends on each experiment, and different values were tested. This is the most straightforward and naive approach for collecting CPU for our prototype. Iteratively, the background thread updates this variable with an updated CPU duration after x seconds so that the heuristics can be triggered considering this last value. The main problem with this approach is that the system makes decisions based on a CPU load outdated by x seconds. In other words, the CPU value considered during the offloading decisions is at least x seconds late. We need a different approach to collect CPU that is not based on outdated results but, at the same time, is not so sensitive to usage spikes. The pros of this approach are that this is quite simple to be implemented but makes offloading decisions based on outdated CPU values.

Multiple threads for CPU collection: this second approach is similar to the previous one in that a variable with the CPU load keeps being iteratively updated in the background. However, instead of having a single thread, this mechanism combines multiple threads collecting the CPU load using the same interval with a time shift calculated with the Equation 5.1. In this equation, α represents the time shift for starting each thread, β represents the duration interval that every thread will use to collect CPU, and γ represents the number of threads. The time shift

for starting each thread is a division of the CPU collection interval by the number of threads. We noticed that a larger CPU interval is not highly sensible to the offloading operations and workload changes, which in specific scenarios is desired. Still, it results in a different problem: the delay in collecting the CPU. Setting a collection interval to 5 seconds, for example, makes the architecture very slow to understand changes in CPU load. Supposing the CPU collection interval is five and there are five threads, the time shift between starting each thread will be 1 second. Each thread will take 5 seconds to bring the CPU load result, but there will be a new result every second, while Figure 22 illustrates this behavior. Given this rationale, another example is to have a CPU interval of 5 seconds but have 10 threads collecting the CPU load. This means that each thread will be started 500 milliseconds after the previous thread, and there will be a new CPU observation every 500 milliseconds while still considering the CPU usage in the last 5 seconds. The pros of this approach are that CPU observations are available in a shorter interval when compared to the previous approach. However, this approach still considers a significant time interval which makes the solution slow to respond to changes in CPU load. For example, if there is a huge and sudden spike in vital signs, this approach will take a few seconds to detect.

$$\alpha = \frac{\beta}{\gamma} \tag{5.1}$$

Figure 22: Multiple threads collecting CPU observations with a time shift. The yellow blocks represent the time to collect the first CPU observation for the thread, while the green and the blue represent the second and the third intervals to collect further CPU observations, respectively.



Source: prepared by the author.

Aging to smooth CPU values: this approach considers previous CPU observations and exponentially gives a lower priority to the older observations instead of simply considering the last CPU observation. This follows the experiment used by Righi et al. (2016) that had successful results. The effect of this approach is the following: the most recent the CPU collection is, the higher impact it has on the calculated CPU usage percent. This technique smooths

possible noises on the CPU collection by giving higher weight to recent observations and generating the most relevant results for our experiments. This is also known as aging and uses the exponentially weighted moving average technique (TANENBAUM; WETHERALL, 2011; HYNDMAN; ATHANASOPOULOS, 2021). An example of how the CPU measurements look when using the aging technique is presented as follows (RIGHI et al., 2016). This works by exponentially dividing the value by exponents of 2 so that recent values have higher importance (HYNDMAN; ATHANASOPOULOS, 2021). For example, given the following CPU measurement values: 70, 85, 75, 40, 20, and 60. Considering the last measurement, naive offloading decisions would be made based on 60. However, if we apply the aging technique to these values, the result will be $\frac{1}{2}$.60 + $\frac{1}{4}$.20 + $\frac{1}{8}$.40 + $\frac{1}{16}$.75 + $\frac{1}{32}$.85 + $\frac{1}{64}$.70 = 48,44. The pros of this approach are that it is fast to compute and also considers the previous observations instead of only considering the last one, reducing noises caused by usage spikes on the workload.

Finally, different values were considered for warning and critical CPU thresholds during the experiments, which will be specified for each experiment in Section 5.7. We searched in the literature for similar work, and we could find some experiments that already make decisions by CPU thresholds (RIGHI et al., 2016). This served as an inspiration for our first experiments. We then analyzed the results and calibrated the thresholds for incoming experiments until we were satisfied. In other words, we did not define thresholds for all experiments before running the first one: this was a dynamic process where we calibrated values iteratively for the incoming experiments. Details about each CPU threshold are presented in section 5.7 because each experiment has a different set of thresholds.

5.5 Evaluation metrics

The main goal of SmartVSO is to reduce response time and to increase execution throughput when running health services with critical vital signs, but we emphasize that most offloading decisions are influenced by the percentage of CPU being used when vital signs are received. We choose the following metrics because they are directly related to the goal of this thesis, which is to reduce response time and increase throughput for high-priority vital signs, in a way that we can understand if the model is effectively solving the vital signs prioritization problem.

- **Response time:** we expect that response time is reduced when ingesting critical vital signs, especially when processing health services with high priority. This is a consequence of critical vital signs being processed on the leaf fog node, which results in lower response time than non-critical vital signs because of the physical distance between fog nodes. It is acceptable that response time increases for non-critical vital signs in specific situations to the detriment of favoring critical vital signs. The works proposed by Cheng et al. (2019) and García Villaescusa, Aldea Rivas, and González Harbour (2023) also analyze response time and latency, which serves as an inspiration for us;

- **Execution throughput:** we expect that execution throughput is maximized for critical vital signs, as well as for health services that complete faster when vital signs have the same calculated ranking. Critical vital signs should have the highest throughput due to the smaller number of offloading operations, while the least critical vital signs should have a lower throughput because of additional offloading operations to the cloud. The work proposed by Wang et al. (2020) also evaluates the throughput and represents an important metric for this thesis;

- **CPU load:** we monitor the percentage of used CPU to check if computing resources are being used appropriately. We expect the percentage of CPU used on the fog nodes to be within the *warning* and *critical* thresholds when triggering the heuristics, and to be above the *critical* threshold when directly offloading vital signs to the parent fog node or the cloud. The work proposed by Righi et al. (2016) also monitors the CPU usage during the evaluation by comparing it with thresholds, which serves as an inspiration for us;

- **Waiting time in the queue:** this is the amount of time a message waits in the queue in the cloud until it is processed after being offloaded from the fog nodes. This information is important because we can have a clear understanding of the impact of the queue on the response time, as the waiting time directly impacts the former metric. The work proposed by Jamil et al. (2023) also considers the waiting time in their evaluation to consume messages from queues. The only exception where this metric is not evaluated is for scenarios where a queue in the cloud is not used, as will be presented in Section 5.7;

- **Offloading operations:** we expect that high-priority vital signs are processed on a leaf fog node as much as possible. At the same time, we also expect non-critical vital signs to be offloaded to parent fog nodes on the hierarchy when CPU is getting overloaded on the leaf fog node. The only situation where it is acceptable that high-priority vital signs take a longer time to complete is when the leaf fog node is extremely overloaded and does not support the processing of additional vital signs, which is a scenario where even critical vital signs are offloaded. The work introduced by Zhao et al. (2023) also considers the number of offloading operations as an evaluation metric for their proposed edge-cloud collaboration network.

Finally, another behavior we evaluated is how the warning and the critical CPU thresholds impact the response time for processing critical and non-critical vital signs. These evaluations helped us discover if a specific threshold best fits most scenarios. Only critical vital signs should be executed on the current fog node when the percentage of used CPU is above the warning threshold and below the critical threshold. In contrast, non-critical vital signs should be offloaded to the parent fog node. All vital signs must be forwarded to the parent fog node when the used CPU is above the critical threshold. This evaluation will also help us to find out if there are threshold values that provide adequate results for all situations. Otherwise, we

conclude that an algorithm that automatically calculates these threshold values will be effective and should be proposed as future work.

5.6 Vital signs generation

Vital signs are artificially generated by virtual machines representing people in the neighborhood, as experiments are made in controlled environments without people effectively wearing smartwatches. We made all experiments having real-world use cases in mind, so vital signs are generated in a way that would make sense for smart cities. For example, generating a constant amount of vital signs would typically not represent a real-world scenario because people constantly move between neighborhoods. The number of vital signs sent to the fog node of a given neighborhood will hardly be the same in a given period. Following this rationale, generating vital signs in the form of a normalized wave or in the form of a constant inclination would typically not represent real life. Scenarios like these were not considered in the experiments. Several factors contribute to the number of vital signs sent to the fog nodes. A less busy neighborhood tends to have a smaller amount of vital signs, and central neighborhoods tend to have more variation in vital signs because people come and go frequently. Seasonality may also impact: people may often visit central areas during the week or go to social events on the weekends and concentrate vital signs in a specific place. We searched for a dataset with vital signs generated during a given time interval in a smart city and its periodicity, but we could not find it. With this in mind, we conducted different experiments with different numbers of vital signs to understand how the system behaves when receiving specific workloads.

5.7 Evaluation scenarios

We guided our experiments with real-world scenarios that may exist in smart cities. Some experiments have a single fog node directly connected to the cloud and represent a neighborhood in a small city. Others, in turn, have interconnected fog nodes and represent a larger city with a higher concentration of people. In the latter scenario, fog nodes that are close to getting overloaded forward vital signs to a centralized fog node in an attempt to process vital signs locally and reduce the offloading operations to the cloud, as the latter incurs extra latency. We consider a fictional social event for the experiments of this thesis since a social event encompasses different and important behaviors we are interested in evaluating: *i)* subtle increase in vital signs at the beginning of the event, while people are arriving, *ii)* small variations in vital signs, representing people leaving the event, arriving in the meantime, or simply moving through different neighborhoods, and *iii)* subtle decrease in the number of received vital signs in a given fog node as a result of people leaving the social event when it comes to an end. Another real-world scenario that could fit the desired behavior is an industrial neighborhood. The number of vital signs increases in the early morning because several people wearing smartwatches

go to this industrial neighborhood to work. There are oscillations in the number of vital signs during the day because, in practice, the number will not be constant: some people may remove the smartwatch, enter the neighborhood, or leave the neighborhood, which causes a fluctuation in the number of vital signs that are generated and sent to fog nodes. At the end of the day, people leave the neighborhood to return home, and the amount of generated vital signs sent to the fog node in this neighborhood is also reduced.

In addition to the scenarios mentioned above, we also stress the architectures with different workloads and priorities to understand how the architectures work with unpredictable workloads. Given the uncertainty of people moving through neighborhoods and also because of external factors, the architecture should be able to deal with sudden spikes in vital signs. Table 8 summarizes the scenarios evaluated in this thesis which are detailed in the following paragraphs, and gives details about how vital signs are generated for each experiment. In summary, we combine vital signs of different priorities and health services of different priorities running concurrently. These health services have different durations in seconds. This helps us evaluate the architecture's throughput, especially when vital signs have the same priority and the duration heuristic is triggered. In the latter situation, the proposed algorithms forward to the parent fog node the execution of health services that take longer to complete, so health services that execute quickly are not harmed by higher response times resulting from offloading operations.

Table 8: Scenarios that will be evaluated along with the workload, the number of threads sending vital signs simultaneously, the number of health services considered in the experiment, and whether the experiment considers fog computing or not. P1, P2, P3, P4, and P5 stands for *very healthy*, *healthy*, *warning*, *critical*, and *very critical* priorities, respectively.

Test	**Vital signs per user priority**					**Threads**					**Services**	**Fog?**
	P1	**P2**	**P3**	**P4**	**P5**	**P1**	**P2**	**P3**	**P4**	**P5**		
#1	300	300	300	300	300	1	1	1	1	1	1	✓
#2	900	900	900	900	900	3	3	3	3	3	1	✓
#3	16.000	16.000	16.000	16.000	16.000	16	16	16	16	16	2	-
#4	8.000	8.000	8.000	8.000	8.000	8	8	8	8	8	1	✓
#5	16.000	16.000	16.000	16.000	16.000	16	16	16	16	16	2	✓

Table 9 complements the details of the experiments and indicates the parameters and thresholds employed in each test. Another difference when comparing Table 8 with 9 is that Table 9 considers the thresholds for specific variations of the test cases. Each test cases consider a specific number of vital signs, but the variations consider different parameters for this number of vital signs. For example, the variations for experiment #1 aim to understand how different approaches for CPU collection behave and impact the decision-making process. As a result, the *aging* technique is the most promising and has the most stable behavior, so this technique is chosen for variations of different tests. Regarding tests #2, #3, and #4, there is no variation in the experiments, and a single combination of thresholds is used. We emphasize that tests #1 and #2 consider the same thresholds for warning and critical CPU. In experiment #3, there

is no threshold or CPU collection information because this scenario only considers the cloud, which has no module to analyze the available CPU percentage. This experiment is also the only one that uses architecture B, which has no fog node involved in the decision-making process. Regarding experiment #5, two different thresholds were evaluated because we were willing to understand their impact on the results.

Table 9: Thresholds and parameters for each variation of test case.

Test	Variation	Architecture	CPU Thresholds		CPU Collection	
			Warning	Critical	Approach	Interval
#1	#1	A	75	90	Naive	5s
#1	#2	A	75	90	Naive	1s
#1	#3	A	75	90	Multithread	5s
#1	#4	A	75	90	Aging	1s
#2	-	A	75	90	Aging	1s
#3	-	B	-	-	-	-
#4	-	C	60	98	Aging	1s
#5	#1	C	60	98	Aging	1s
#5	#2	C	80	98	Aging	1s

Scenario 1 - Low load - Few vital signs to fog and cloud with Architecture A: this first experiment involves a small neighborhood sending a few vital signs to the fog node. In other words, it is a neighborhood with a small concentration of people. The vital signs generated for this experiment are the same for every user priority, 300 vital signs for each. In total, five threads send these vital signs simultaneously, and each thread is responsible for sending vital signs with a different priority. This experiment happens in the context of architecture *A*, which has no queue on the cloud to process vital signs asynchronously. All vital signs are processed synchronously by spawning replicas of serverless functions on the cloud whenever a vital sign arises on this layer. If the fog node needs to offload the processing to the cloud, when it is close to getting overloaded, it synchronously spawns a lambda function on the cloud and waits for its response. At first glance, we were unsure of the most appropriate approach to collect the CPU because these were the first experiments made with the architecture. We emphasize that all variations of this test consider the same number of vital signs and the same number of threads to send vital signs, and all of them consider the architecture *A*. The first variation uses the most naive approach explained in Subsection 5.4, where each CPU observation is collected in the background using a time interval of 5 seconds. This means that it will take 5 seconds to get a new CPU usage value, and the modules will slowly detect this change in CPU. The second variation is similar to the first but considers a smaller time interval for collecting the CPU in the background. This means that the experiment will react faster to changes in the usage of the CPU caused by the offloading operations. The third variation considers the multithreaded method for CPU collection, also presented in Subsection 5.4 with five threads and a time shift of 1 second between starting each thread. Therefore, each thread considers the CPU of the past

5 seconds but there is a new observation available every second as it has five threads. Finally, the fourth variation encompasses the aging technique to smooth previous observations of CPU.

Scenario 2 - Medium load - Medium vital signs to fog and cloud with Architecture A: this second experiment is also made having a small neighborhood in mind, but this time it considers a higher number of vital signs generated simultaneously. The main goal is to understand how architecture A deals with the increase in vital signs, or if it will fail and require a different architecture to process this number of vital signs. This is still not a huge amount in vital signs, though. In summary, each user priority generates three times more (900) vital signs for each user priority, and three threads send vital signs for each user priority. This experiment also considers a single health service, which is responsible for monitoring the body temperature (*body-temperature-monitor*). This experiment is done with *aging* technique for collecting CPU observations, a warning threshold of 75%, and the critical threshold of 90%.

Scenario 3 - High load - Many vital signs only to the cloud with Architecture B: this experiment encompasses a more significant number of vital signs by emulating a smart city with hundreds of smartwatches generating vital signs. This approach is interesting because it does not have fog nodes to process vital signs locally, so we can further compare the results with an architecture that employs fog nodes to understand how better the latter is. This is the only experiment that uses architecture B and sends vital signs directly to a queue in the cloud. This experiment consists of 16.000 vital signs for each user priority and 16 threads for each priority. In other words, the experiment encompasses 80.000 vital signs and 80 threads sending vital signs simultaneously, so each thread sends 1.000 vital signs. This experiment considers two health services: *body-temperature-monitor* and *heart-failure-predictor*, while half of the vital signs are processed by the first service and the other half is processed by the second service. According to Cassel et al. (2022), from the analyzed works that use serverless computing as the main technology in the Internet of Things, 57% of them only execute serverless functions on the cloud and not on the fog. Although not all of them are specifically focused on the health field and collect the most different types of IoT data, the cloud is dominant when processing IoT data with serverless functions. We are interested in comparing how different architectures work and how better our proposed solution is by comparing vital signs ingestion using only serverless functions on the cloud and comparing the same workload using both fog and cloud with offloading heuristics. Unlike architecture *A*, the vital sign is asynchronously processed when it arrives on the *Cloud API Adapter* module. On architecture *B*, this module adds the message to the end of the queue. The *ingestion function* is automatically triggered when messages arrive on the queue and enough function replicas are available to consume these messages. This serverless function serves as an entry point for the ingestion of the vital sign and identifies the name of the health service from the content of the message. This lambda will then trigger, synchronously, another lambda function that has been specifically written to represent that health service.

Scenario 4 - High load - Many vital signs to fog and cloud with a single service and Architecture C: this experiment uses the architecture *C*, consisting of 8.000 vital signs for each

user priority and 8 threads for each priority, which means there are 8 vital signs simultaneously sent for each user priority. Since there are five user priorities, this experiment results in a total of 40.000 vital signs and 40 threads sending data simultaneously to fog nodes. In addition, since there are two fog nodes on the lower layer of the hierarchy, each fog node will receive 20.000 vital signs. This experiment was made with four edge nodes, in a way that each one sends 10.000 each and therefore matches the total number of 40.000 vital signs. Also, we configured the following thresholds which directly impact the offloading decisions: *i)* 1 second between each CPU collection sample with the aging technique, *ii)* 1 thread for CPU collection, *iii)* 60% of the CPU as the warning threshold, and *iv)* 98% of the CPU as the critical threshold. Only the health service *heart-failure-predictor* is considered in this experiment because the main goal is to understand how the *ranking heuristic* behaves. This service was chosen in favor of *body-temperature-monitor* because the former is a more heavyweight service and stresses the CPU on the architecture. The combination of different priorities and a single health service makes all offloading decisions either by exceeding the critical CPU threshold or by activating the ranking heuristics, but never because of the duration heuristic. The next experiment will evaluate the duration heuristic.

Scenario 5 - High load - Many vital signs to fog and cloud with two services and Architecture C: this is the last experiment for this thesis, which also uses the architecture *C*. This experiment is also situated in the context of a social event happening in two city neighborhoods and to some extent is similar to the experiment presented in Scenario 4. The main difference between them is that the current experiment considers two health services (*body-temperature-monitor* and *heart-failure-predictor*), instead of only considering the *heart-failure-predictor*. We also doubled the number of vital signs because we now have two health services to consume vital signs. We emphasize that we include the service name on the payload of the messages sent to *service-executor*, instead of leaving this optional field empty. Therefore, since we now have two health services, we also need to consider the double amount of vital signs. A total of 16.000 vital signs are sent for each user priority, and 16 threads send data simultaneously, which results in a total of 80.000 vital signs among all user priorities. This will result in an interesting behavior of ranking collision, where multiple vital signs are being processed with the same calculated ranking. Therefore, the ranking heuristics sometimes cannot determine the offloading operation because the intermediate ranking may be the same as the ranking for the vital sign being ingested. On this situation, the algorithm that will decide the offloading operation will be the duration heuristic, which offloads vital signs depending on how long the health service takes to complete.

5.8 Partial considerations

This chapter presented details about the methodology to evaluate SmartVSO, including the evaluated metrics, the infrastructure overview, and cost analysis. We also presented a testbed

to check if this computational model works as expected and effectively reduces response time when dealing with critical vital signs. We emulate the environment of a smart city by using virtual machines in São Paulo, while each virtual machine represents a specific fog node. The tests mentioned above will stress SmartVSO with different workloads so that we can evaluate this model's effectiveness. This chapter also explains that data is artificially produced to emulate the workload of a smart city, as we do not have access to a real dataset generated by vital signs from people in smart cities. This workload includes vital signs of people with different priorities, representing healthy people, people with comorbidities, or people of different ages.

We also indicate that experiments focus on understanding how SmartVSO behaves with services of different priorities, so we can see how effective the combination of both user and service priorities is in practice. Finally, we expect the results achieved with these tests will be effective and help improve the quality of experience for smart cities. Once our goal is to minimize response time and maximize throughput, these are the main metrics monitored during experiments. We also monitor the CPU level during the tests to identify how the CPU is used according to the number of incoming vital signs and the number of offloading operations. This information is important because it helped us to calibrate the warning and the critical thresholds previously presented in Subsection 5.7. Therefore, these experiments will help us discover the appropriate values for these thresholds and conclude if we can use specific thresholds that are appropriate for all situations. In case static thresholds are inappropriate, we can suggest algorithms that automatically calibrate these values in future work.

6 EVALUATION

This chapter presents the results collected after running the experiments with the developed prototype. Several architectures and charts will be presented throughout the chapter and were analyzed to indicate situations where algorithms perform well and situations where they can be improved as future work. First, experiments are made with a naive architecture without asynchronous processing. Second, experiments are made with asynchronous processing in the cloud, without using fog nodes. Finally, experiments are made with the latest version of the architecture, combining fog and cloud into an architecture with synchronous processing on the fog and asynchronous processing on the cloud.

