DIOGO JOÃO DE PAIVA GOMES

Bachelor in Computer Science

DECOMPOSING MONOLITIC COMPUTATIONS FOR EXECUTION ON THE EDGE FOR FUN AND PROFIT

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

DEPARTMENT OF
COMPUTER SCIENCE

DECOMPOSING MONOLITIC COMPUTATIONS FOR EXECUTION ON THE EDGE FOR FUN AND PROFIT

DIOGO JOÃO DE PAIVA GOMES

Bachelor in Computer Science

Adviser: João Leitão
Associate Professor, NOVA University Lisbon

Examination Committee:

Chair: Doutor Nuno Manuel Robalo Correia
Full Professor, FCT-NOVA

Rapporteur: Doutor João Nuno de Oliveira e Silva
Assistant Professor, Instituto Superior Técnico

Adviser: Doutor João Carlos Antunes Leitão
Associate Professor, FCT-NOVA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

Decomposing monolitic computations for execution on the Edge for fun and profit

To my family and friends.

Acknowledgements

I am deeply grateful to my supervisor João Leitão for his invaluable guidance and support throughout the development of my Thesis. His extensive knowledge, dedication, and patience were crucial in helping me achieve my research goals. He always made himself available to answer my questions and provide valuable feedback, which helped me to improve the quality of my work significantly. I am truly fortunate to have had the opportunity to work under his guidance, and I will always be indebted to him for his contributions to my academic growth.

I am grateful to the Department of Informatics and the teachers for providing me with a high-quality education, conducive learning environment, and valuable insights and feedback. Thank you for your dedication and commitment to helping students achieve their academic goals.

I would also like to express my heartfelt gratitude to my parents and girlfriend for their unwavering support throughout my academic journey. Their love, encouragement, and sacrifices have been a constant source of inspiration and motivation for me. Their emotional and financial support have made it possible for me to pursue my academic aspirations and achieve my goals. I am truly blessed to have them in my life, and I will always be grateful for their unwavering support and love. Thank you, Mom, Dad, and my loving girlfriend, for being my pillars of strength and supporting me every step of the way.

"Our greatest weakness lies in giving up." (Thomas A. Edison)

Abstract

Cloud computing is emerging in the IT industry and is being established as the standard way of providing high quality applications and services, being able to scale when necessary and maintaining low latency. This high availability allows new users and devices to connect which may possibly cause some saturation in the network.

The edge computing paradigm has recently emerged to address raising concerns on the ability of cloud computing infrastructures (and data links connecting end user devices to such infrastructures) to receive, process, and reply to operations generated by edge devices (e.g., sensors or other more sophisticated devices) in an expedited way. One way to mitigate this challenge, put forward by the edge computing paradigm, is to decompose the computations executed by applications in smaller components that can then be pushed into the devices (either on the cloud or edge) that are in close vicinity to the location of data generation and consumption. This allows to remove pressure from network links (by avoiding to move data) and can, in particular cases, speedup the production of results to be consumed in the edge. However, achieving this intuitive vision is cumbered by several challenges, in particular:

1. How to decompose the logic of a (large) computation into smaller units;
2. Understanding the requirements for the correct processing of such smaller units;
3. Define mechanisms to compose the results of the smaller computations;
4. Effectively pushing the computations to the locations where they will be conducted and ensuring their execution despite failures.

In this thesis we will start to pave the way towards such a vision by studying how to decompose computations, classifying relevant properties of a subset of classes of these computations, and exploring the design space of the runtime support to execute smaller units in an integrated way across the cloud and edge. We will focus on settings with multiple cloud datacenters potentially with a few edge locations.

Keywords: Cloud, Edge, IoT, Computation

Resumo

A computação em Cloud (na Nuvem) está a crescer na indústria de Tecnologias de Informação e está a ser estabelecida como o padrão para fornecer aplicações e serviços de qualidade, sendo capaz de serem escalados quando necessário, mantendo sempre uma rápida resposta. Esta grande disponibilidade permite que novos utilizadores e dispositivos se conectem de uma forma exponencial, causando saturação na rede.

O paradigma da computação em Edge (na fronteira) começou a emergir para solucionar o problema da disponibilidade da infraestrutura de uma arquitetura em Cloud para receber e processar informação e operações recebidas por dispositivos de Edge (sensores ou outros dispositivos mais sofisticados) de uma forma rápida e eficiente. Uma forma de mitigar este problema, com ajuda da computação em Edge, passa por dividir as computações que são executadas pelas aplicações em blocos de menor dimensão sendo depois processados perto da localização onde a data é criada ou consumida. Este fator permite aliviar e remover pressão das conexões entre vários dispositivos na rede, evitando transferir dados e por sua vez aumentar a velocidade de saída de resultados dos dados que são consumidos.

Apesar de tudo, alcançar esta visão traz alguns desafios:

1. Como decompor a lógica de grandes computações em unidades mais pequenas;
2. Entender os requisitos para um correto processamento de cada unidade;
3. Definir mecanismos que possam compor os resultados das sucessivas unidades;
4. Passar efetivamente as computações para uma localização mais ideal onde serão tratadas e assegurar a sua execução apesar da possibilidade de falhas.

Nesta dissertação iremos começar a traçar o caminho para esta visão, estudando como se deve decompor computações, classificando as suas propriedades e explorar as possibilidades para a execução de unidades mais pequenas de uma forma integrada. Iremos focar em cenários com vários datacenters em cloud e vários dispositivos de Edge.

Palavras-chave: Cloud, Edge, IoT, Computação

Contents

List of Figures

1 Introduction

1.1 Context

Nowadays, everyone is used to carrying a smart device in their pocket throughout the day. These devices are used for checking the news, communicating with friends, playing games, or even, as a work essential tool. Our smartphones are an example of a device that is constantly posting and requesting information from a remote location (not on their local storage) and this was made possible by the appearance of Cloud Computing [44], an almost infinite scalable infrastructure that allows users and companies to use resources and services in the palm of their hands, without having the need for big devices with very computational power.

Although Cloud computing seems to be the way to go for our daily usage, it will not be a viable solution to build fast and efficient systems in the near future. The increase of Internet of Things (IoT) [49] applications and mobile devices is leading to a fast increase in the amount of data created and transmitted, which is consequently increasing the time required for the cloud data centers to process new requests. Another problem that is faced by the cloud computing paradigm is that we may have some situations where the amount of data produced at a certain time is bigger than the capacity of the transmission itself, as the network capacity is not evolving at the same rhythm as the need of more edge centric [29] applications.

Edge computing [29] is a new concept that is emerging as a way to mitigate the limitations described above, as it is considered a way to perform computations outside of the cloud data centers and closer to the devices or end-users. This is a very broad concept, as the edge can be considered as being composed of any device with the ability to communicate between the source of the data and the cloud and having the ability to perform data computations (analyzing, filtering, flagging, etc). These devices might include sensors, actuators and other endpoints, as well as IoT (Internet of Things) servers and routers.

By moving some computations from