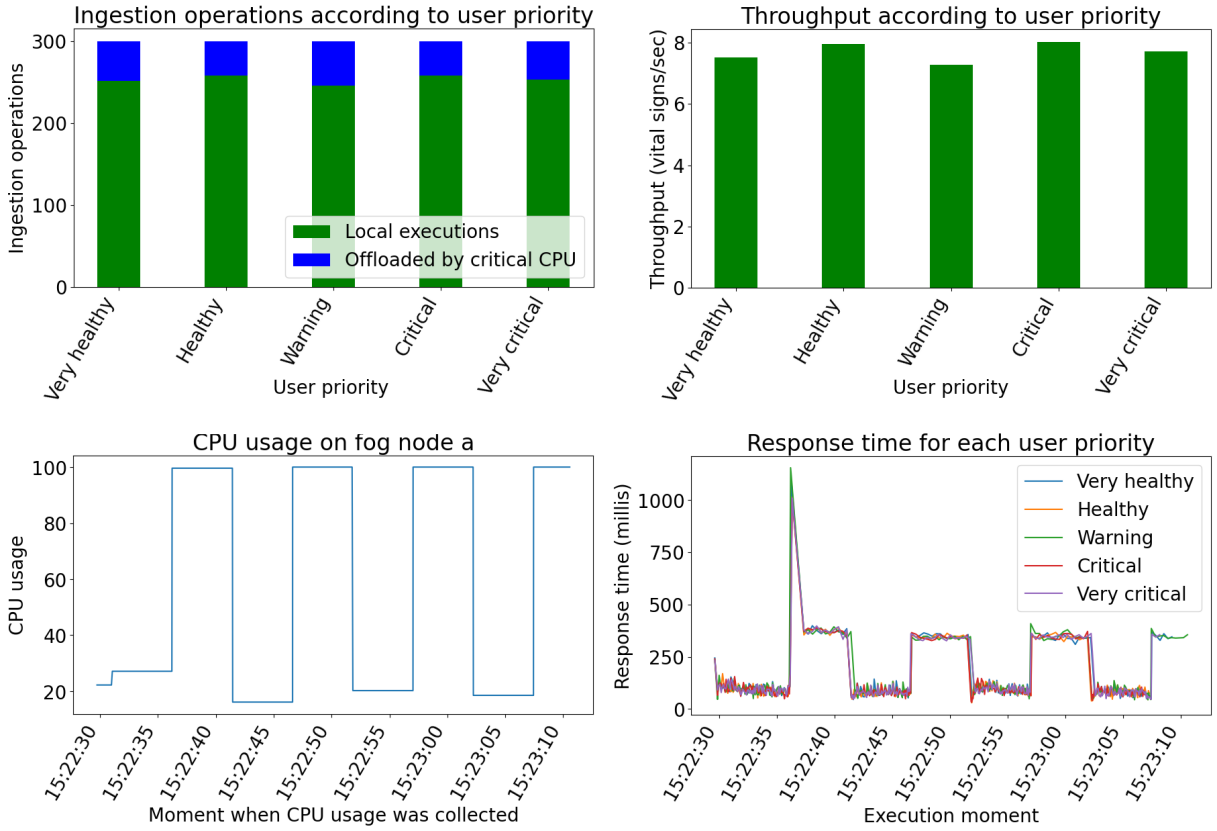
6.1 Scenario 1 - Low load - Few vital signs to fog and cloud with Architecture A

This section presents the results of the first scenario as detailed in Section 5.7, which only considers a few vital signs being sent from an edge node to the fog node. Also, the incoming variations of this experiment are achieved with architecture *A*, which was also previously presented in Section 5.2. When the fog node is overloaded, it offloads incoming vital signs to the cloud and directly invokes the serverless function, without any asynchronous strategies. There will be presented 4 variations for this experiment, using the same number of vital signs and the same offloading thresholds, with the main goal of understanding which CPU collection approach works best. These approaches were previously presented in Section 5.4. All variations of this experiment ingest a total of 1.500 vital signs and use 5 threads to send vital signs simultaneously, in a way that each thread represents a different user priority ranging from 1 to 5. Therefore, there will be 300 vital signs for each user priority. Only the *body-temperature-monitor* health service was used in this experiment, which is responsible for monitoring the body temperature and sending alerts if the person has a fever or hypothermia. Results indicate that the *aging* technique has the most stable behavior, as will be indicated with Variation #4, but incoming paragraphs will detail each result in particular.

Variation #1: this first variation considers a naive strategy for CPU collection, which has a single thread collecting CPU in the background without multithreading or smoothing technique, and considering 5 seconds as the interval to collect the observation. Also, a warning threshold of 75% and a critical threshold of 90% are used because were originally suggested for this thesis. This means that offloading heuristics should only be triggered when the CPU usage is between 75% and 90%, with the main goal of offloading non-critical vital signs to the cloud and processing high-priority vital signs locally. Also, when the CPU usage exceeds the critical threshold of 90%, all incoming vital signs are offloaded to the cloud regardless of the user priority and without triggering any offloading heuristic.

Figure 23 presents a compilation of results for this experiment variation. We notice a repeating and undesired behavior when collecting CPU observations with a single thread and an

Figure 23: Results for variation #1 of the first scenario. CPU observations are collected every 5 seconds. The ranking heuristic ends up never being triggered and all user priorities have a similar throughput, where vital signs are all either processed locally or offloaded to the cloud. Algorithms understand that CPU usage is either much overloaded or underloaded.



Source: prepared by the author.

interval of 5 seconds between each collection. This means that the thread running in the background in the *cpu provider* module sleeps for 5 seconds and considers how much CPU has been used during these 5 seconds, and this information represents the percentage of used CPU that our offloading algorithm considers. When the experiment starts, the algorithm correctly understands that CPU is close to zero because there was no vital sign being ingested yet. Since the *cpu provider* takes 5 seconds to give an updated CPU observation, all vital signs are processed locally during the next 5 seconds, regardless of the user's priority and considering the same CPU observation. No offloading heuristic is triggered in the meantime because it is only triggered for this experiment when the last CPU observation is between 75% and 90%. The algorithm will have another CPU observation after 5 seconds. However, the latest CPU observation is greater than the critical threshold of 90% because all vital signs have been processed locally during the last 5 seconds. During the next interval of 5 seconds, all vital signs will be offloaded to the cloud without triggering any offloading heuristic, while the *cpu provider* module keeps sleeping for the next 5 seconds until another CPU observation is available. This time, the CPU

observation will represent low usage because no vital sign has been processed locally in the last 5 seconds. This behavior repeats until all 1.500 vital signs have been ingested. From the perspective of the offloading algorithm, the behavior is correct because offloading operations are only performed when the CPU is busy and above the critical threshold, or are executed locally otherwise. However, the problem is that CPU observations are not being detected in a timely manner and the system makes decisions based on outdated CPU values, therefore making the system behave inappropriately and never triggering the ranking heuristic. The most appropriate behavior would be to detect smaller fluctuations in CPU usage with a shorter time interval to make decisions fast.

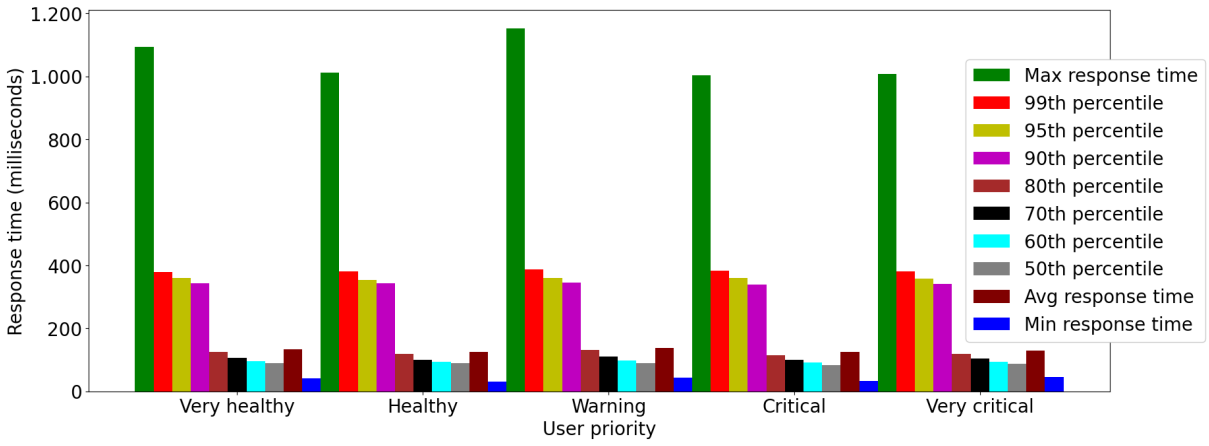
Figure 23 also indicates the throughput regarding vital signs of each priority processed per second. Given the aforementioned behavior of the CPU usage, throughput is not good because vital signs were all either processed locally or offloaded to the cloud, without triggering the ranking heuristic, which causes the system to not favor high-priority vital signs regarding people that require more attention. We highlight that heuristics are only triggered when the CPU load is between the warning and critical thresholds, but heuristics were not triggered because the CPU load either exceeded the critical threshold or was below the warning threshold. This explains why all vital signs had a similar throughput, which concludes that an interval of 5 seconds without the aging technique is not appropriate. Our expectation was that higher priorities would result in higher throughput because they would run locally most of the time, while lower priorities would result in lower throughput because of the extra latency imposed by the offloading operations from São Paulo to London, but this did not happen. In its turn, this figure also indicates the numbers of how many vital signs were executed locally and how many offloading operations happened for each user priority. A similar distribution of offloading operations can be seen for each user priority, where the number of local executions on the fog node is around 250 for each user priority, and 50 offloading operations happened for each user priority.

In addition, Figure 23 presents a chart with the response time for each user priority, which indicates that offloading operations are happening simultaneously for all vital signs received in a given period regardless of user priorities. Higher response times represent offloading operations because of the extra network latency. The serverless functions we use to run health services in the cloud are located in London, while the fog node is situated in São Paulo. We see a subtle increase in response time when performing offloading operations. The response time for processing health services on the fog usually takes around 150 milliseconds, while execution of health services in the cloud takes around 400 milliseconds. This shows the importance of processing vital signs physically close to the user since there is no extra network latency, in a way that users will receive results from the health services faster. We emphasize that this experiment considers response time as a result of synchronous invocations, which considers: *i)* the network latency to send requests from the edge to the fog and to the cloud, *ii)* the duration that the health service takes to process the vital sign, and *iii)* the network latency for the response to navigate back to the edge. In summary, it is a combination of network latency with the

duration of the health service. There is also an intriguing spike in response time in the first offloading operations that is caused by the cold start of serverless functions. AWS Lambda needs to initialize the container of the serverless function when the first few requests for this health service arise. Further offloading operations automatically reuse these same pre-warmed containers so that execution becomes faster because the container has already been initialized. Finally, the impact of these spikes in response time caused by the cold start vanished during the execution of the experiment, as they happened only once.

Figure 24 complements previous paragraphs by showing percentiles of response times for each user priority, in milliseconds. In other words, this chart presents how common it is to have vital signs processed within a given duration. We understand that the maximum response times range between 1.000 and 1.200 milliseconds because of the cold start on AWS Lambda. The maximum response time does not have a huge impact on the overall performance of the experiment, though, since up to 99% of the vital signs were processed with a response time almost three times smaller than the cold start. Up to 99% of vital signs were processed in less than 400 milliseconds, and up to 90% of the vital signs were processed with a similar response time. In turn, up to 80% of the vital signs were processed in less than 100 milliseconds. We understand from this information that higher response times represent offloading operations, while response times up to the 80% percentile represent local executions on the fog node. As a last thought, since the experiment did not favor vital signs with higher user priorities, similar results for the percentiles can be seen regardless of the user priority.

Figure 24: Response time (milliseconds) for each user priority. The maximum response times represent cold start initializations on the cloud, while the 99th and 90th percentiles represent offloading operations, and the 80th percentile and below represent processing on the fog node.

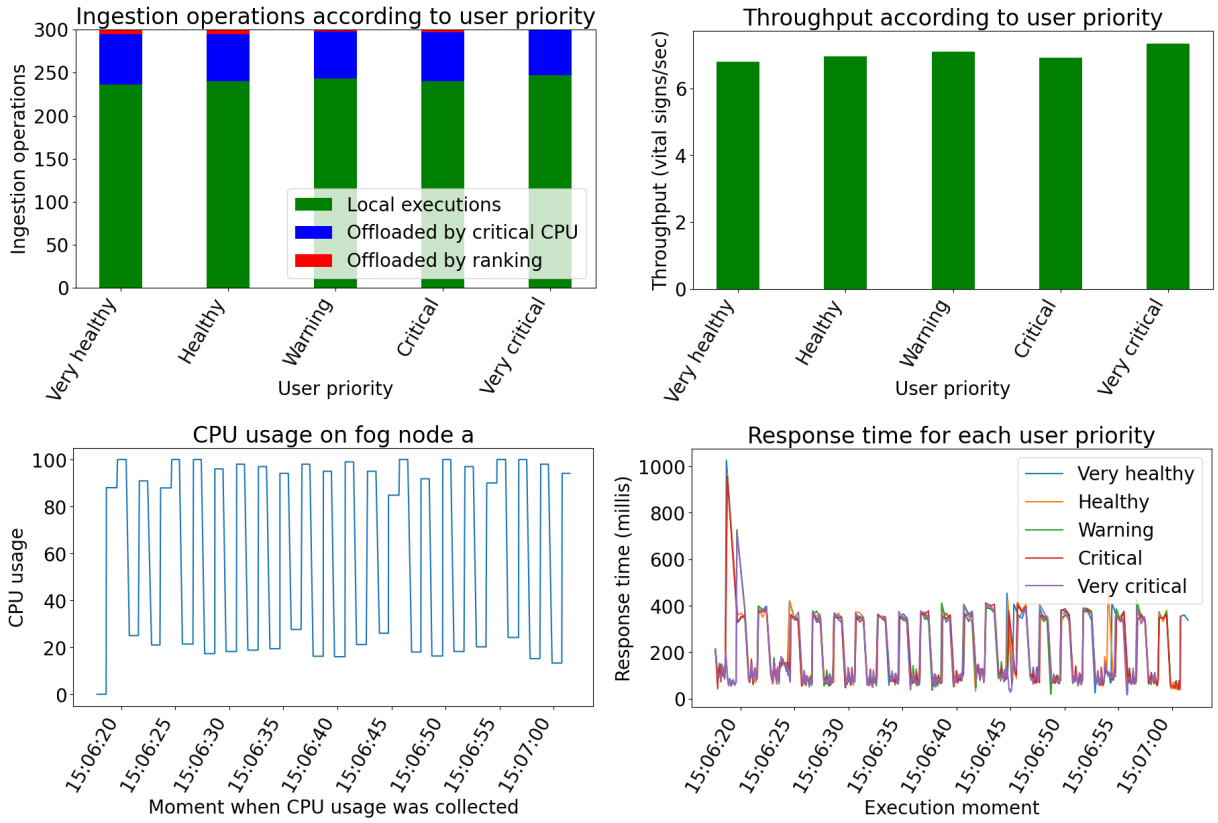


Source: prepared by the author.

Variation #2: this second variation is similar to variation #1 in the sense that the CPU load is collected using a single thread in the background, without employing any smoothing technique on the CPU observations, but this time considering a time interval of 1 second. Our goal with

this variation #2 is to reduce the time window for CPU collection, from 5 seconds to 1 second, so observations are available to the offloading algorithm sooner and decisions can be made taking more recent usage into account, therefore being more reactive to changes in processing. As a result, the CPU observations continue to range from extremely low to extremely high values, such as close to 20% and close to 100% of CPU usage. Figure 25 shows a compilation of the results and elucidates the aforementioned behavior. Regarding CPU observations, this chart has a similar format when compared to variation #1, but has a higher frequency since the CPU collection interval is smaller. We see high and low usage peaks every second because almost all vital signs were either offloaded or processed locally, except for a few situations where the CPU observation was between the warning and critical thresholds and could trigger the heuristic. This makes it clear that simply reducing the time interval is still not the best approach for CPU detection with our algorithms.

Figure 25: Results for variation #2 of the first scenario. CPU observations are collected every second. The ranking heuristic ends up being eventually triggered, as the algorithm now reacts faster to processing efforts. User priorities still have a similar throughput because the ranking heuristic is only invoked a few times and does not have a significant impact on the offloading.



Source: prepared by the author.

Figure 25 also shows the throughput regarding vital signs processed per second during the experiment with variation #2. This is similar to the throughput of variation #1 that was presented

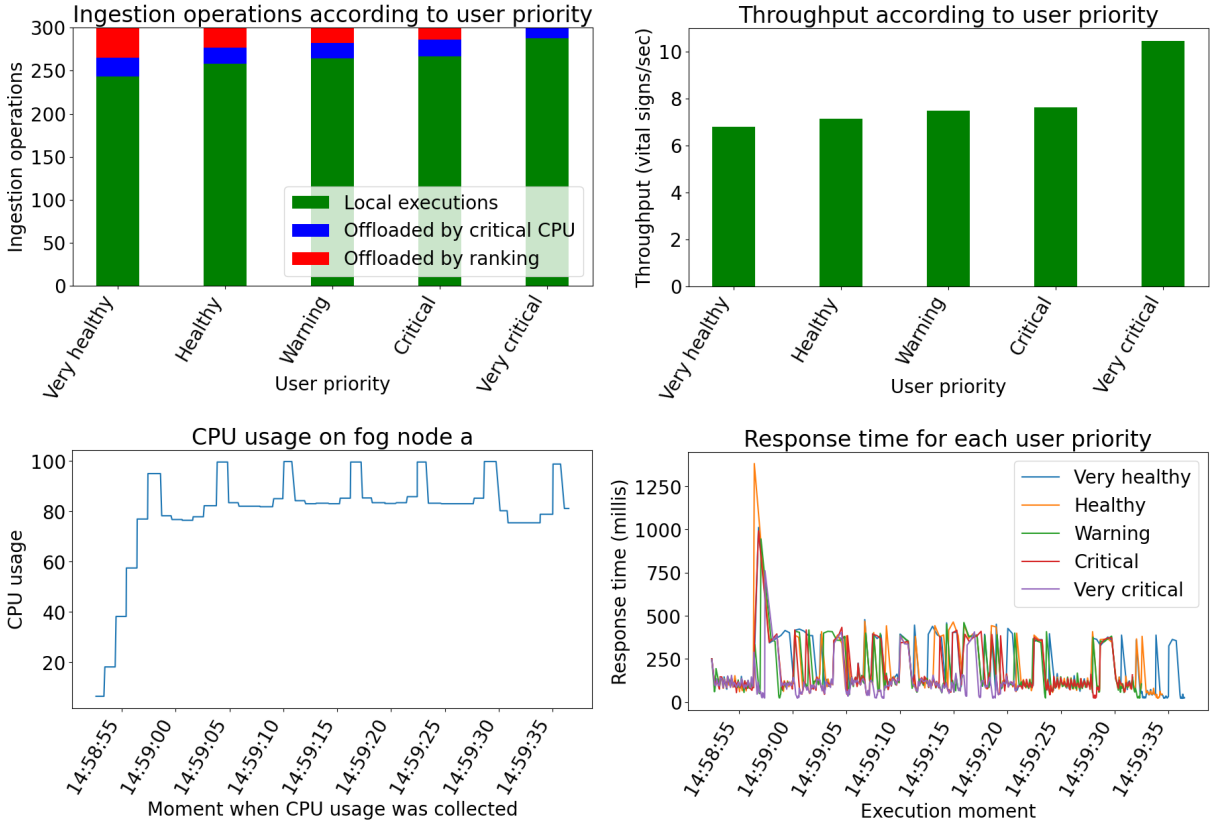
in Figure 23. The throughput is not effectively improved for vital signs with higher priorities as a direct consequence of not triggering the ranking heuristic many times. The algorithm either processes the vital sign locally or offloads it to the cloud regardless of the user priority most of the time, which is not the expected behavior. This makes the throughput similar for all user priorities. The small fluctuations in throughput among user priorities typically happen because of the nature of distributed systems, such as small variations in the network and the usage of distributed resources during the experiments. This figure also presents a chart with the response time for each user priority that complements the previous conclusions: offloading operations result in higher response times because of the extra network latency, which happens in a higher frequency than variation #1 as a direct consequence of the smaller CPU collection interval. It can be seen that vital signs are processed in an intercalated manner with offloading operations (higher response time) and local executions on fog nodes (lower response time). The first offloading operations have a higher response time because of the cold start in the serverless platform, with serverless functions being initialized on the cloud provider, while additional requests reuse the same container. As a conclusion for this variation of the experiment, we still need a different strategy to collect CPU more smoothly in order to reduce subtle peaks on the presented charts, in a way that heuristics can be appropriately triggered with CPU observations between the warning and the critical CPU thresholds.

Variation #3: this third variation has a substantial difference in the approach for collecting CPU observations. In this specific variation, CPU measurements consider a time window of 5 seconds in a similar manner employed for variation #1, but with a substantial difference of having 5 threads responsible for collecting the CPU in the background instead of a single thread. Each thread is started with a time shift of 1 second after the other. This is an attempt to combine approaches of variations #1 and #2 in a single strategy: we still consider a longer duration but we have an updated value each second because of the time shift for starting each thread. This approach indicates better results when compared to the previous variations of the experiment. Figure 26 shows a compilation of the results for variation #3, including the the CPU observations during this experiment, the response time, and how vital signs were ingested with offloding operations. We can see a new CPU observation each second, but considering the usage of the last 5 seconds. An interesting behavior to be noticed in this figure is the repeated iterations of spikes in the CPU observations, which are usually close to 80%, but suddenly reach a value close to 100%. The generation of vital signs is the same as the previous variations of this experiment, which leads us to conclude that the difference in the chart with CPU observations is directly associated to the CPU collection strategy employed for this variation. Our rationale for this behavior is the following: given the warning threshold of 75% and the critical threshold of 90%, the ranking heuristic is triggered most of the time, as the chart indicates that CPU usage mantains close to 80%. During 5 seconds the CPU was usually triggering the heuristics in a way that most vital signs were executed locally, but some of them were also offloaded. After the first thread waits for 5 seconds, it understands most vital signs were processed locally during the last

5 seconds and will assume a CPU observation close to 100%. During the next second, all vital signs are offloaded to the cloud because other threads are still sleeping and waiting to complete 5 seconds. This single second of where all vital signs are offloaded contributes to other threads understanding a reduced CPU usage on the local fog node. Differently from variations #1 and #2, it is not drastically reduced to 20% because the collection takes into account the other 4 seconds of processing vital signs locally. The process repeats and another thread identifies most vital signs being processed locally during the next 5 seconds, and the iteration repeats until all vital signs are consumed.

this should

Figure 26: Results for variation #3 of the first scenario. CPU observations are collected with 5 threads considering a time interval of 5 seconds each, but each thread is initiated with a time shift of 1 second. This leads to better results than previous variations, but there is still room for improvements because the CPU usage chart still has undesired observation spikes.



Source: prepared by the author.

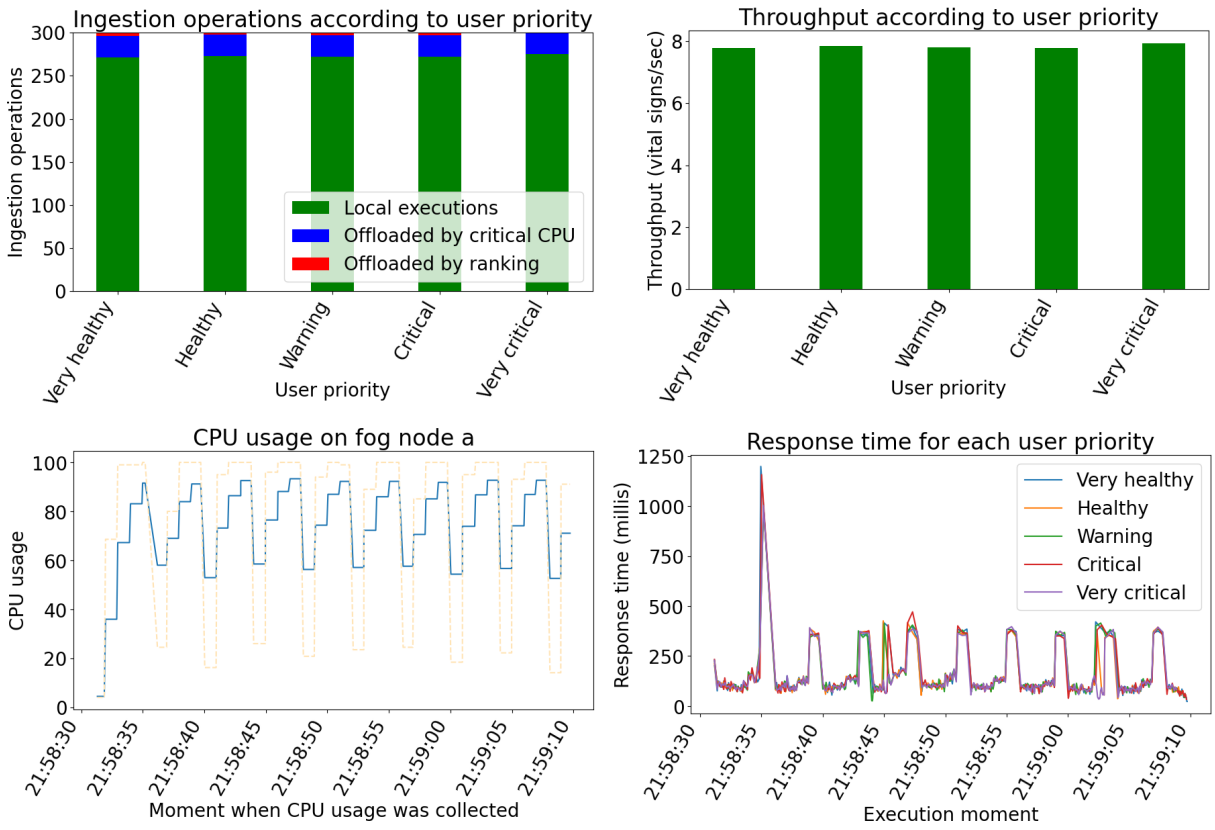
Figure 26 also indicates a more appropriate behavior regarding the throughput of vital signs processed per second, when compared to variations #1 and #2. This is a direct consequence of this specific CPU collection strategy. Vital signs with higher priorities result in improved throughput because most of them are processed locally, therefore not being offloaded to the cloud. On the other hand, a reduced throughput for vital signs with lower priorities can be seen because they are typically offloaded to the cloud, which is expected, as cloud employs

additional network latency. We can also see most offloading operations being made especially for vital signs regarding very healthy users, due to the effectiveness of the ranking heuristic, as the CPU collection usually ranges between the warning and critical thresholds. In contrast, part of the offloading operations was made due to exceeding the critical CPU threshold. Vital signs regarding users in very critical conditions were only offloaded when exceeding the critical threshold, as these vital signs should never be offloaded because of the ranking heuristic when being processed with a single health service. In summary, the lower the priority, the greater the number of vital signs being offloaded because of the heuristics. This chart also elucidates that most vital signs were executed locally on the fog node, regardless of the priority. As a final thought, variation #3 presents a better behavior when compared to previous experiments but it still does not seem appropriate, as the CPU collection still has subtle usage spikes that are not desired. We still need an approach that collects CPU in a more smooth manner.

Variation #4: finally, the fourth variation of the experiment uses the smoothing technique to achieve more stable values regarding CPU usage (RIGHI et al., 2016). Offloading decisions are not made by considering only the last CPU observation but the previous six observations instead. To the last observation is assigned the highest weight, which decreases exponentially, so recent observations have more impact on the final CPU usage percentage. In other words, the offloading decisions are based on a CPU usage resulting from multiple observations. Figure 27 shows a compilation of charts with results for variation #4, including a chart with CPU observations expressed as two lines with different colors. The yellow one represents the non-smoothed CPU usage, which is analogous to the traditional approach indicated in the variation #1 of this experiment, and the blue line represents the smoothed CPU calculated with the aging technique, considering the last 6 observations. Offloading operations do not cause a drastic impact on the observed CPU anymore, which means the decisions are smoother.

This same compilation of charts presented in Figure 27 also indicates the throughput for variation #4. We see that the number of vital signs processed per second is quite similar between priorities, which is still not the best-expected behavior. We understand that the number of vital signs being ingested simultaneously is too tiny and cannot be compared to a real-world use case, which leads us to redesign further experiments that will be presented in this chapter considering a larger number of vital signs ingested simultaneously. The expected behavior will be that higher priorities will result in higher throughput, while lower priorities will have the throughput reduced in favor of vital signs with critical priorities. The approach to collect CPU presented in this variation #4 looks very promising, though, as it is already used in the literature to make decisions based on CPU thresholds and makes much sense in our scenario. In turn, this figure also indicates a chart with how many ingestion operations happened according to user priority, indicating how many vital signs were processed locally on the fog layer, how many vital signs were offloaded because of exceeding the critical CPU threshold, and how many were offloaded because of the ranking heuristic. Most vital signs were executed locally regardless of the priority, contributing to a similar throughput for the experiment. More than 250 vital

Figure 27: Results for variation #4 which employs the aging technique to collect CPU usage.



Source: prepared by the author.

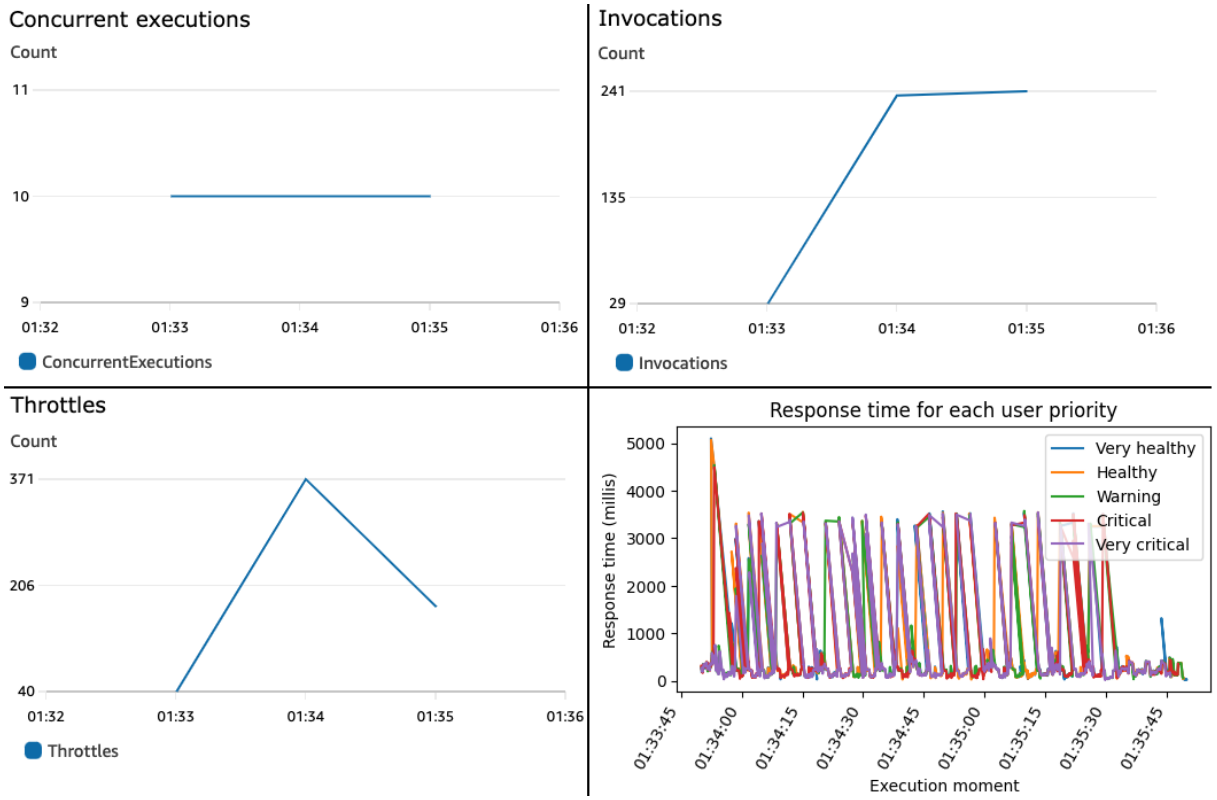
signs were processed locally for each user priority. Only a small percentage of vital signs were offloaded to the cloud, but most offloading operations were made because the smoothed CPU (considering the last 6 observations) resulted in a percentage that exceeded the critical CPU threshold. Therefore, vital signs were directly offloaded to the cloud regardless of the priority. Only a small fraction was offloaded because of the ranking heuristic. Similar to the previous variations of this experiment, this happens because the health service *body-temperature-monitor* is lightweight and executes in a scale of milliseconds, which sometimes causes a single vital sign to be processed in a given time period instead of multiple vital signs being processed simultaneously. The ranking heuristic understands that the execution should happen locally when no more than a single vital sign is being processed, so this is the cause of the low number of offloading operations triggered by the ranking heuristic. Finally, user priority 5 has only offloading operations caused by exceeding the critical CPU threshold, which is correct, since it should never be offloaded because of the ranking heuristics with a single health service.

6.2 Scenario 2 - Medium load - Medium vital signs to fog and cloud with Architecture A

The four variations evaluated and presented in Section 6.1 considered different techniques to collect CPU observations with a total number of 1.500 vital signs. This has a major limiting factor, though: only five vital signs are ingested simultaneously, one for each user priority. This number is not comparable to a large smart city, so there is a need to stress the architecture with a more significant number of vital signs. We emphasize that not only the total number of vital signs should be higher, but especially the number of vital signs being processed simultaneously. This is analogous to a higher number of people concentrated in a single neighborhood using smartwatches. With this in mind, this scenario considers a total number of 4.500 vital signs and 15 vital signs sent simultaneously, with three simultaneous vital signs regarding each user priority. This number may still not represent a large neighborhood in a smart city, but for this second experiment, it is interesting to understand how the architecture behaves with a medium load. Results are impressive and even required us to redesign part of SmartVSO model, as will be further explained. We noticed several offloaded vital signs failing to be processed on the serverless platform on the cloud because of Throttling errors in AWS Lambda, as the maximum limit of 10 concurrent Lambda executions was exceeded. This represents a potential problem for architecture *A* in a real smart city where the number of people in a given neighborhood fluctuates during the day and the week. In this experiment, 809 out of 4.500 requests failed, while 3.691 requests succeeded, considering both fog and cloud layers.

Figure 28 indicates a compilation of results for this experiment, while the first chart indicates the number of concurrent executions of *body-temperature-monitor* function on the cloud. We see a constant number of 10 replicas being executed all the time, as there was a spike in requests and the serverless platform on the cloud could not exceed this number of replicas. Under these settings, this indicates that not more than 10 vital signs can be processed simultaneously on the cloud and additional offloading operations will fail and will not be executed. In turn, the second chart indicates the invocations of serverless functions that succeeded, excluding invocations that failed because of throttling errors. There is an interesting relation between the number of throttlings and the number of invocations of serverless functions in a given time interval, as the number of throttling errors reduces when vital signs stop being offloaded to the cloud. The third chart indicates the throttling errors during the experiment, as the serverless platform cannot handle such a number of requests when several vital signs are offloaded. All vital signs are consumed synchronously in this experiment with architecture *A*, in a similar manner that was made for the first scenario presented in Section 6.1. In other words, the response indicating if the person has a health problem can be collected from the HTTP response body after the vital sign is ingested. Regarding design complexity, building an architecture that makes synchronous requests (without dealing with edge cases) is simpler than introducing an architecture that processes messages asynchronously because there is no additional component responsible for consuming requests from a queue and storing responses in a data store to be further col-

Figure 28: Metrics regarding response time and execution of serverless functions on the cloud. This figure presents a compilation of concurrent executions, invocations, response time, and throttling errors regarding the execution of serverless functions during the experiment with AWS Lambda.



Source: prepared by the author.

lected. Even though the literature indicates that serverless computing offers virtually infinite computing resources by allocating replicas of the function according to the number of requests, we discovered that it still has a limit in practice. The experiment resulted in several throttling errors from the cloud provider because functions scaled up and consumed the unreserved concurrency previously configured for the serverless functions (AWS, 2022a). Finally, the fourth chart presents the response time for the workload in this experiment. There can be noticed that executions also take longer to complete when throttling errors happen, as the HTTP connection keeps established with the cloud provider for a few seconds until the throttling error is returned. This cart indicates that throttled requests take around 3 seconds to finish. We assume that the cloud provider holds the connection for a while as an attempt to execute the request in case existing functions finish their execution. However, this is only an assumption because, as far as we know, the AWS documentation does not make this behavior explicit. As a last thought, regardless of the aforementioned throttling problem, the ranking heuristic gives a higher preference to processing urgent vital signs on the fog nodes, which leads to more non-critical vital signs being processed on the cloud. With this in mind, the conclusion for this experiment is that

we could employ a queue in the cloud so that vital signs are added to this queue and serverless function replicas consume the messages at a speed they can deal with, which does not result in vital signs being lost because of throttling problems. Also, it may not be a problem if non-critical vital signs take longer to be processed in the cloud to detriment of critical vital signs being processed locally in the fog nodes. Finally, another option would be to include additional fog nodes on the hierarchical architecture to mitigate the number of offloading operations to the cloud. These ideas lead us to propose an evolved and different architecture that will be evaluated in the incoming sections.

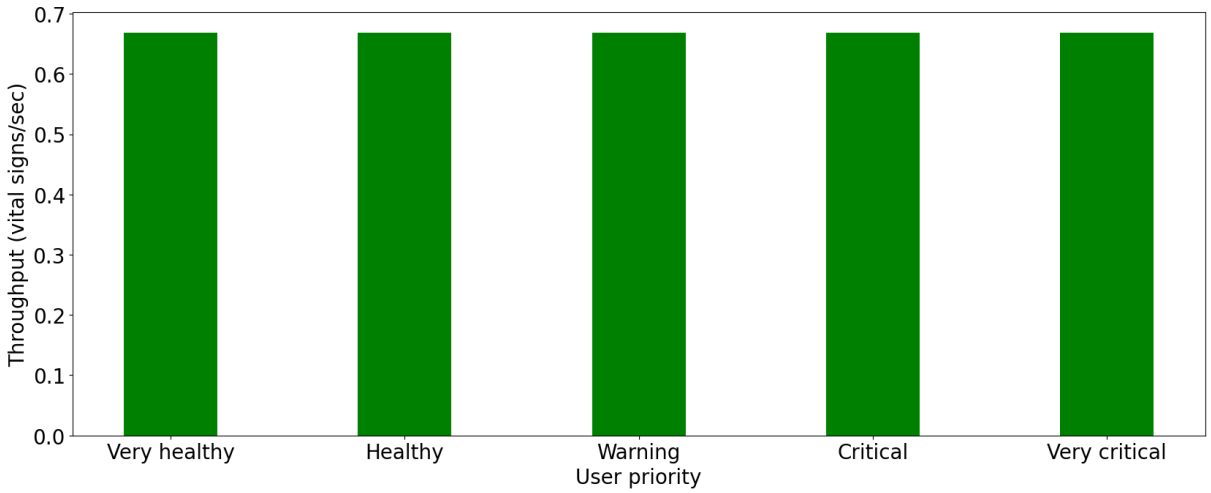
6.3 Scenario 3 - High load - Many vital signs only to the cloud with Architecture B

The previous experiment clarified that invoking serverless functions synchronously during offloading operations can result in throttling if the total offloading operations exceeds the maximum limit of concurrent replicas. A more robust architecture needed to be proposed to solve this problem. In summary, the lesson learned from previous experiments is that instead of spawning function replicas whenever a vital sign arises on the cloud, these vital signs could be added to a queue to be consumed asynchronously according to the availability of function replicas. In other words, this allows functions to consume vital signs from the queue at a speed that does not cause throttling and therefore does not cause vital signs to be lost and discarded. The current experiment uses architecture *B* and is only focused on the cloud, without taking fog nodes into account. We reiterate that details regarding architecture *B* have been presented in Section 5.2. This experiment evaluates and analyzes how asynchronous processing of vital signs behaves considering a neighborhood with a high concentration of people, with smartwatches sending vital signs directly to the cloud. Vital signs generated during this experiment will be processed asynchronously by serverless functions connected to the queue, which will be further compared with other experiments that employ asynchronous processing in the cloud, but also consider fog computing to reduce response time for high-priority vital signs. In other words, results will be compared with other experiments that employ fog computing to understand how effective they are. The current experiment considers a total of 80.000 vital signs, with 80 vital signs being artificially generated simultaneously. The expectation for this experiment is that all vital signs will result in similar response times, as the asynchronous approach employed in architecture *B* does not favor urgent user priorities and ingests vital signs as they arrive in the queue.

Figure 29 shows the throughput for each user priority after processing messages from the queue with 10 replicas for serverless functions. We see a similar throughput for each user priority, which means, there is no such difference in throughput for specific user priorities when sending vital signs directly to the cloud with a single queue. For each received message, a serverless function will be executed, which means that each serverless function will receive a single vital sign as the input. Up to 10 vital signs can be processed simultaneously because this is the concurrency limit for the configured on the AWS account, which can also be increased

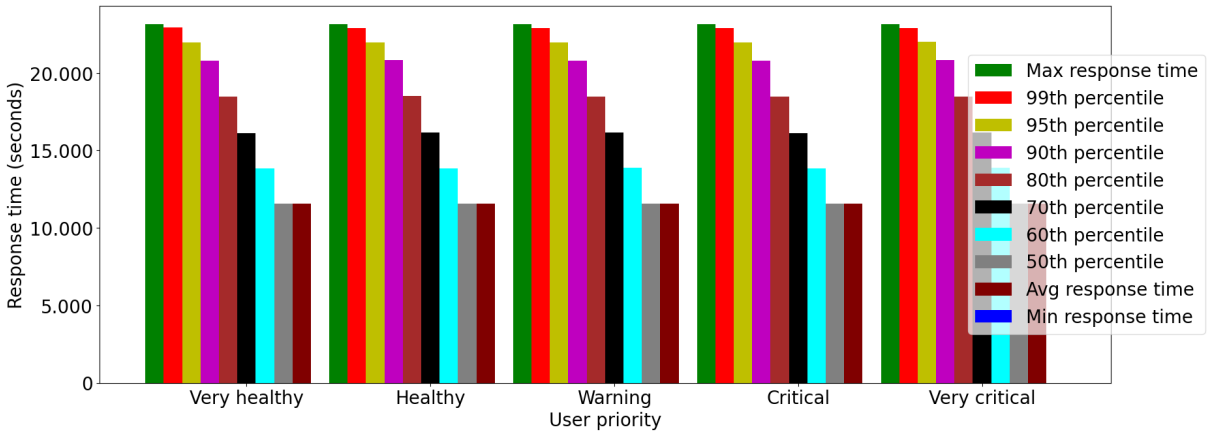
in future work. After requesting an increase in quotas with the cloud provider support, for example, more replicas of the serverless function can be used as consumers of vital signs in the queue and consume messages more quickly. This would reduce the response time for all priorities, but would still not favor critical vital signs in the cloud, as the architecture *B* has a limitation of not employing heuristics to favor urgent vital signs in the cloud.

Figure 29: Throughput for user priorities when vital signs are always processed on the cloud.



Source: prepared by the author.

Figure 30: Response time for user priority when vital signs are always processed on the cloud.



Source: prepared by the author.

Finally, Figure 30 indicates an interesting and expected behavior for this naive architecture that processes all vital signs on the cloud in an asynchronous manner, without using heuristics to favor urgent vital signs. Since the vital signs are typically consumed in the order they arrive on the queue, and the people in the neighborhood (represented on this architecture by edge nodes)

simultaneously send vital signs with different priorities directly to the cloud, all user priorities share a similar response time for all analyzed percentiles. High-priority vital signs are not favored because no heuristic is being used on the cloud. The incoming experiments presented in the following sections will clarify that leveraging heuristics with fog nodes greatly improves response time for high-priority vital signs. In summary, this experiment makes it clear that using a queue in the cloud makes it possible to handle vital signs for such a high concentration of people in smart cities. It may take longer to process a large number of pending messages, but none will be lost because of throttling.

6.4 Scenario 4 - High load - Many vital signs to fog and cloud with a single service and Architecture C

The main limitation of the previous experiment is that vital signs are only processed in the cloud without favoring vital signs with higher priorities. Also, the previous experiment does not consider fog nodes to ingest vital signs with a shorter response time because of the shorter physical distance between the smartwatch and the fog node. With this limitation in mind, the current experiment considers architecture *C* and combines synchronous processing on fog nodes with asynchronous processing in the cloud, to favor urgent vital signs and still deal with high usage spikes, respectively. In this approach, vital signs are only offloaded to a queue on the cloud by the last fog node on the hierarchy, which are asynchronously processed with multiple serverless function replicas consuming vital signs from the queue. This works as a producer-consumer model, where the last fog node on the hierarchy is the producer and the function replicas in the cloud are the consumers. This experiment is situated in a fictional social event happening in two neighborhoods of a city, which encompasses a large number of vital signs sent to two fog nodes. Each fog node would be situated in a neighborhood of a real-world smart city. The goal of this experiment was to stress the architecture with a large number of vital signs and was done with: *i)* four edge nodes, *ii)* 10 threads on each edge node sending vital signs simultaneously to the leaf fog nodes, which represents 40 vital signs being sent simultaneously because of the four edge nodes, and *iii)* a total of 10.000 vital signs sent by each edge node. The current experiment is composed of two variations, but before diving deeper into the results of each variation, we present several failed attempts and fixes that needed to be made to achieve appropriate results. It was an iterative process of running the experiment, having undesired results, understanding what happened, fixing the code, and repeating the experiment. This was repeated until the experiment could successfully complete and appropriate results could be provided. The following paragraphs give details about each attempt and the outcomes.

Execution attempt #1 - Failure: this first attempt to run the experiment did not work as expected, as some fog nodes became so overloaded that they stopped responding to HTTP requests. At first glance, we did not understand what was happening with the resources of the virtual machines because the VMs were frozen, so we could not even connect by SSH to check

what happened on the operating system. We were tempted to change the offloading thresholds and rerun the experiment to see what would happen, with the hypothesis that the machine was processing so many vital signs simultaneously and could not handle such a workload. However, we wanted to dive deeper into what really happened and draw accurate conclusions instead of making assumptions. Unfortunately, we could not collect all the required information about the virtual machine when it stopped working because it did not respond to any HTTP request, which blocked us from checking the machine resources used at this exact moment. As a result of this execution attempt, we understood that further investigation is required, which is the goal of the next attempt.

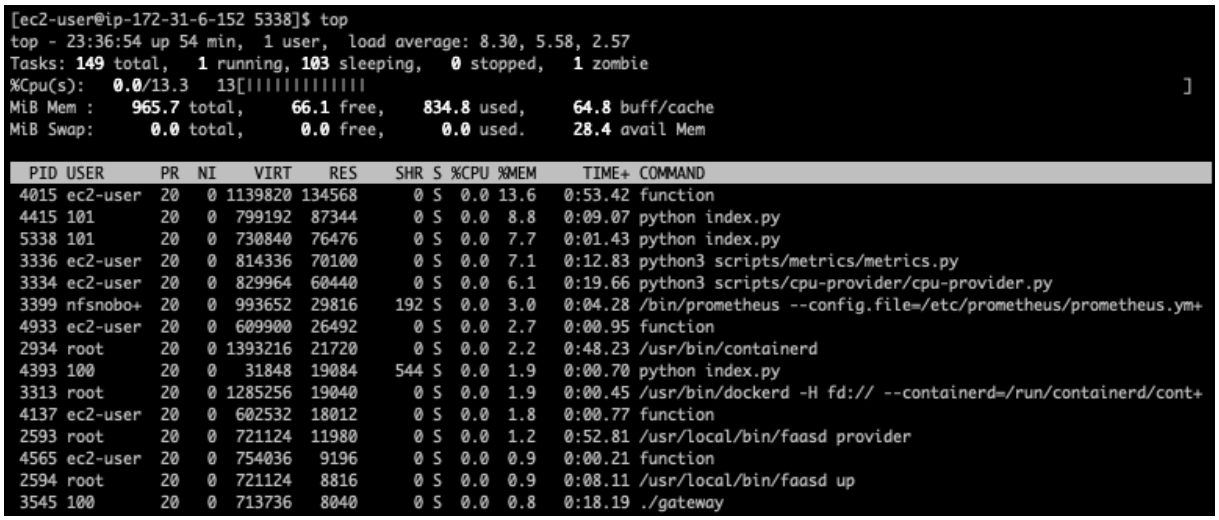
Execution attempt #2 - Failure: given the knowledge gained from the previous attempt, we executed another repetition of the experiment using the same workload and the same parameters, but this time logging into the virtual machines via SSH before they froze. This attempt aims to understand what happened with the computing resources on the fog nodes during the vital signs ingestion. We analyzed the processes running on the operating system and noticed that the problem was a lack of memory. However, it was not simply due to the large workload but to memory leaks: an incorrect software behavior where the software does not return the memory to the operating system after it finishes working with the data. This was not noticed before because the first experiments used a small number of vital signs, which was not enough to make this problem evident. We could spot three memory leaks that needed to be fixed as a requirement to complete this experiment with 40.000 vital signs. The first memory leak was found on the *service executor* module itself. The *duration offloading* heuristic requires historical duration about the last few executions of the health service, but not all of them. However, *service executor* was holding all previous durations in the memory while only the last six observations were needed. The second memory leak was in the *cpu-provider* module: a simple web server written in Python that provides information about the resources used on the machines. This web server uses a multithread approach by mixing the classes *ThreadingMixIn* with *HTTPServer* to instantiate a new thread whenever a request is received. However, due to a bug[1] present in older versions of Python, the thread was added to an internal list and was only removed when shutting down the process. The solution is presented here[2], and we needed to upgrade the Python version to 3.8 to get this fix.

Figure 31 shows the machine resources on fog node C while running the experiment when the machine started rejecting HTTP requests. To identify the processes with higher memory usage, the *top* command was used after logging into the EC2 machine. This command allows us to sort processes by different data, while the memory was selected to sort processes by pressing *SHIFT* + *M*. We see that only 28,4 megabytes of memory were available when the machine started to reject requests. The *service executor* function is the first process on the list, assigned to PID 4015 and consuming most of the memory. Since the command itself is only

[1] https://bugs.python.org/issue37193

[2] https://github.com/python/cpython/commit/0064d561b8e44f7a991188d7c5016c165bc89326

Figure 31: Processes consuming most of the virtual machine memory.



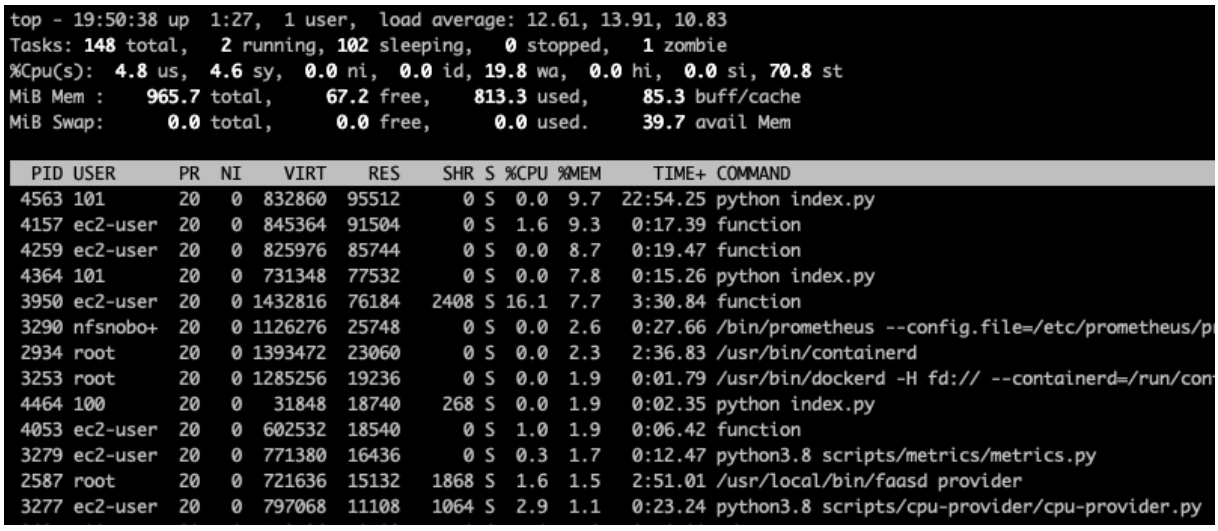
Source: prepared by the author.

named *function*, we could not directly understand which function it was. We needed to inspect information about the process with the command *cat /proc/4015/status*, which showed us the file *application.properties*. We then inspected the content of this file and noticed that it matched the file with the same name located under the *service executor* source code of the project, which made us understand that this function is related to this specific module. The other two processes with high memory usage, respectively assigned to PIDs 4415 and 5338, represent the *heart-failure-predictor* health service and the *predictor* function. These commands also do not make it explicit, so we also inspected information under the */proc/<PID>* virtual folder. This folder contains the Python files used by each function. After examining the content of the files, we could identify the serverless function to which they were related. Even though they are on the top memory usage, the memory number does not increase as the execution continues, which makes us understand that they are not causing harm to the system and do not have memory leaks. As an initial conclusion, this second attempt makes it clear that memory is an important factor when deploying and running health services on fog nodes. This certainly requires extra attention before using in production, as SmartVSO is introduced in a way that authorized third parties can implement and deploy their own health services to the serverless platforms. It is important to have tools that monitor the number of memory on the system, as well as limit the maximum number of memory that a given serverless function can use on fog nodes, as a single function can use a lot of memory and compromise the correct behavior of the fog node, thus making it unstable.

Execution attempt #3 - Failure: the source code of the *service executor* module was then changed to remove the oldest duration in the list when it exceeds the maximum number of six durations for each health service. In other words, after the health service executes, the *service*

executor will have seven durations, and the oldest one is removed to maintain a maximum total of six elements. Interestingly, only making this change did not solve the problem because the Java Virtual Machine (JVM) still used more and more memory. We needed to run a command to explicitly execute the garbage collection when finishing the execution of the *service executor* module, which was done by calling *System.gc()* at the exit of the function. Also, before running this third experiment, we updated the Python version to 3.8 to get the fix on the *cpu-provider* module. Given that problems were fixed, we ran a third experiment attempt, but it continued to fail because of a lack of memory. Specifically focusing on the *service executor* module and the *cpu-provider*, the experiment was successful because the memory leak on these processes was solved. Still, in the sense of the whole experiment, it was a failure, as the system could not ingest the 40.000 vital signs. Since we developed most modules as Java serverless functions they also suffered from the same garbage collector problem. There was no memory leak on the functions because the memory was eventually released, but the garbage collector was taking so long to clean unused memory that the operating system achieved its limit. Our tests use *t2.micro* machines with 1GB of RAM, so a function using 100 MB represents around 10% of the total memory. We are now aware that we also need to manually invoke the garbage collection at the exit of each function written in Java.

Figure 32: Additional functions still consuming a considerable number of memory.



Source: prepared by the author.

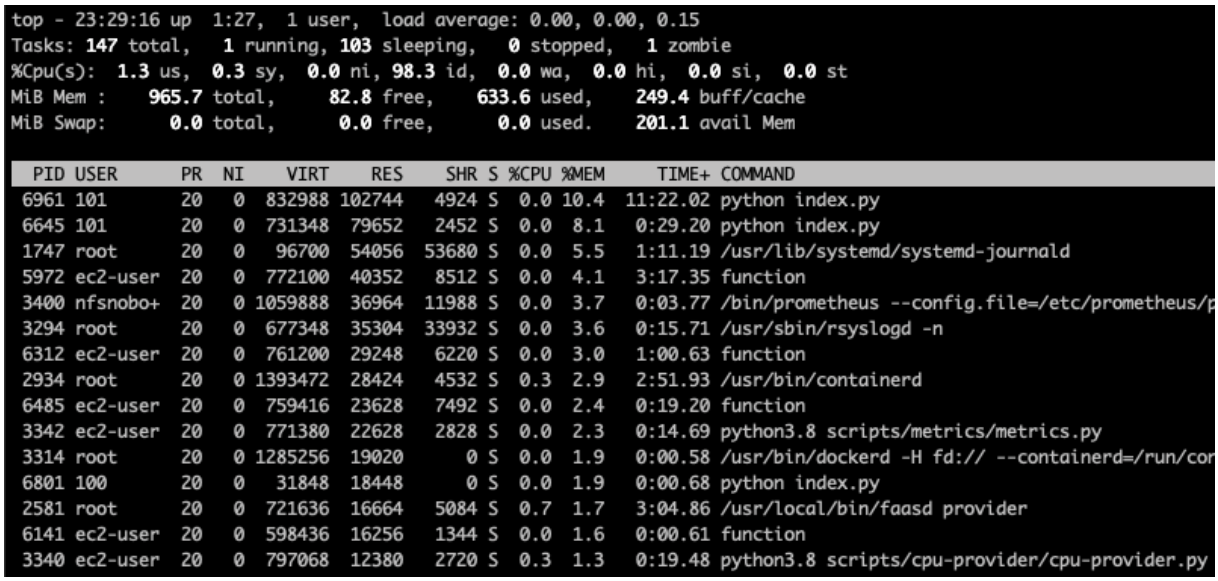
Figure 32 shows some of the processes running on the fog node, also sorted by memory usage. Processes whose command is *function* are modules compiled in Java with native compilation enabled. We have not tested functions compiled without native mode, but we assume that they would use even more memory since the native mode aims to reduce both processing and memory footprint. When comparing Figures 31 and 32, the processes named *function* were using a small number of memory (ranging between 80 and 90 MB) on the former figure. Still,

the memory increased as the system continued ingesting vital signs. One might wonder why Figure 31 does not show these functions with high memory usage. The reason is that after solving the two first memory leaks, the experiment lasted a lot longer and ingested a lot more vital signs before freezing the virtual machines again. This second problem was masked behind the previous problems and has only now become evident. From a total of 40.000 vital signs, only 22.413 of them could be processed before the machines stopped working on the current attempt. After some time, it was not even possible to log into the virtual machines using SSH since there was almost no memory available.

Execution attempt #4 - Success: we have invoked the garbage collection on all Java serverless functions and have executed the same experiment again. The execution attempt was finally successful because all 40.000 vital signs could be ingested, and fog nodes did not freeze during the experiment. Figure 33 indicates fog node C having around 200MB of available memory at the end of the experiment and shows functions using less memory than before. We see processes named *function* using way less memory, ranging from 16 to 41 MB, as opposed to the previous attempts where *function* processes ranged from 19 to 93MB. Another intriguing process on the operating system that requires attention is the one with PID 3342 and command containing *metrics.py*. This process receives from the *service executor* module detailed information about how vital signs ingestion is behaving on the fog node. This module is essential for the experiments because it represents the source of truth from where charts were plotted and presented for this thesis. This module stores the number of times each heuristic was triggered, for example, as well as how much CPU was available when vital signs were being processed and how many vital signs needed to be offloaded to the parent fog node, among others. However, this module stores all information in the memory. The longer the experiment, the more memory this process will use. This experiment (composed of 40.000 vital signs) did not make the memory reach its limit since the other memory leaks were fixed, so we did not need to change anything on the *metrics* module. When monitoring data for longer experiments or using the proposed model in production, though, these metrics should be stored on disk, preferably in an asynchronous manner, to avoid exceeding memory limits and also avoid performance issues with synchronous write operations.

Figure 34 shows a compilation of results for this successful execution attempt, such as the throughput indicating the number of vital signs processed per second for each user priority and the ingestion operations on each fog node. Regarding throughput, it indicates the number of vital signs processed per second for each user priority considering the duration between processing the first and the last vital sign with that priority. In other words, it encompasses the subtraction between the timestamp when the last vital sign was processed and the timestamp when the first vital sign was processed. This also considers the waiting time on the queue in the cloud: a single vital sign waiting for a long time in the queue represents an outlier and impacts this chart, therefore considerably reducing the throughput for that user priority. For example, let us suppose that almost all vital signs regarding users with critical priority were processed on the

Figure 33: Processes on the operating system with low memory footprint.

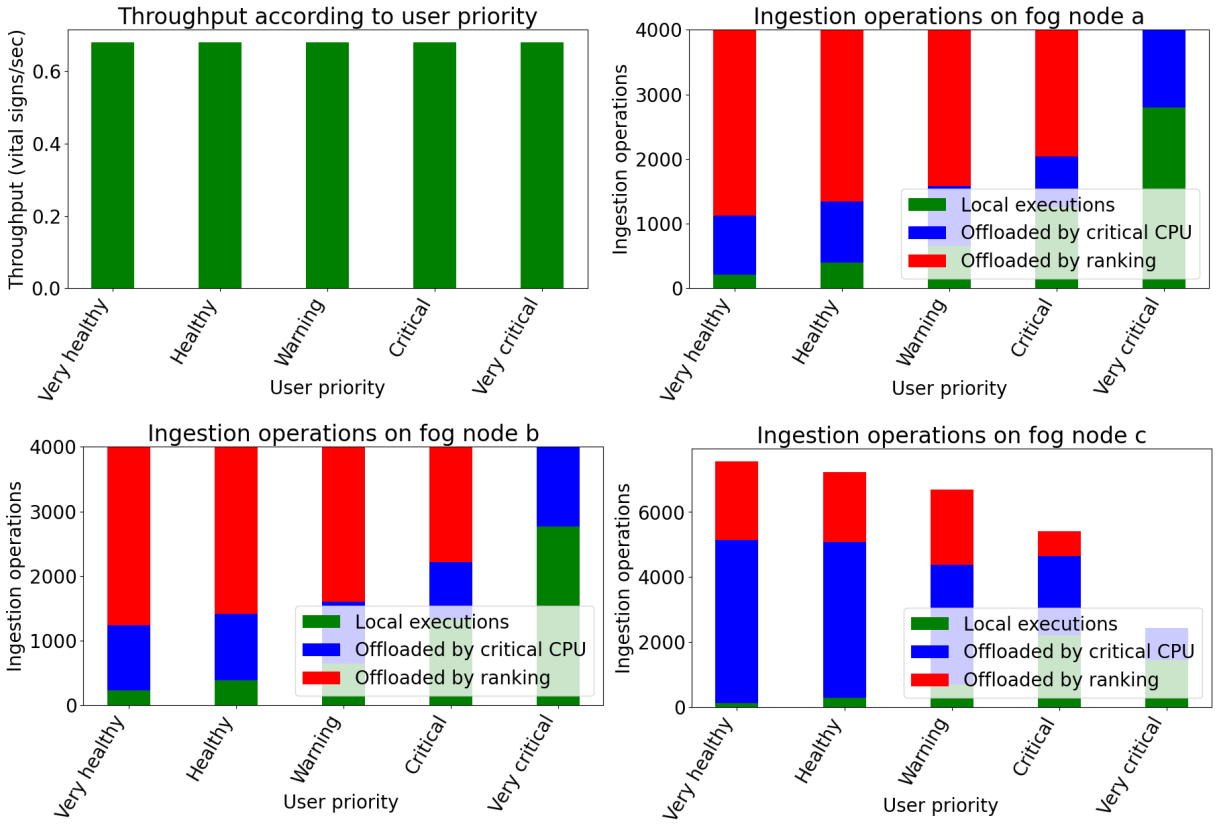


Source: prepared by the author.

fog in a few minutes. Let us also suppose that, at a given moment during the experiment, a single vital sign regarding a user with critical priority was offloaded to the cloud because the fog was completely overloaded, and waited for two hours in the queue until being processed. This will directly reduce the calculated throughput for vital signs with this user priority. Since the vital signs are added at the end of the queue, and this queue has the limitation of not considering a heuristic to favor critical vital signs, it may take a long time until this critical vital sign is processed. At first glance, simply looking at the throughput chart presented in Figure 34 looks like the architecture is not favoring critical vital signs. Still, in practice, it is favored, since smaller percentiles will show that the response time is smaller for critical vital signs. However, this may still not be acceptable for real-world usage, as no critical vital sign should take so long to be processed.

Figure 34 also details how the offloading operations behave on the fog nodes and indicates the number of vital signs processed locally. The offloading process is working as expected: the healthier the person is, the higher the number of vital signs offloaded because of the ranking heuristic. The duration heuristic is never triggered in this specific experiment because this experiment was done with a single health service, but this heuristic requires durations of at least two different health services to be triggered. Further sections of this thesis will present results for experiments with both heuristics by running multiple health services simultaneously. We see similar offloading behavior on fog nodes a and b, as they have the same computing resources and receive the same number of vital signs from edge nodes (representing people in a smart city). When exceeding the critical CPU threshold, or when the ranking heuristic indicates that the vital sign should not be processed locally, the vital sign is offloaded to the fog node c

Figure 34: Throughput for the experiment with fog nodes and a single queue on the cloud.



Source: prepared by the author.

because the aforementioned fog nodes are directly connected to the latter. We emphasize that vital signs for *very critical* user priority are never offloaded because of the ranking heuristic on this experiment and are only offloaded when the CPU exceeds the critical CPU threshold. Also, vital signs for this user priority are usually processed locally, which is different from the other user priorities. The *very healthy* user priority, for example, has approximately 200 vital signs processed locally, while *very critical* user priority has around 3.000 vital signs processed locally. The number of vital signs offloaded because of exceeding the critical CPU usage is similar for every user priority, though: the machine is running out of resources in this scenario and therefore unable to execute the health service locally, regardless of the user priority. Since this experiment simultaneously sends the same number of vital signs for each user priority, it is expected that the number of offloading operations by exceeding the critical threshold is similar for each user priority. Regarding the ranking heuristic, it is notable that the *very healthy* user priority has been more offloaded than other user priorities. The more critical the vital signs, the less the offloading operations. Finally, this figure also makes it clear that fog nodes *a* and *b* received exactly 20.000 vital signs each, being 4.000 vital signs of each priority.

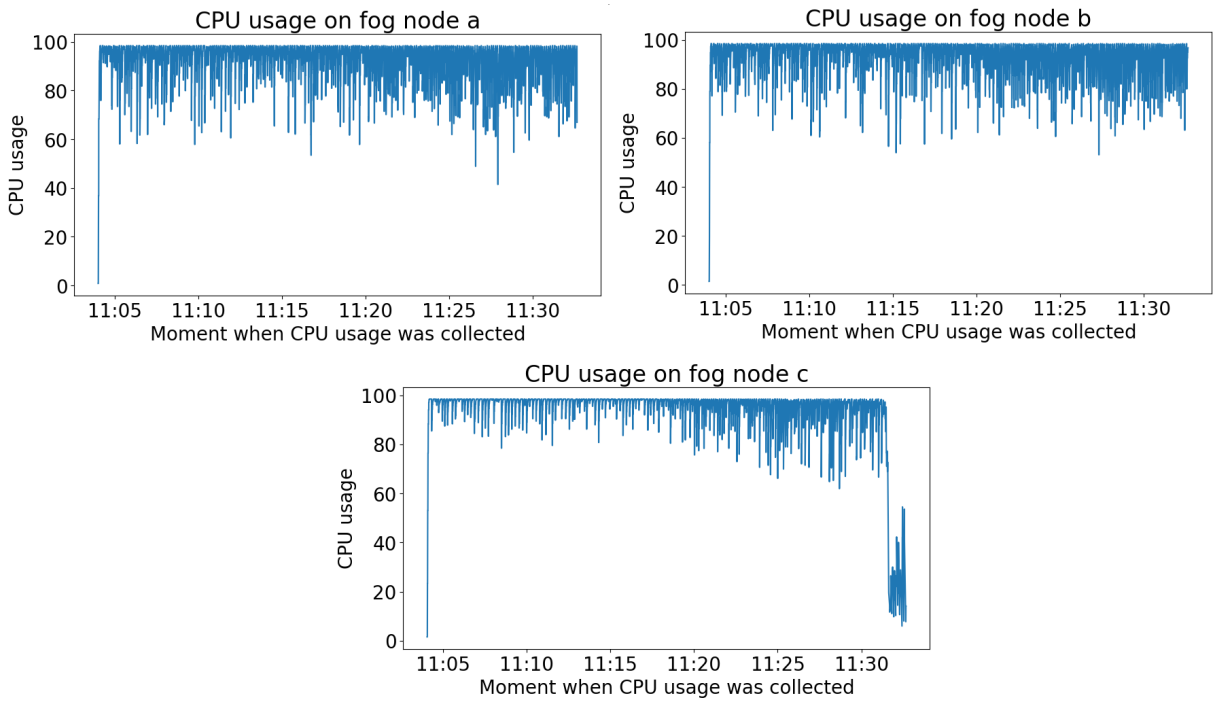
On its turn, Figure 34 also presents how the fog node *c* handles vital signs that were offloaded by fog nodes *a* and *b*. All vital signs received by this fog node were sent by fog nodes at

the lower level on the hierarchy, which is, the edge nodes did not send vital signs directly to fog node *c*. An interesting behavior is that the offloading reasons for the fog node *c* is quite different from the other fog nodes. Most vital signs were offloaded because the CPU usage exceeded the critical CPU threshold and not because of the ranking heuristic. This happens because this machine is the parent of two fog nodes and for this reason receives a higher number of vital signs simultaneously. This can be seen by the number of vital signs for each user priority that is presented on the chart. Except for the *very critical* user priority, all priorities exceeded the number of 4.000 vital signs, representing a higher number of vital signs being received when compared to fog nodes on the first level of the hierarchy. Also, a higher number of vital signs with *critical* priority was executed locally compared to vital signs with *very critical* priority. This can be arguably considered wrong at first glance, as the expectation is to have more vital signs with higher priorities being executed locally. However, there is a key difference for this specific fog node: a total of 2.432 vital signs with *very critical* priority was received by fog node *c*, while this same fog node received 5.410 vital signs with *critical* priority. The number of vital signs with the former priority represents only 44,95% when compared to the latter on this fog node. Therefore, since the number of vital signs with *critical* priority is higher, it is acceptable to consume more vital signs locally. More vital signs with *critical* user priority were offloaded to the fog node *c*, but as a percentage, the algorithm sends the appropriate number of vital signs to the cloud.

The following paragraphs will give insights regarding the CPU usage on fog nodes during the experiment. In summary, fog nodes maintained high CPU levels most of the time. It is important to highlight that the presented values are smoothed based on the last 6 CPU observations to reduce the impact of spikes and outliers. Figure 35 presents the CPU usage on the three fog nodes used during the experiment. Fog nodes *a* and *b* received 20 vital signs each, from two edge nodes, while each edge node sends 10 vital signs simultaneously. This figure shows that both fog nodes have similar behavior regarding CPU usage. The usage maintains high most of the time, but several peaks make the line go up and down. We understand that this is a consequence of the offloading operations: the thread on the edge node remains synchronously blocked while the fog node is offloading the vital sign to the parent fog node. In turn, the edge node has one less thread sending vital signs because the thread keeps waiting for the response of the offloading operation. A possibility to reduce these differences in CPU usage is to employ a queue on each fog node instead of making synchronous calls that make the thread wait for a response. This way, the fog node would offload the vital sign to the parent fog node and not hold the thread until the health service is processed. We emphasize that the last fog node does not wait until the vital sign is processed when it offloads the vital signs to the cloud, which is processed asynchronously. Finally, this figure also presents the CPU usage for the fog node *c*, which is the root node on the hierarchy and receives vital signs that were offloaded from fog nodes *a* and *b*. In other words, the fog node *c* never receives vital signs directly from the edge nodes (representing people in a smart city) but always from fog nodes on the lower level of the

hierarchy when these nodes cannot process vital signs locally. The CPU behavior presented for fog node *c* has a similar format when compared to the CPU usage on fog nodes *a* and *b*. Still, for the fog node *c*, the fluctuation is smaller and usually ranges between 80% and 100%. This means that the CPU was so busy that vital signs were usually offloaded by exceeding the critical CPU threshold and not because of the heuristics, regardless of the priority. This makes sense, as two fog nodes were sending vital signs simultaneously to the fog node *c*, therefore receiving a larger number of vital signs than the other fog nodes.

Figure 35: CPU usage collected on fog nodes during the experiment.



Source: prepared by the author.

Another contributing factor to this unstable behavior on CPU usage on fog nodes *a* and *b* is the time interval of 1 second to collect CPU observations. This means that all offloading decisions are made considering a frozen value corresponding to the CPU usage of 1 second before. An interesting experiment to be done is reducing this time window to half a second to see if the spikes in CPU usage are reduced. Another interesting value to pay attention to is the warning and the critical thresholds: as the warning threshold is set to 60%, all vital signs are processed locally when the current CPU usage is below 60%, which causes no offloading operation to happen regardless of the user priority. This explains why CPU usage fluctuates between 60% and 98%, but usually not much below the warning threshold. We highlight that the timestamp range for charts in Figure 35 is around 30 minutes because it is how long it takes for the edge nodes to send vital signs to fog nodes, while many of them are offloaded to the cloud and processed asynchronously after the first 30 minutes. Finally, another interesting

behavior on the CPU usage for fog node *c* can be noticed after 11:30 AM, when edge nodes are finalizing the process of sending vital signs, but a noticeable drop in CPU usage did not appear for the other fog nodes. Our understanding of this is the following: fewer vital signs are being offloaded from the fog nodes *a* and *b* to fog node *c* at the end of the experiment, which causes the CPU usage on the latter fog node to be reduced as there are fewer vital signs being received and processed.

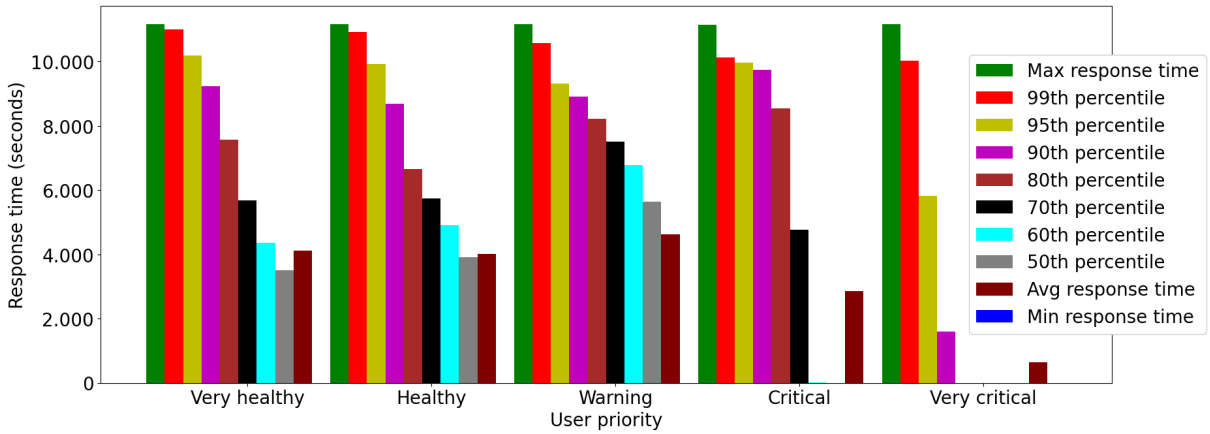
Figure 36 complements the previous paragraphs and introduces a chart with percentiles of the response time for each user priority, while Table 10 presents the values used to plot this chart. Results indicate that the response time for users with *very critical* priority is shorter than other priorities, which means the algorithm is correctly favoring urgent vital signs by running them on fog nodes whenever possible. Less critical vital signs, though, take longer to complete because more critical vital signs were favored. The maximum response time and the 99th response time percentile are similar for vital signs with user priorities *very healthy*, *healthy*, and *warning* because they were offloaded to the end of the queue on the cloud at some point during the experiment. Regarding user priorities *critical* and *very critical*, the response time on the 99th percentile is notably shorter than the maximum response time, though, as 99% of the vital signs were processed in no more than 10.136 and 10.039 seconds, respectively. The 95th percentile, on the other hand, shows a great difference for vital signs regarding the *very critical* user priority: they were processed in no more than 5.832 seconds (100 minutes), while these two percentiles are close to each other for the other user priorities. Still, on user priority 5, there can be seen an improvement of 40% between the 99th and 95th percentiles, representing 10.039 and 5.832 seconds, respectively.

Table 10: Response time (seconds) according to user priority. Priorities 1, 2, 3, 4, and 5 represent *very healthy*, *healthy*, *warning*, *critical*, and *very critical*, respectively.

Pri.	**Max**	**99th**	**95th**	**90th**	**80th**	**70th**	**60th**	**50th**	**Avg**	**Min**
1	11.173	10.996	10.193	9.231	7.573	5.675	4.361	3.504	4.127	0,163
2	11.172	10.923	9.935	8.683	6.653	5.736	4.903	3.911	4.016	0,170
3	11.170	10.588	9.328	8.911	8.219	7.515	6.786	5.632	4.623	0,165
4	11.144	10.136	9.975	9.754	8.547	4.772	9,168	3,447	2.855	0,154
5	11.170	10.039	5.832	1.600	3,617	2,409	1,892	1,489	650	0,163

We see an even more discrepant behavior for vital signs with *very critical* user priority regarding the 90th percentile, as 90% of the vital signs were ingested in no more than 1.600 seconds. In other words, a person with serious health problems will not wait more than 27 minutes 90% of the time until a health service processes their vital sign. This is still arguable, though, as a person having a heart attack might need an even faster response. The 90th percentile for this user priority is 66% better than the 95th percentile and 80% better than the 99th percentile, which represents a big difference in response time for such a small difference in the percentiles. For user priorities *warning* and *critical*, though, the 90th percentile does not

Figure 36: Percentiles of the response time for the experiment with three fog nodes and a single queue on the cloud. There is a clear difference in the response time for vital signs regarding users in very critical conditions.



Source: prepared by the author.

have such a significant difference compared with the 95th percentile. For user priorities *very healthy* and *healthy*, there is a more evident difference between these two percentiles but are still extremely higher when compared to the *very critical* user priority. The 80th response time percentile for *very critical* user priority is not visible on this chart because it was processed in no more than 4 seconds. This suggests that they were synchronously processed on fog nodes without being offloaded to the cloud. It is arguably appropriate for a person in urgent conditions: results for health services processed with these vital signs will be received almost instantly. The 80th percentile continues high for other priorities, ranging from 6.653 seconds (111 minutes) to 8.547 seconds (143 minutes). User priorities *warning* and *critical* have a slight difference in response time between the 80th and 90th percentiles, while user priorities *very healthy* and *healthy* have a more noticeable difference between these two percentiles. This behavior is quite interesting because, in theory, higher priorities would always have a shorter response time. Still, this did not happen in practice: the response time for user priorities *very healthy* and *healthy* is shorter than user priorities *warning* and *critical* in the 80th percentile. This chart also indicates a thought-provoking behavior for user priority *warning*, which has a longer response time for the analyzed percentiles when compared to people in healthier conditions. Our understanding is the following: vital signs with user priorities *warning* onwards are typically processed on fog nodes, while priorities *very healthy* and *healthy* are usually offloaded to the cloud. This means that vital signs with lower priorities are dominant in the cloud. Also, an important characteristic is that all user priorities share the same queue, so vital signs are usually polled from the queue in the order they arrive. However, vital signs with priority *warning* are more suitable for being offloaded to the cloud when compared to more critical priorities depending on the ranking heuristic and the percentage of used CPU. With this in mind, since the queue already

has several vital signs waiting to be consumed, vital signs with priority *warning* are added to the end of the queue. The consequence is that it ends up having a longer waiting time. Future work could address multiple queues on the cloud to mitigate this behavior, while each queue could be specific for a single user priority.

There is also a noticeable difference for *critical* user priority regarding the 70th percentile because vital signs are processed in no more than 4.772 seconds (80 minutes). This is 55% better than the 80th percentile for this same user priority, where vital signs are processed in no more than 8.547 seconds (143 minutes). This may still be considered a long time for such an urgent user priority. On the other hand, it is faster than the 70th percentile for priorities *very healthy*, *healthy*, and *warning*, where response times range from 5.675 seconds (95 minutes) and 7.515 seconds (126 minutes). Also, response times for *critical* priority stand right behind user *very critical* priority and represent the second smallest response time for the 70th percentile. Vital signs for *very healthy* and *healthy* user priorities are close to each other for this percentile, with 5.675 and 5.736 seconds, respectively. User priority *warning* takes up to 7.515 seconds to process vital signs for this percentile, as well as interestingly takes longer to be processed when compared to healthier user priorities, as previously explained.

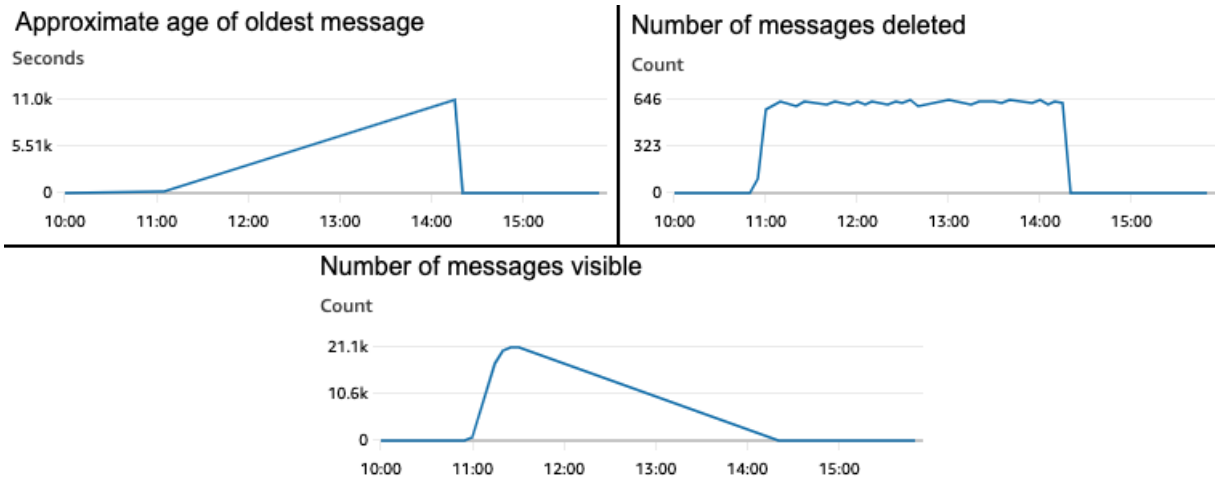
The 60th response time percentile can only be seen in Figure 36 for non-critical user priorities, which means, urgent vital signs were processed almost instantly. In practice, there is no such difference between the 60th and 70th percentile regarding *very critical* user priority, which takes up to 2,5 seconds and 1,9 seconds to have vital signs processed, respectively. However, this is not the same for *critical* user priority, where there is a great difference between them: 70% percent of vital signs were processed in up to 4.772 seconds (79 minutes), while 60% of the requests were processed in up to 9,168 seconds. This is a huge difference of 52.050%, which happens because of the offloading operations to the cloud. The queue already has several messages waiting to be processed but the vital sign is added to the end of the queue, which means it needs to wait for a long time until being processed. On the other hand, vital signs that could be processed on the fog node were processed in a few seconds. This shows the effectiveness of running vital signs in the fog versus sending them to the cloud when the queue is heavy.

Finally, regarding the 50th percentile, response time is similar for *critical* and *very critical* user priorities. This value is up to 1,489 seconds for *very critical* user priority and up to 3,447 seconds for *critical* user priority, but ranges from 3.504 seconds (59 minutes) and 5.632 seconds (95 minutes) for the other priorities. Explicitly talking about the *very critical* priority, this percentile is close to the values for the 60th percentile, which is 1,489 and 1,892 seconds, respectively. This represents a difference of 21,3%. From now on, percentiles for *critical* and *very critical* priorities will not have much difference because they are already very small. This can still be improved for other priorities, though, especially to maintain an ordered set of results for them. In other words, response times for healthier priorities should never be lower than response times for urgent user priorities.

Since the previous paragraphs presented the behavior of the fog nodes and showed per-

centiles for ingesting vital signs, incoming paragraphs will focus on details regarding the queue in the cloud. We emphasize that architecture *c* is proposed in a way that only the last fog node on the hierarchy sends vital signs to this queue. It takes around 30 minutes to generate all vital signs during the experiment, but the queue continues being asynchronously consumed until all messages are processed, even though it takes more than 30 minutes. The queue is consumed by 10 replicas of serverless functions, while 10 is the default limit available for AWS accounts before requesting an increase in quotas. Different metrics are presented in the following paragraphs, while a detailed explanation of what each metric means can be found on the AWS official documentation for CloudWatch (AWS, 2022b), which is the service we used to generate charts for resources on the cloud.

Figure 37: Compilation of metrics regarding usage of the queue in the cloud. This figure encompasses charts that present the waiting time in the queue, the number of messages deleted in an interval of 5 minutes, and the number of messages visible in an interval of 5 minutes.



Source: prepared by the author.

Figure 37 shows a compilation of metrics regarding the queue in the cloud. The first chart on this figure shows the age of the oldest message in the queue, which means how long the oldest message stayed in the queue until a replica of the serverless function processed it. The oldest message stayed in the queue for around 11.100 seconds (185 minutes) until it was processed, which matches the moment when the messages stopped being generated by the edge nodes. Generation of vital signs finished at around 11:30; since then, 185 minutes have passed until all messages could be consumed and processed from the queue. At 14:30, this metric went down to zero because a replica of the serverless function finally processed the oldest message on the queue. The chart on the right presents the number of messages deleted from the queue considering a time interval of 5 minutes. The message is only deleted when processed successfully by the serverless function, or stays in the queue to be reprocessed later. In other words, the vital sign is only removed from the queue when the health service can process the

message successfully and send results to the *notificator* module. This chart also indicates that the number of removed messages (successfully processed messages) is around 646 each five minutes, which means that serverless functions consume messages at their speed based on the limit of function replicas. In this experiment, the number of replicas is 10 because of the quota limit for the AWS account. The chart line tends to zero as soon as there is no message on the queue. Another interesting insight from this chart is that consumers process vital signs as soon as the queue has messages available. To be more specific, the current chart makes it clear that messages are consumed as soon as the queue is not empty. Consumers do not wait 30 minutes (until all the vital signs are generated by edge nodes) to start consuming messages from the queue, for example.

Finally, the chart on the bottom part of Figure 37 shows the number of visible messages in the queue during the experiment. By visible messages, AWS means the messages available on the queue that have not been processed by any consumer yet. In this case, the consumer is a replica of the serverless function. This chart indicates that many vital signs were offloaded to the cloud during the first 30 minutes of the experiment while edge nodes were artificially generating vital signs. Still, the fog nodes could not process all of them because fog nodes were getting overloaded. These vital signs were offloaded to fog node *c* and, in turn, were offloaded to the cloud. Messages on the queue are automatically consumed whenever consumers (replicas of serverless functions) are available to process them. However, this chart clearly indicates that the number of produced messages is larger than the number of consumers, as the number of visible messages greatly increases in the first 30 minutes. This is also an interesting aspect of this architecture because it can handle high usage peaks and the system will still be able to process all messages, regardless of whether it will take more or less depending on the number of messages waiting to be processed. In summary, no message will be lost, but this can turn out to be a problem if there is a large concentration of people in a single neighborhood and most vital signs end up being offloaded to the cloud. It could take days until all messages were processed in the cloud, but there is not much sense in alerting a person that is close to having fever *x* days after receiving the vital sign. Approaches to process messages with less waiting time include increasing the number of fog nodes, so fewer vital signs will be offloaded to the cloud, or increasing the quota of serverless functions replicas with the cloud provider, which may result in extra costs.

6.5 Scenario 5 - High load - Many vital signs to fog and cloud with two services and Architecture C

This experiment is somewhat similar to the experiment presented in Section 6.4, but significantly differs by processing vital signs with two health services instead of a single one. We evaluate two variations of this scenario by changing the warning threshold of the CPU, which is the threshold where offloading heuristics start being triggered based on ranking or service

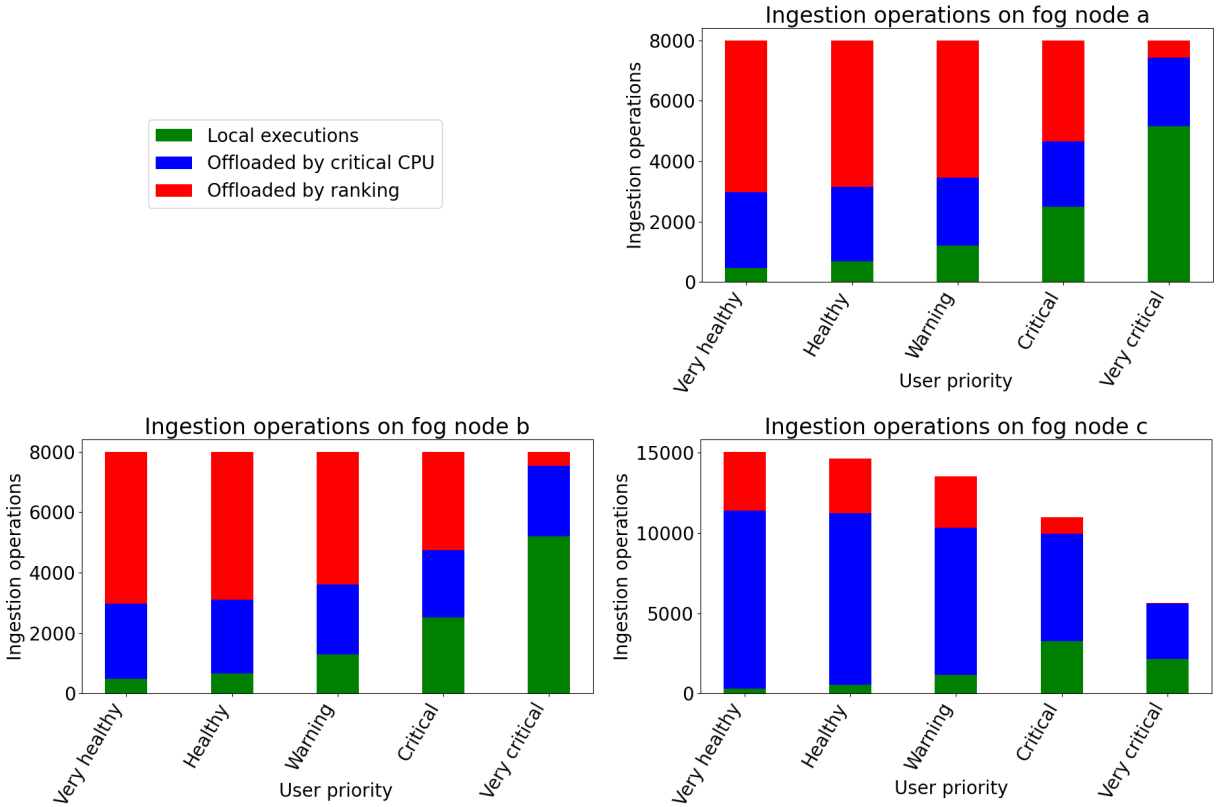
duration. This experiment considers a total number of 80.000 vital signs, being 40.000 processed with the *body-temperature-monitor* health service, and the other 40.000 being processed with the *heart-failure-predictor* health service. The previous experiment presented in Section 6.4 focuses on the ranking heuristic because only the *heart-failure-predictor* health service was used to process vital signs. This means that offloading operations were mainly decided by either the ranking calculated for the vital sign or by exceeding the critical CPU usage threshold. However, the module *duration heuristic* introduced in SmartVSO has not been triggered in the previous experiments since the duration heuristic requires at least two different services running concurrently on the fog node to compare their durations. Detailed information about how the duration heuristic works can be revisited in subsection 4.4.3.

Variation #1 - Lower warning threshold: this first variation considers a warning threshold of 60% and a critical threshold of 98% to offload vital signs based on CPU usage. We emphasize that ranking and duration heuristics are only triggered when the current CPU usage is between 60% and 98%. In addition, this variation considers 1 second between each CPU collection and uses the aging technique. Regarding the workload, this variation of the experiment was evaluated with: *i)* four edge nodes, *ii)* 20 threads on each edge node sending vital signs simultaneously to the leaf fog nodes (representing 80 vital signs being sent simultaneously because we have four edge nodes), and *iii)* a total of 20.000 vital signs on each edge node. In summary, 5 priorities * 2 threads per priority * 1.000 vital signs per thread * 2 health services = 20.000 vital signs per edge node. Since we have four edge nodes, this results in a total number of 80.000 vital signs generated during the experiment. Two attempts were made until we got a successful result for this variation, which will be presented as follows.

Execution attempt #1 - Failure: the first attempt to execute this variation of the experiment failed, as we noticed that the duration heuristic was never offloading vital signs. The first version of the algorithm was only considering the durations of health services processing vital signs with the same ranking that was calculated for the vital sign being ingested. For example, let us suppose that several vital signs are being processed with services A, B, and C. Let us also suppose that the vital sign being ingested has a calculated ranking of 7. The algorithm was initially proposed to ignore health services processing vital signs with a ranking different than 7, which ended up limiting the effectiveness of the duration heuristic. We noticed that many times there were not many vital signs being processed with the same ranking, as the duration heuristic is only triggered on specific scenarios, when the ranking heuristic is not able to determine the offloading operation. Figure 38 shows the number of offloading operations on fog nodes a, b, and c, while the first two receive vital signs directly from the edge nodes (representing people in a smart city). We notice the ranking heuristic correctly offloading vital signs based on the calculated rankings, followed by several offloading operations based on exceeding the critical CPU threshold. However, the charts presented in this figure do not show any offloading operation due to the duration heuristic. This is not expected: the service *body-temperature-monitor* executes much faster than *heart-failure-predictor*, so it was expected that many vital

signs processed with the second service were offloaded to the parent level of the hierarchy. In this same figure, the chart regarding offloading operations on fog node *b* is similar to fog node *a* because both receive the same number of vital signs directly from the edge nodes, and both virtual machines have the same computing resources available. Interestingly, the offloading operations on the fog node *c* indicate that this fog node received many more vital signs with *very healthy* user priority (exceeding 14.000 vital signs) because many vital signs with this priority were offloaded from both the fog nodes *a* and *b*. Fewer vital signs are being processed on fog node *c* with more urgent priorities because most were directly processed on fog nodes *a* and *b* and did not need to be offloaded. Since fog node *c* receives vital signs from two fog nodes, the CPU usage is usually higher than fog nodes on the lowest level of the hierarchy. Therefore, many offloading operations happened because of exceeding the critical CPU threshold. These three charts indicate no offloading operation happened because of the duration heuristic, while the offloading operations happened either by decision of the ranking heuristic or by exceeding the critical CPU threshold.

Figure 38: Ingestion operations on each fog node of architecture *c* during the variation #1 of the fifth scenario. No offloading operation happened due to the duration heuristic, which lead us to the conclusion we need to modify the heuristic and stop filtering vital signs with the same calculated ranking.



Source: prepared by the author.

Based on the results presented in the previous paragraphs and the fact that the first version

of the duration heuristic filters vital signs with the same ranking, we then decided to change this heuristic to consider the duration of all vital signs being processed, regardless of the ranking associated with the vital sign. The main idea of the latest version of the heuristic is presented in Subsection 4.4.3 and summarized as follows: given multiple health services being executed on the current fog node, checks if the service associated with the incoming vital sign will be slower or faster than the other services already running, regardless of the ranking of vital signs being processed by these services. The incoming vital sign will be processed locally if the service is faster than other services. Otherwise, it will be offloaded to the parent fog node or the cloud. The result for the algorithm with this change is shown in the following execution attempt.

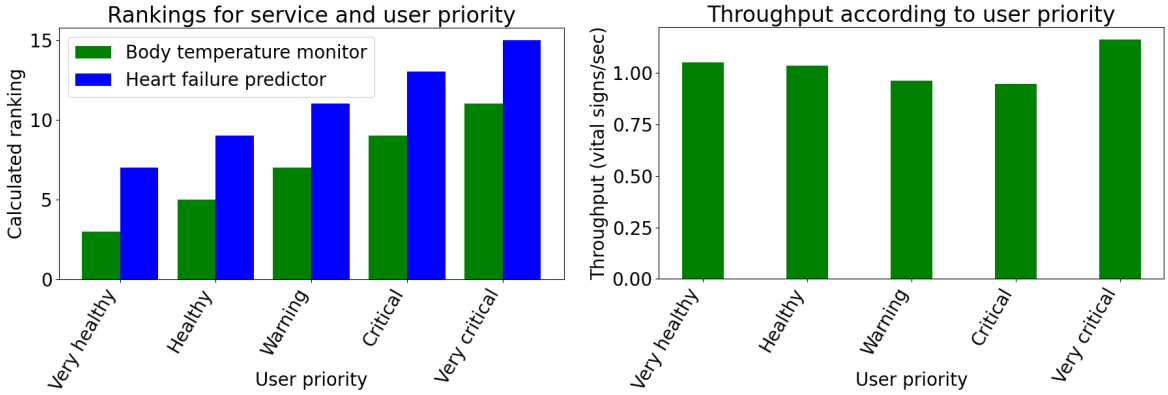
Execution attempt 2 - Success: this second attempt to run the experiment was made using the modified version of the duration heuristic, which also considers health services processing vital signs with different calculated rankings. As explained in the previous paragraphs, no offloading operation was made because of the duration heuristic on the previous attempt, since it only considered vital signs with the same calculated ranking. We modified the source code for this heuristic and have noticed exciting results. Table 11 presents the ranking for each combination of user priorities and health services considered in the experiment, while Figure 39 presents this same information in the form of a chart. We reiterate that user and service priorities range from 1 to 5, but user priority has double weight in the ranking calculation, as presented in Subsection 4.4.1. Given that the *body-temperature-monitor* health service has priority 1 and the *heart-failure-predictor* health service has priority 5, the following table presents the rankings calculated for each combination of user priority and service priority. We reiterate that user priority is dynamic and comes from the request payload, representing the user priority of the person at that moment. In contrast, the service priority is static and is the same for all users.

Table 11: Rankings calculated for each combination of health service and user priority.

Health service	**User priorities**				
	Very healthy	**Healthy**	**Warning**	**Critical**	**Very critical**
Body temperature monitor	3	5	7	9	11
Heart failure predictor	7	9	11	13	15

An interesting behavior can be noticed in the priorities chosen for the services in this experiment. The ranking calculated for *body-temperature-monitor* health service with *very critical* user priority is 11, which is the same ranking calculated for *heart-failure-predictor* health service with *warning* user priority. This happens because the service *body-temperature-monitor* has priority 1 (the smallest possible priority), and the service *heart-failure-predictor* has priority 5 (the maximum possible priority). Therefore, there will be a ranking collision when the CPU usage is between the warning and the critical thresholds and both health services are running simultaneously. The ranking heuristic will be unable to determine the offloading decision because both executions have the same ranking assigned. The final offloading decision will be

Figure 39: Rankings calculated for each health service and user priority combination, as wella s the throughput for each user priority at the end of the experiment. The *very critical* user priority has the highest throughput, but *warning* and *critical* priorities have smaller throughputs than healthier people.

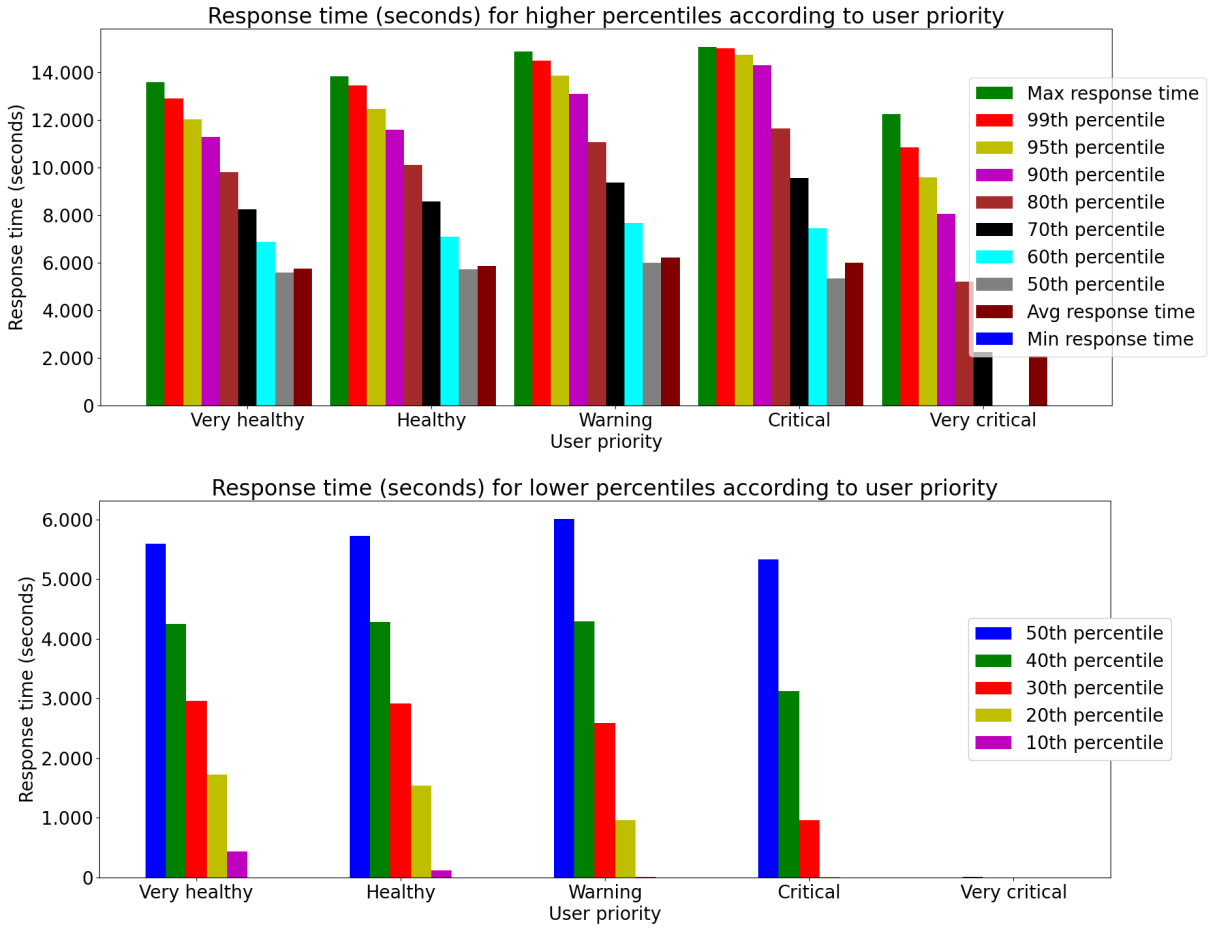


Source: prepared by the author.

leveraged to the duration heuristic, which is responsible for comparing the duration of the services. In practice, the *body-temperature-monitor* health service executes fast because it simply compares with a given threshold the temperature found on the vital sign. On the other hand, the *heart-failure-predictor* health service takes longer to execute, as it makes predictions based on historical data to tell whether the person is close to having a heart attack. The duration heuristic will check the previous durations for these services and understand that the *body-temperature-monitor* health service is faster, executing it locally when there is a ranking collision. Figure 39 also presents the throughput calculated for each user priority. The overall throughput is not so different for each user priority because vital signs of other priorities are eventually offloaded to the end of the queue on the cloud when fog nodes are overloaded. These values are outliers and directly impact the calculation of throughput. Subsection 4.4.1 explains that the ranking is calculated as a combination of the user priority with the service priority, and different health services may have specific priorities depending on the classification of a doctor. Offloading decisions are not simply done considering the user priority but considering this calculated ranking, which means that services may have a higher or lower probability of being offloaded when working with vital signs of the same user priority.

Figure 40 shows the percentiles for each user priority collected after running this experiment, regardless of the health service. This figure is composed of two charts, the first presenting higher percentiles and the second presenting lower percentiles. We noticed a great difference regarding low percentiles for *very critical* user priority, which is why we decided to plot a specific chart to have a better understanding of this. At first sight, results may not look promising because priorities ranging from *very healthy* to *critical* share similar response times from the 99th to the 50th percentile. In theory, the response time should be lower as the user priority

Figure 40: Percentiles of the response time for the experiment with multiple health services. The more critical the user priority, the smaller the response time for lower percentiles. Interestingly, regarding higher percentiles, healthier people have results with a shorter response time.



Source: prepared by the author.

increases, but this is not what happened for higher percentiles. On the other hand, the response time percentiles for *very critical* user priority have quite interesting behavior: regarding the 50th and 60th percentiles, the response time ranges between 3 and 5 seconds, respectively. However, there is a massive difference when moving from the 60th to the 70th percentile: a vital sign may take up to 2.235 seconds (37 minutes) to receive a response. This difference happens when the vital sign is offloaded to the end of the shared queue in the cloud and the queue already has several messages waiting to be consumed. Even though the 70th percentile looks high for the *very critical* user priority, it is smaller than the other user priorities, such as the *critical* priority, which takes up to 9.561 seconds (159 minutes) to process 70% of the vital signs. We conclude that higher response time percentiles are similar among user priorities when a large number of vital signs are offloaded to the cloud, as no algorithm has yet been employed to favor urgent vital signs in the cloud. The second chart on this figure presents the response time, in seconds, for smaller percentiles that could not be visible in the first chart regarding higher percentiles.

There is a big difference in response time when comparing small percentiles with higher ones, especially regarding the *very critical* user priority. No value could be seen for this user priority in the first chart for the 50th percentile and below, representing health services processing vital signs almost instantly in these scenarios.

Table 12 presents the same numbers used to plot charts for Figure 40. We noticed that, until the 40th percentile, response time for ingesting vital signs is directly related to user priority: the higher (urgent) the priority, the smaller the response time. This happens because these vital signs could be successfully processed in the local fog nodes instead of being offloaded to the cloud. We see undesired behavior in this experiment from the 50th percentile onwards for *warning* and *critical* user priorities, as response time percentiles are higher when compared to *very healthy* and *healthy* priorities. In turn, Table 13 complements the analysis regarding response times and shows information about the maximum, minimum, and average response times. The minimum is less than half a second regardless of the user priority, as at least one vital sign for each priority was processed locally in an arbitrary fog node during the experiment, without being offloaded to the cloud. This is especially true at the beginning of the experiment, where fog nodes were not processing any vital signs and therefore had more CPU available. Also, the maximum response time is extremely high for all user priorities, as at least one vital sign of each priority was offloaded to the end of the shared queue in the cloud when computing resources were unavailable on the fog nodes. This employs a higher waiting time when the queue has a high number of messages waiting to be processed. Finally, the average response time for processing vital signs ranges between 2 and 6 seconds, approximately.

Table 12: Response time (seconds) according to user priority. Priorities 1, 2, 3, 4, and 5 represent *very healthy*, *healthy*, *warning*, *critical*, and *very critical*, respectively.

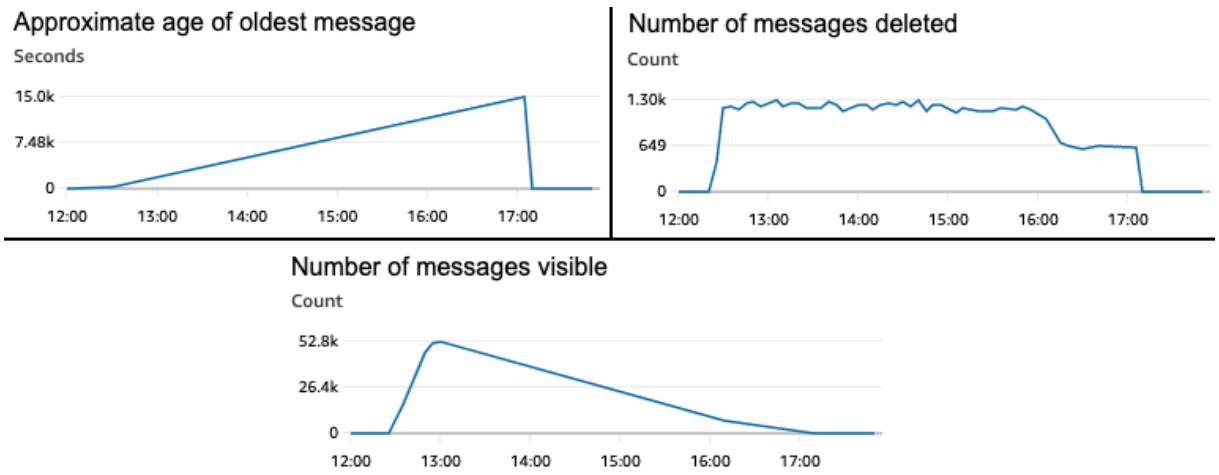
Pri.	**99th**	**95th**	**90th**	**80th**	**70th**	**60th**	**50th**	**40th**	**30th**	**20th**	**10th**
1	12.920	12.024	11.298	9.816	8.256	6.873	5.596	4.243	2.958	1.721	436
2	13.446	12.475	11.585	10.118	8.573	7.083	5.724	4.280	2.912	1.536	113
3	14.485	13.866	13.093	11.080	9.382	7.678	6.009	4.294	2.589	961	3,481
4	15.029	14.752	14.294	11.644	9.561	7.453	5.334	3.118	958	3,796	1,426
5	10.843	9.579	8.067	5.198	2.235	5,258	2,775	1,605	1,079	0,788	0,594

Table 13: Additional information regarding response time (seconds). Priorities 1, 2, 3, 4, and 5 represent *very healthy*, *healthy*, *warning*, *critical*, and *very critical*, respectively.

Pri.	**Max.**	**Min.**	**Avg.**
1	13.600	0,039	5.752
2	13.828	0,030	5.868
3	14.888	0,031	6.209
4	15.082	0,045	6.006
5	12.255	0,045	2.079

The following paragraphs will present the queue's behavior on the cloud during the experiment. The first chart presented in Figure 41 indicates the approximate age of the oldest message in the queue, which means how long the oldest message had to wait until being processed by an available serverless function. Our approach uses a single queue without favoring messages based on their priorities, which can be improved in future work. In this experiment, the message that stayed in the queue for the longest time had a waiting time of around 15.100 seconds (4 hours and 11 minutes). This can be a problem if a user in urgent conditions (associated with the *very critical* user priority) is having health problems, which is unacceptable to wait 4 hours to get a response. It is important to reiterate that this experiment generated 80.000 vital signs while Scenario 3 generated 40.000 vital signs, representing the double number of vital signs to be processed. This makes it clear that the higher the number of messages offloaded to the queue in the cloud, the longer it will take for messages to be processed. Approaches to mitigate this undesired behavior of messages waiting a long time in the queue include: *i)* adding more fog nodes in the neighborhoods so that more vital signs can be processed locally, *ii)* requesting an increase in quotas for serverless functions on the cloud provider to process more vital signs simultaneously, and *iii)* employing strategies to favor critical vital signs when consuming messages from the queue in the cloud.

Figure 41: Metrics regarding the queue on the cloud. These charts help us understand how vital signs are asynchronously consumed when could not be processed by fog nodes.



Source: prepared by the author.

The chart on the right part of Figure 41 presents the number of messages deleted from the queue in the interval of 5 seconds. We see fewer messages being deleted after 16:00 because there is a concentration of messages regarding the *heart-failure-predictor* service, which is more heavyweight and takes longer to execute. The total time interval for plotting this chart ranges from 12:30 to 17:30, representing the moments when the experiment started and the last message was consumed from the queue, respectively. We see a pattern in message deletion from

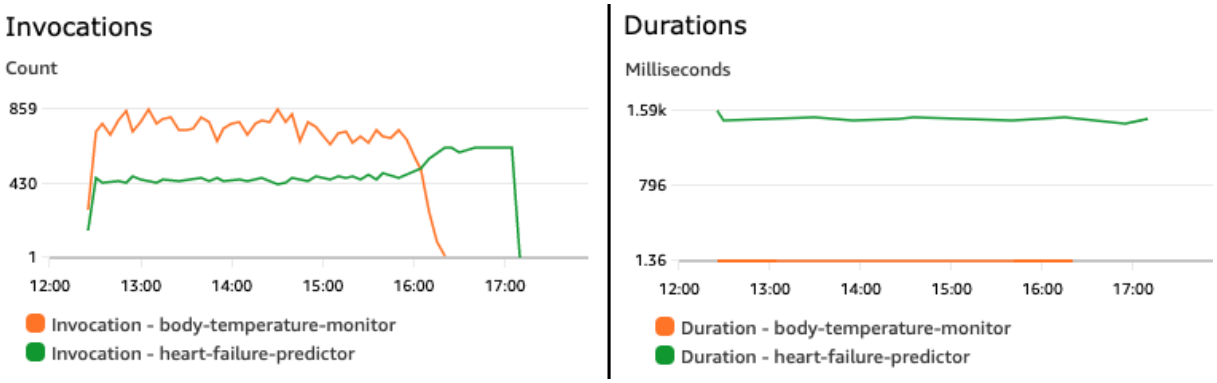
when the experiment started until around 16:00, which indicates around 1.20k and 1.30k deletions every 5 minutes. After 16:00, the number of messages deleted from the queue was reduced by half. This happened because there was a concentration of messages regarding the *heart-failure-predictor* health service at that moment. This health service takes longer to execute than the *body-temperature-monitor* health service. Also, messages for *body-temperature-monitor* have already been consumed from the queue at 16:00. For this reason, all serverless functions were busy processing the more heavyweight health service that takes longer to complete. This results in a smaller number of messages processed in the time window of 5 minutes and hence fewer messages being deleted from the queue.

Finally, the chart at the bottom part of Figure 41 gives information about the messages visible in the queue on the cloud by considering a time window of 5 minutes. We see vital signs being added to the queue until 13:00, which is the moment when edge nodes finish generating vital signs for the experiment. After this moment, no other message is added and the queue is only consumed. We see an increase during the first 30 minutes, which is the time that edge nodes take to generate and send artificial vital signs to fog nodes. Vital signs that could not be processed locally on the fog nodes were offloaded to the cloud during these 30 minutes. After 13:00, there was a decrease in the number of messages visible on the queue, which means that messages were being consumed by the serverless functions (health services). Around 16:00, there is a slightly lower speed for consuming messages, as the line in the chart is less inclined. This is explained by the fact that vital signs for *heart-failure-predictor* were concentrated during this time interval, and almost no vital sign was processed with the *body-tempetarute-monitor* function.

Regarding the behavior of serverless functions, Figure 42 presents metrics about the execution of health services in the cloud. The chart on the left indicates the total invocations of each function. We reiterate that each health service is deployed as a serverless function, while this figure considers the sum of invocations in a time window of 5 minutes. In other words, the chart is updated every 5 minutes with the total number of invocations made in this period. Since the health service *body-temperature-monitor* finishes faster, we see that it has a higher number of invocations. We re-emphasize that no specific logic is implemented to favor this health service in the cloud, so this higher number of invocations is because this service takes less time to complete compared to *heart-failure-predictor*. The former service makes a simple *if* condition to check whether the person has fever or hypothermia, which is faster than making predictions as the second health service does. This results in more messages being ingested in 5 minutes and consumed before the messages regarding the *heart-failure-predictor* function. In turn, the *heart-failure-predictor* function is invoked fewer times every five minutes than the former function because it takes longer to complete. In summary, the number of invocations for *body-temperature-monitor* function ranges between 800 and 850 times every five minutes, while *heart-failure-predictor* is invoked around 430 times with this same time interval. The combination of both values matches the number of messages deleted from the queue previ-

ously presented in Figure 41, as messages are automatically deleted after being successfully processed by the serverless function. Finally, we see an increase in invocations of *heart-failure-predictor* after 16:00 because there were no messages regarding the *body-temperature-monitor* health service left. Therefore, fewer function replicas were busy processing this service and could process *heart-failure-predictor* instead. The chart on the right complements the details regarding serverless functions and presents how long each health service (serverless function) takes to complete when processed via AWS Lambda, considering an average interval of 5 minutes. This chart clearly shows that the *heart-failure-predictor* function takes longer to complete than the *body-temperature-monitor*. While the former usually takes around a second and a half, the latter takes around one millisecond and a half to be processed, which is almost instantly. We also see that the *body-temperature-monitor* service has no duration value close to 16:30 because all of the vital signs regarding this health service were already processed and there was nothing left regarding this health service on the queue.

Figure 42: Metrics of health services in the cloud considering a time interval of 5 minutes. The *body-temperature-monitor* service executes faster and more messages regarding this service can be processed during this time interval, while *heart-failure-predictor* takes longer to complete.

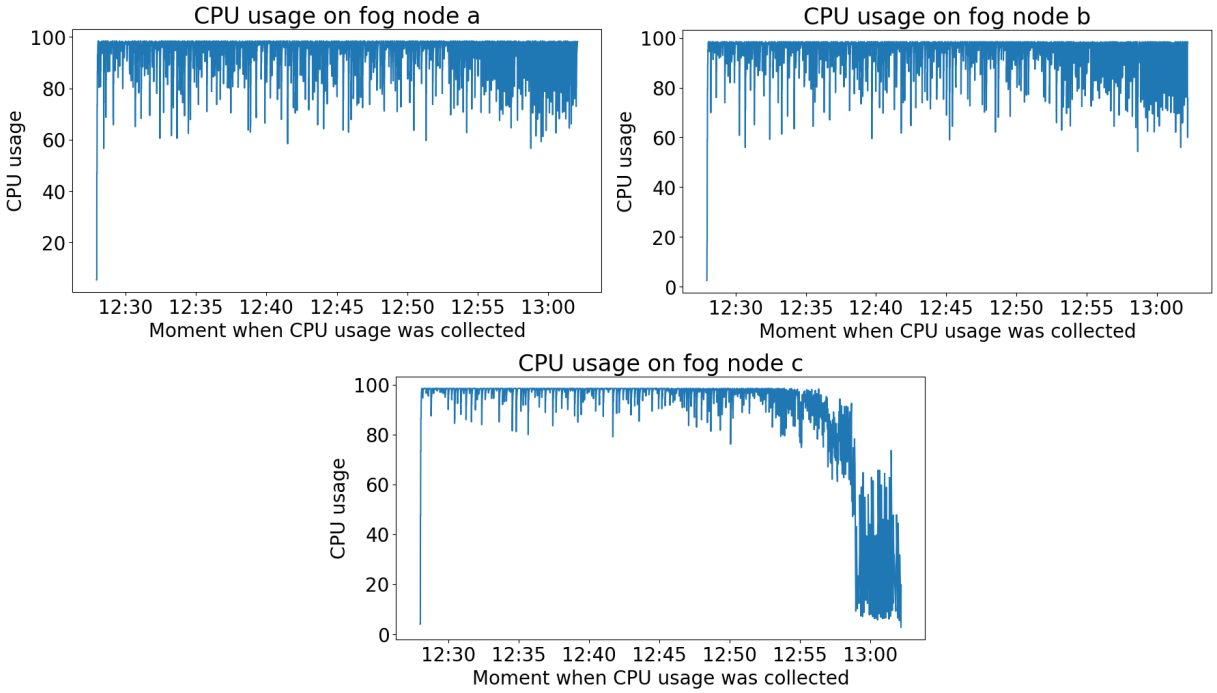


Source: prepared by the author.

Figure 43 presents the CPU usage on each fog node during this experiment, which was done with the aging technique to collect the CPU based on the last six observations. Details regarding the aging technique can be read in Subsection 5.4. We see that the three fog nodes had high CPU usage most of the time, but fog nodes *a* and *b* had eventual decreases in usage. On the other hand, fog node *c* has less subtle spikes and maintains the CPU closer to 100% most of the time. This explains why most offloading operations on fog node *c* were made because of exceeding the critical CPU threshold. The CPU usage on fog node *b* looks quite similar to fog node *a* because both receive the same number of vital signs artificially generated by edge nodes. Finally, there is a noticeable decrease in CPU usage on fog node *c* when the generation of vital signs comes to an end, which happens close to 30 minutes after the experiment starts. This occurs because the number of vital signs sent from edge to fog nodes is reduced. Therefore,

fewer offloading operations from the fog nodes *a* and *b* happen in the meantime, which causes fog node *c* to have more CPU available.

Figure 43: CPU usage on fog node when using *aging* technique. Fog nodes *a* and *b* share a similar behavior because they receive exactly the same number of vital signs from edge nodes, while fog node *c* receives vital signs offloaded from the other fog nodes.

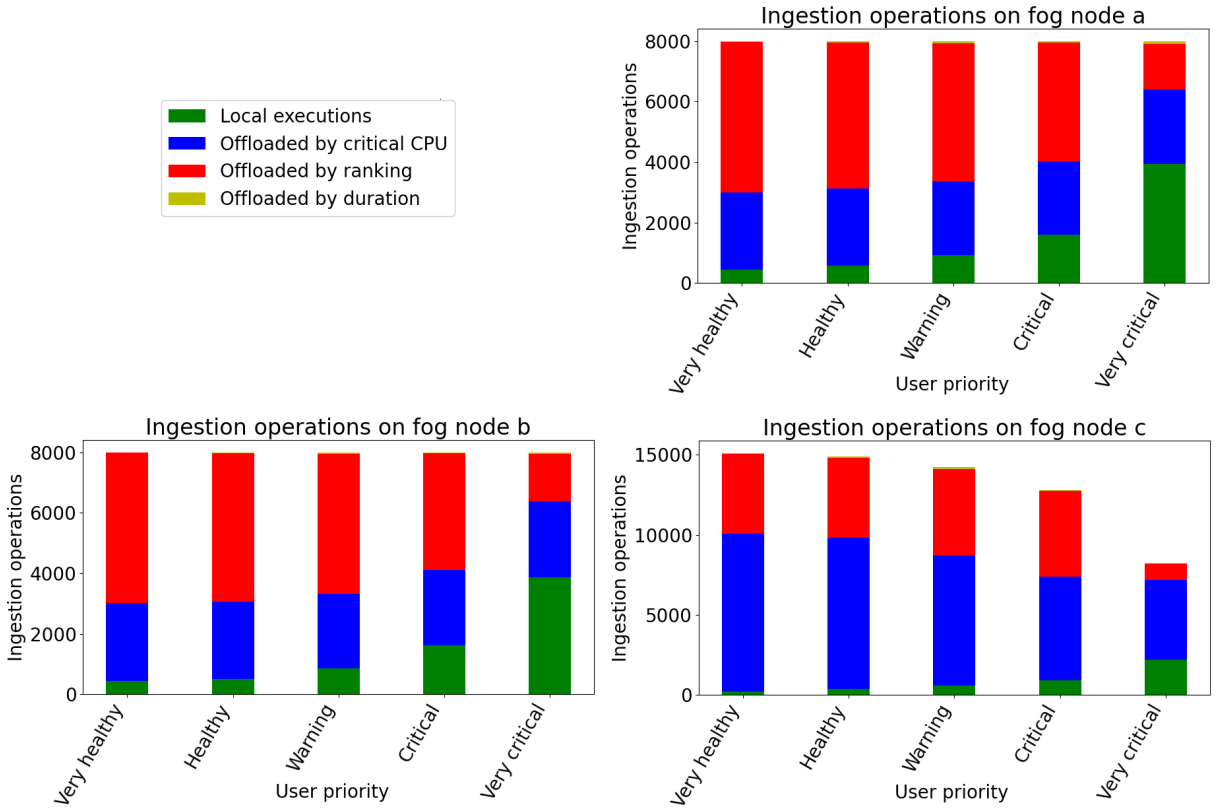


Source: prepared by the author.

Moving forward with the analysis, this paragraph gives insights into this experiment's offloading operations on the fog nodes. Many vital signs were offloaded to the cloud when arriving at fog node *c*. Still, the architecture *c* does not consider specific queues on the cloud for each user priority, which means the vital sign is added to the end of the single queue when offloaded by fog node *c*. This behavior has already been noticed in the previous experiment, but it now becomes more evident because we doubled the number of vital signs. The last experiment was done with 40.000 vital signs, and the current one was made with 80.000 vital signs. This represented a more significant number of vital signs being executed locally and also offloaded to the end of the queue on the cloud. Figure 44 combines charts regarding offloading operations and local processing of vital signs on each fog node. Starting with fog node *a*, the offloading operations behave as expected because the number of local executions increases as the user priority gets more critical. Offloading operations due to the ranking heuristic follow this same rationale and decrease as the user priority increases, as urgent vital signs represent more substantial importance in real-life scenarios. An interesting behavior is noticed in offloading operations caused by exceeding the critical CPU threshold, which in this experiment the threshold is 98%. Since the same number of vital signs is ingested for each priority, it makes sense that the to-

tal number of offloading operations caused by exceeding the critical CPU threshold is similar among user priorities. Finally, offloading operations caused by the duration heuristic are small and can hardly be seen on this chart. This leads us to rethink the importance of this heuristic, though, since it had almost no impact on the final number of offloading decisions. Future experiments may use different workloads that trigger the duration heuristic more frequently, such as concentrating the generation of vital signs for a specific user priority. A real-world use case where the duration heuristic can be effective is a neighborhood with a hospital, where there is a considerable number of people in critical conditions and, therefore, a concentration of vital signs with higher priorities. This would result in more collisions in the ranking heuristic, resulting in the duration heuristic frequently triggered.

Figure 44: Reasons for offloading operations on fog nodes. Since this experiment was done with a large number of 80.000 vital signs, fog node *c* received more vital signs offloaded from fog nodes on the lower level of the hierarchy when compared to previous experiments. In practice, fog node *c* is mostly acting as a proxy to the cloud and is not much beneficial to this experiment.



Source: prepared by the author.

The offloading behavior on fog node *b* can also be seen in Figure 44 and is similar to fog node *a*. This is expected, as both fog nodes have the same available computing resources and receive the same number of vital signs from edge nodes. On the other hand, the offloading behavior on fog node *c* is quite different from the two other fog nodes since it receives vital signs from fog nodes *a* and *b* at the same time. This explains why the total offloading operations is

higher on fog node *c*, but the local processing of vital signs is minimal, which is not beneficial to the overall throughput of the experiment. This fog node is primarily working as a simple proxy to the cloud because it is almost always overloaded with such a high number of incoming vital signs. Even though the number of vital signs executed locally is more significant for *very critical* user priority, it is still lower when compared to the total number of vital signs received. This means that fog node *c* was not very beneficial to this experiment at all, as many vital signs could have been directly offloaded to the cloud by fog nodes *a* and *b*. This leads us to conclude that fog nodes located at higher levels of the hierarchy should have more available computing resources, such as more CPU cores, to deal with the higher number of vital signs sent by fog nodes on the lower level of the hierarchy. The current experiment was done with three virtual machines having the same number of available computing resources, regardless of the level where the fog node is located in the hierarchy. Additionally, since the duration heuristic was not very effective in this scenario, Table 14 presents the exact numbers of each offloading reason to make this information more evident. These numbers are the same used to plot charts for Figure 44 and help understand the real impact of these operations. Also, this table exhibits information about all fog nodes. We see most local executions being made on fog nodes *a* and *b*, while fog node *c* had a high number of vital signs being offloaded because of exceeding the critical CPU threshold. Also, fog node *c* offloaded thousands of vital signs because of the ranking heuristic.

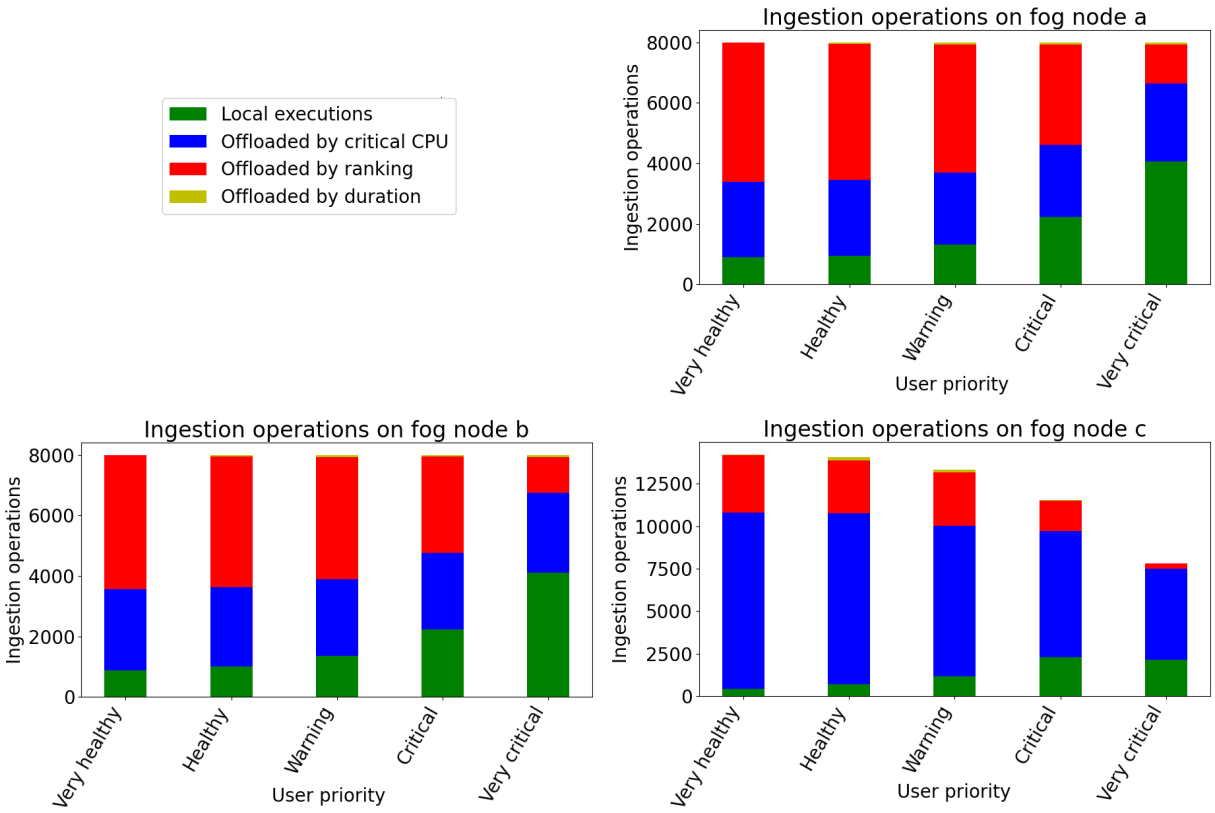
Table 14: Offloading reasons for each user priority on each fog node.

Name	Priority	Local exec.	Critical CPU	Ranking heuristic	Duration heuristic
Fog node A	1	451	2.554	4.968	27
	2	583	2.541	4.830	46
	3	919	2.445	4567	69
	4	1.589	2.428	3.935	48
	5	3.928	2.471	1.510	91
Fog node B	1	439	2.577	4.975	9
	2	509	2.542	4.912	37
	3	847	2.484	4607	62
	4	1.615	2.487	3.868	30
	5	3.857	2.519	1.573	51
Fog node C	1	230	9.820	5.016	44
	2	386	9.452	4.961	109
	3	570	8.155	5.363	146
	4	899	6.506	5.352	39
	5	2.181	4.985	1.026	23

Variation #2 - Higher warning threshold: this variation of the experiment is similar to the previous variation, except for the warning threshold of CPU to trigger the offloading heuristics. This threshold was increased from 60% to 80% because the previous variation had many vital signs being offloaded to fog node *c* and to the cloud. Most of the time, fog node *c* was a simple

proxy to the cloud and processed almost no vital signs locally. Increasing the warning threshold is an attempt to process more vital signs locally on fog nodes. For simplicity purposes, we do not provide a deep analysis of each component involved in the architecture for this variation of the experiment because some of them are similar to the previous variation, such as the queue on the cloud. The values for CPU usage are also omitted from this specific analysis for the same reason. The analysis made on this variation focuses on elements that are directly affected by the increase in the warning threshold, which in this case are the fog nodes.

Figure 45: Ingestion of vital signs on the fog nodes considering a warning threshold of 80% and a critical threshold of 98%.



Source: prepared by the author.

Figure 45 combines information about the ingestion of vital signs on the three fog nodes. Starting with fog node *a*, we see the overall number of local executions increasing when compared to variation #1 of this experiment. This is beneficial because we aim to process more vital signs locally, as extra network latency is involved when offloading vital signs. Also, when processing vital signs locally, the processing is done synchronously and there is no need for messages to wait in a queue. This is especially important for *very critical* user priority because it represents a person in an urgent condition that needs instant assistance. Compared to the previous variation, there is an increase in vital signs being processed locally, especially for *very healthy* user priority. There were 451 local executions in fog node *a* in the last experiment

for *very healthy* user priority, but with the increased warning threshold, this value increases to close to 1.000. This almost doubles the number of vital signs processed locally for *very healthy* user priority. Regarding the *very critical* user priority, there is no such difference at all. Fog node *b* has a similar behavior when compared to fog node *a* because both receive the same number of vital signs artificially generated by edge nodes. Regarding fog node *c*, the number of local executions increased, especially regarding user priorities ranging from *very healthy* to *critical*. There was no such significant difference for *very critical* user priority, though, but this is the most relevant user priority and should have a higher number of local executions. Most offloading operations for *very critical* user priority continue happening because of exceeding the critical CPU threshold. Even though the critical threshold is 98%, which is already a high margin to consider offloading operations, future experiments could also consider increasing the critical threshold to 100% to understand how the architecture behaves. This means the offloading operations would only happen when the CPU is constantly used. Until the CPU reaches this value of 100%, the ranking and duration heuristics would do the hard work of making offloading decisions. This depends on each scenario, though, such as how much CPU a given health service requires and how long they take to complete. If the health service consumes a lot of CPU and takes several seconds to complete, running several instances of the health services locally may cause the machine to become overwhelmed when setting the CPU threshold to 100%. However, if health services are less heavyweight, it sounds reasonable to increase the critical CPU threshold to 100% or something close to it. In conclusion, no specific threshold would fit all scenarios and depends on the health services being executed.

6.6 Discussion - Achievements and limitations

This section discusses the results previously presented in this chapter, as well as emphasizes the achievements of the proposed model and indicates limitations that can be addressed in future work. Different architectures were evaluated during the experiments. We started with a naive approach where serverless functions are spawned on the cloud provider whenever a vital sign is offloaded, followed by more robust architectures that consider asynchronous processing on the cloud to deal with high usage peaks. The reason why multiple architectures were considered in the evaluation, instead of only evaluating the architecture proposed in Chapter 4 of this thesis, is because the final version of SmartVSO is a result of iterative improvements during the evaluation phase. For example, we originally proposed architecture *A* as a solution to ingest vital signs for this thesis, but we faced throttling problems that we will discuss in the incoming paragraphs. We evolved the architecture until it could satisfy our scalability and reliability requirements, as well as we identified situations where SmartVSO performed well and situations where it did not, which can be improved in future work and will be presented as follows.

Regarding situations where SmartVSO performed well, we see the ranking heuristic as a really promising approach because it easily combines service priority with user priority into a

single numerical value. This is a simple calculation that does not require much computational effort and still generates exciting results. We could identify percentiles indicating vital signs processed almost instantly in the fog (within a few seconds), especially regarding the *very critical* user priority, as people in urgent situations should have their vital signs monitored as soon as possible. We also noticed that the number of vital signs processed locally increases as the user priority also increases, in a way that non-critical vital signs take longer to be consumed in favor of critical vital signs. Taking the variation #1 of scenario 5 as an example, 60% of vital signs regarding *very critical* user priority are processed in less than approximately 5 seconds, while for the other user priorities, it takes between 6.873 (114 minutes) and 7.678 (127 minutes) to process this same percentage of vital signs. This clearly shows the effectiveness of the ranking heuristic. In addition, we see two main strategies to process vital signs faster, which is especially useful when dealing with a higher concentration of people in smart cities: *i)* including additional fog nodes in the hierarchy (city neighborhoods) to process vital signs without the extra network latency of sending vital signs to the cloud, or *ii)* increasing the number of function replicas quota on the cloud provider, so the queue can be consumed faster. Regarding the duration heuristic, it is useful when running experiments with multiple health services because it offloaded vital signs when being processed with slower services, but we expected it to be more effective. This heuristic did not result in a significant number of offloading operations. Also, forecasting the duration to make the offloading decision with this heuristic imposes extra overhead, while it could be faster to simply process the vital sign locally without triggering the duration heuristic itself. This is especially true for health services such as *body-temperature-monitor* that complete within a few milliseconds. Future work can consider the time to execute the duration heuristic as a variable in the decision between triggering it or not.

Regarding situations where SmartVSO did not perform well, a major limiting factor we see in the experiments is that no strategy is employed to favor urgent vital signs when consuming the queue on the cloud. All vital signs offloaded by the last fog node in the hierarchy are added to the end of a single, shared queue, including vital signs regarding people in urgent situations. The queue itself solves the problem of scalability, allowing it to work with huge and unexpected workloads, but a strategy should favor the consumption of critical vital signs. This directly impacts the response time percentiles because it will take longer to process vital signs when the queue is large, so only small percentiles will have short response times, which is not the desired behavior. One approach would be to have five different queues, one for each user priority, and only process less critical vital signs when the queue regarding more critical vital signs is empty. There are challenges, though, such as the problem of starvation. Non-critical vital signs may become stale and take hours to be consumed if critical vital signs are continuously added to the queue. There is not much sense in sending a notification to a healthy person that he or she has a fever, hours after the person has a fever, for example. Future work should combine strategies to consider a maximum time limit to process vital signs along with the ability to favor urgent ones from the queue in the cloud.

Finally, we learned from experiments with architecture *A* that cloud providers may limit the number of serverless function replicas. In other words, there is a limit of how many serverless functions can be executed simultaneously. Our experiments with architecture *A* spawned functions on the cloud in a synchronous manner, in a way that whenever a vital sign was offloaded to the cloud, a function was spawned to consume this vital sign. This leads to throttling errors, as explained in Subsection 6.2. The solution we employ to prevent throttling errors in SmartVSO is to use a queue in the cloud, which is used on architectures *B* and *C*. With this approach, vital signs offloaded to the cloud are first stored in a queue, instead of the fog nodes directly spawning serverless functions in the cloud. The queue can even grow much faster than the number of function replicas available to process the vital signs, and even in this situation, no vital sign will be discarded or will be lost. This greatly contributes to the resilience and stability of the proposed model. Another possible approach to deal with throttling problems is to continue with the original approach of architecture *A*, but handle throttling errors on the fog nodes by inserting these failed requests into a local queue to retry the execution later. However, we see this as a complex solution because the fog node may not have enough memory or storage to include these pending vital signs. Given the mentioned pros and cons, we decided to move further with the approach of always inserting vital signs into a queue in the cloud when performing offloading from the last fog node on the hierarchy.

6.7 Partial considerations

This chapter presented the results of experiments made with the prototype of SmartVSO. We presented exciting results that indicate this model's potential to reduce response time for vital signs regarding people in critical conditions. The final version of SmartVSO, as presented in this thesis, results from an iterative process of implementing, testing, evaluating the results, and improving the model to deal with a higher number of vital signs. We see that fog nodes help process vital signs because of the following reasons: *i)* they contribute to reducing the response time when processing vital signs with health services, especially for critical user priorities, *ii)* cloud providers have specific regions where resources are available (such as São Paulo), therefore being unavailable in the other areas of Brazil, such as Recife. We see the usage of a queue in the cloud as a solution to deal with high usage peaks because the quotas for serverless function replicas may eventually exceed and vital signs be lost because of Throttling problems. The proposed solution combines the synchronous and agile processing of vital signs in the fog nodes with the support for high usage peaks on the cloud, by leveraging a queue where vital signs are stored and are consumed asynchronously at a speed the lambda replicas can process.

We performed experiments considering multiple CPU thresholds, multiple numbers of vital signs, and different strategies to collect CPU usage. These results are exciting because they have a social contribution that helps people in smart cities have a higher quality of life. We analyzed several metrics when running the experiments, such as the number of offloading operations on

each fog node, the reason why each offloading operation was made, the CPU usage during the experiment, the duration of running health services in the cloud, and metrics regarding the storage and consumption of messages in the queue. Experiments considered a single queue in the cloud. Also, these experiments conclude that no specific CPU threshold fits all scenarios and we suggest dynamic calibration of these values as future work, as well as employing strategies to favor the processing of critical vital signs in the cloud since all vital signs are typically consumed in the order they arrive, regardless of the user priority and the calculated ranking.

7 CONCLUSION

Healthcare applications are of paramount importance in smart cities, as they improve the health quality of the population by running services with vital signs collected from smartwatches. These services can detect problems in a proactive manner and perform intelligent actions based on the type of the problem, such as sending a notification to the person's smartphone or automatically calling an ambulance when a critical situation is detected. With this in mind, this thesis has proposed SmartVSO - a model of an architecture that hierarchically integrates fog computing and cloud computing, in a way that critical vital signs are favored when computing resources are getting overloaded in the fog. We aim to achieve short response times when running health services with critical vital signs. In other words, people with health problems will continue receiving notifications as soon as possible, even when machines are experiencing high usage peaks. An example of a high usage peak would be people concentrating in specific neighborhoods because of a social event, with smartwatches sending vital signs to the same fog node.

Section 1.2 presented the following research question: *how could be a computational model for running health services with short response time and high quality of experience by considering user and service priorities and dealing with high usage peaks in a transparent manner?* To answer this question, the proposed SmartVSO model connects fog nodes in the form of a tree to process vital signs with short response time, so each fog node is connected to a single parent. The last fog node in the hierarchy is connected to the cloud, which benefits from virtually infinite computing resources, but results in higher response times because of the long physical distance. We introduce a recursive offloading strategy based on a ranking calculated for each vital sign that combines the user priority and the health service priority into a single numerical value. With this recursive strategy in mind, incoming non-critical vital signs are offloaded to the parent fog node when computing resources on the current node are getting overloaded. In turn, vital signs are offloaded again if the parent node is also overloaded, but this time are offloaded to the third level of the hierarchy. The process repeats until the vital sign is finally processed by an available fog node, or is added to a queue on the cloud when all fog nodes on the path are overloaded. Cloud is only employed as a last option, especially for non-critical vital signs, as it results in higher response time because of the long physical distance. Finally, the higher the calculated ranking for a vital sign, the smaller the chances of it being offloaded, as each offloading operation imposes extra network latency and increases the response time. The proposed heuristics favor critical vital signs whenever possible, by reducing the number of offloading operations for them and offloading non-critical vital signs instead. This architecture is promising because it considers response time and scalability techniques, which are very important in the context of a smart city. We hope this contribution will help advance this field of research for ubiquitous access to health services in smart cities.

7.1 Contributions

This thesis has the following main scientific contributions:

- Priority-oriented heuristic to minimize response time when processing health services with critical vital signs, which favors people with health problems by combining user priority and service priority into a single numerical value that we name *ranking*;
- Recursive strategy to offload vital signs in a hierarchic fog-cloud architecture with the shape of a tree, while vital signs are only processed in the cloud when no fog node in the path from the leaf to the root has available computing resources to process them.

7.2 Limitations and future work

This thesis has some limitations that can be addressed in future work. To leverage this mechanism to real-world usage, where health services can be implemented by authorized third parties as serverless functions, additional security concerns need to be addressed. Examples include not allowing the health service to invoke other functions on the serverless platform, that are intended for internal usage only, or using an exorbitant and unlimited amount of computing resources that could cause instability to the fog node. The following items summarize the topics that are out of scope for the current work and we suggest further investigation in future work:

- Consider priorities when consuming vital signs from the queue in the cloud, so critical vital signs can also be favored on this layer. This thesis only considers the calculated ranking during offloading operations in the fog. However, high usage spikes may also offload critical vital signs to the cloud, when fog is completely overloaded, but vital signs are processed in the order they arrive in the queue in the cloud regardless of the priority;
- Allow fog nodes to communicate with siblings or multiple parents, instead of always offloading vital signs to the same pre-configured parent fog node. This would minimize problems caused by a lack of connectivity to the parent node, as a single communication path represents a single point of failure. Multiple communication paths would allow alternative routes for the vital sign to navigate, which also allows choosing a route that minimizes the response time with fewer hops between nodes. Algorithms such as Dijkstra's shortest path may be employed to this end;
- Consider additional information besides the percentage of used CPU, such as memory, disk, and network, among others. Some health services may be I/O bound and others may be CPU bound, as well as complex health services that employ machine learning techniques may use a larger amount of memory. It is important to consider additional variables on the offloading decisions, or even specific thresholds for each health service;

- Employ strategies to automatically send low-priority vital signs to the cloud instead of giving this responsibility only to the last fog node. Nodes at lower levels of the hierarchy could know beforehand that all of the other nodes are overloaded, and therefore offload vital signs directly to the cloud to reduce the number of hops between fog nodes;

- Propose algorithms to dynamically discover appropriate values for Warning and Critical offloading thresholds, instead of being static and needing to be manually defined beforehand. Our results indicate that no single threshold fits all scenarios, so additional heuristics could calibrate these values in real time, without needing to redeploy the modules.

7.3 Scientific publications

Serverless computing and computational offloading have been some of the main research topics for this Masters since the beginning of 2021. In this context, we published a scientific paper and a conference paper, as follows:

- CASSEL, G. A. S. et al. Serverless computing for internet of things: a systematic literature review. Future Generation Computer Systems, [S.l.], v. 128, p. 299–316, 2022;

- CASSEL, G.; RIGHI, R.; RODRIGUES, V. A novel fog-cloud architecture to process serverless functions with adaptive timeout. In: XXII ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIãO SUL, 2022, Porto Alegre, RS, Brasil. Anais. . . SBC, 2022. p. 81–82.

The first is named *Serverless computing for internet of things: a systematic literature review*, which presents a Systematic Literature Review presenting recent papers that deal with this technology in the field of IoT. This serves as the basis for this thesis because it is another use case for using serverless computing with IoT, more specifically in the field of health. After the selection process, we reviewed 60 papers and identified several characteristics, such as challenges and components to incubate serverless functions, protocols, and programming languages. We also investigated how these papers address offloading techniques with serverless computing, which is directly related to this thesis. In our context, we employ vertical offloading to forward vital signs to the parent fog node (or the cloud) when the local fog node is overloaded. This survey was an inspiration for decisions on this thesis (CASSEL et al., 2022).

We also published a conference paper on ERAD 2022, named *A novel fog-cloud architecture to process serverless functions with adaptive timeout*. This conference document presents an initial architecture that evolved in several aspects until it became this thesis. We have modified several ideas so far, but the original computational offloading idea remains the same. In summary, this paper presents an architecture that employs fog and cloud computing and would benefit from the dynamic calculation of timeout constraints for serverless functions. This would prevent health services from exceeding timeout constraints and prevent them from re-executing

deliberately. However, we do not employ this idea in this thesis but suggest improvements for adaptive timeout as future work instead (CASSEL; RIGHI; RODRIGUES, 2022).

REFERENCES

ALZAILAA, A. et al. Low-latency task classification and scheduling in fog/cloud based critical e-health applications. In: ICC 2021 - IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS, 2021. **Anais...** [S.l.: s.n.], 2021. p. 1–6.

ARORA, U.; SINGH, N. Iot application modules placement in heterogeneous fog–cloud infrastructure. **International Journal of Information Technology**, [S.l.], v. 13, n. 5, p. 1975–1982, Oct 2021.

AWS, A. **Identifying and managing throttling**. 2022.

AWS, A. **Available cloudwatch metrics for amazon sqs**. 2022.

BALDINI, I. et al. Serverless computing: current trends and open problems. In: **Research advances in cloud computing**. Singapore: Springer Singapore, 2017. p. 1–20.

BERMBACH, D. et al. Towards auction-based function placement in serverless fog platforms. In: IEEE INTERNATIONAL CONFERENCE ON FOG COMPUTING (ICFC), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p. 25–31.

BUKHARI, M. M. et al. An intelligent proposed model for task offloading in fog-cloud collaboration using logistics regression. **Computational Intelligence and Neuroscience**, [S.l.], v. 2022, p. 3606068, Jan 2022.

BUYYA, R.; SRIRAMA, S. N. Internet of things (iot) and new computing paradigms. In: **Fog and edge computing**: principles and paradigms. [S.l.]: Wiley, 2019. p. 1–23.

CASSEL, G. A. S. et al. Serverless computing for internet of things: a systematic literature review. **Future Generation Computer Systems**, [S.l.], v. 128, p. 299–316, 2022.

CASSEL, G.; RIGHI, R.; RODRIGUES, V. A novel fog-cloud architecture to process serverless functions with adaptive timeout. In: XXII ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIãO SUL, 2022, Porto Alegre, RS, Brasil. **Anais...** SBC, 2022. p. 81–82.

CHEN, S.; WANG, L.; LIU, F. Optimal admission control mechanism design for time-sensitive services in edge computing. In: IEEE INFOCOM 2022 - IEEE CONFERENCE ON COMPUTER COMMUNICATIONS, 2022. **Anais...** [S.l.: s.n.], 2022. p. 1169–1178.

CHENG, B. et al. Fog function: serverless fog computing for data intensive iot services. In: IEEE INTERNATIONAL CONFERENCE ON SERVICES COMPUTING (SCC), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p. 28–35.

CHOWHAN, K. **Hands-on serverless computing**: build, run, and orchestrate serverless applications using aws lambda, microsoft azure functions, and google cloud functions. [S.l.: s.n.], 2018.

CICCONETTI, C.; CONTI, M.; PASSARELLA, A. A decentralized framework for serverless edge computing in the internet of things. **IEEE Transactions on Network and Service Management**, [S.l.], v. 18, n. 2, p. 2166–2180, June 2021.

DEHURY, C. K. et al. Def-drel: systematic deployment of serverless functions in fog and cloud environments using deep reinforcement learning. **CoRR**, [S.l.], v. abs/2110.15702, 2021.

GASMI, K. et al. A survey on computation offloading and service placement in fog computing-based iot. **The Journal of Supercomputing**, [S.l.], v. 78, n. 2, p. 1983–2014, Feb 2022.

GEORGE, G. et al. Nanolambda: implementing functions as a service at all resource scales for the internet of things. In: IEEE/ACM SYMPOSIUM ON EDGE COMPUTING (SEC), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p. 220–231.

GUPTA, H. et al. ifogsim: A toolkit for modeling and simulation of resource management techniques in internet of things, edge and fog computing environments. **CoRR**, [S.l.], v. abs/1606.02007, 2016.

HABIBI, P. et al. Fog computing: a comprehensive architectural survey. **IEEE Access**, [S.l.], v. 8, p. 69105–69133, 2020.

HAGHI KASHANI, M. et al. A systematic review of iot in healthcare: applications, techniques, and trends. **Journal of Network and Computer Applications**, [S.l.], v. 192, p. 103164, 2021.

HARTMANN, M.; HASHMI, U. S.; IMRAN, A. Edge computing in smart health care systems: review, challenges, and research directions. **Transactions on Emerging Telecommunications Technologies**, [S.l.], v. 33, n. 3, p. e3710, 2022. e3710 ett.3710.

HASSAN, H. O.; AZIZI, S.; SHOJAFAR, M. Priority, network and energy-aware placement of iot-based application services in fog-cloud environments. **IET Communications**, [S.l.], v. 14, n. 13, p. 2117–2129, 2020.

HYNDMAN, R.; ATHANASOPOULOS, G. **Forecasting**: principles and practice. 3rd. ed. Australia: OTexts, 2021.

JAMIL, B. et al. Irats: a drl-based intelligent priority and deadline-aware online resource allocation and task scheduling algorithm in a vehicular fog network. **Ad Hoc Networks**, [S.l.], v. 141, p. 103090, 2023.

KATZER, J. **Learning serverless**: design, develop, and deploy with confidence. [S.l.]: O'Reilly Media, Incorporated, 2020.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for performing systematic literature reviews in software engineering**. 2007.

KRATZKE, N. A brief history of cloud application architectures. **Applied Sciences**, [S.l.], v. 8, n. 8, 2018.

NATH, S. B. et al. Containerized deployment of micro-services in fog devices: a reinforcement learning-based approach. **The Journal of Supercomputing**, [S.l.], v. 78, n. 5, p. 6817–6845, Apr 2022.

PAREEK, K.; TIWARI, P. K.; BHATNAGAR, V. Fog computing in healthcare: a review. **IOP Conference Series: Materials Science and Engineering**, [S.l.], v. 1099, n. 1, p. 012025, mar 2021.

PELLE, I. et al. Latency-sensitive edge/cloud serverless dynamic deployment over telemetry-based packet-optical network. **IEEE Journal on Selected Areas in Communications**, [S.l.], v. 39, n. 9, p. 2849–2863, Sep. 2021.

PINTO, D.; DIAS, J. P.; SERENO FERREIRA, H. Dynamic allocation of serverless functions in iot environments. In: IEEE 16TH INTERNATIONAL CONFERENCE ON EMBEDDED AND UBIQUITOUS COMPUTING (EUC), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p. 1–8.

RAUSCH, T.; RASHED, A.; DUSTDAR, S. Optimized container scheduling for data-intensive serverless edge computing. **Future Generation Computer Systems**, [S.l.], v. 114, p. 259–271, 2021.

REZAZADEH, Z.; REZAEI, M.; NICKRAY, M. Lamp: a hybrid fog-cloud latency-aware module placement algorithm for iot applications. In: CONFERENCE ON KNOWLEDGE BASED ENGINEERING AND INNOVATION (KBEI), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p. 845–850.

RIGHI, R. d. R. et al. Autoelastic: automatic resource elasticity for high performance applications in the cloud. **IEEE Transactions on Cloud Computing**, [S.l.], v. 4, n. 1, p. 6–19, Jan 2016.

RODRIGUES, V.; RIGHI, R. Minhahistoriadigital: an scalable fog-based architecture for efficient vital signs monitoring over smart cities. In: XXII ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIãO SUL, 2022, Porto Alegre, RS, Brasil. **Anais...** SBC, 2022. p. 117–118.

SANCHEZ-GALLEGOS, D. D. et al. Puzzlemesh: a puzzle model to build mesh of agnostic services for edge-fog-cloud. **IEEE Transactions on Services Computing**, [S.l.], p. 1–1, 2022.

SANTOS, G. L.; MONTEIRO, K. H. d. C.; ENDO, P. T. Living at the edge? optimizing availability in iot. In: LYNN, T. et al. (Ed.). **The cloud-to-thing continuum**: opportunities and challenges in cloud, fog and edge computing. Cham: Springer International Publishing, 2020. p. 79–94.

SBARSKI, P. **Serverless architectures on aws**: with examples using aws lambda. [S.l.]: Manning, 2017.

SHEIKH SOFLA, M. et al. Towards effective offloading mechanisms in fog computing. **Multimedia tools and applications**, [S.l.], p. 1–46, Oct 2021. 34690529[pmid].

SIMON JUNIOR, H. et al. Pediatric emergency triage systems. **Rev Paul Pediatr**, Brazil, v. 41, p. e2021038, July 2022.

TANENBAUM, A.; WETHERALL, D. **Computer networks**. [S.l.]: Pearson Prentice Hall, 2011.

WANG, F. et al. Fault tolerating multi-tenant service-based systems with dynamic quality. **Knowledge-Based Systems**, [S.l.], v. 195, p. 105715, 2020.

YANG, Y. et al. A review of iot-enabled mobile healthcare: technologies, challenges, and future trends. **IEEE Internet of Things Journal**, [S.l.], v. 9, n. 12, p. 9478–9502, June 2022.

ZHANG, G.; NAVIMIPOUR, N. J. A comprehensive and systematic review of the iot-based medical management systems: applications, techniques, trends and open issues. **Sustainable Cities and Society**, [S.l.], v. 82, p. 103914, 2022.

ZHAO, P. et al. Selfish-aware and learning-aided computation offloading for edge-cloud collaboration network. **IEEE Internet of Things Journal**, [S.l.], p. 1–1, 2023.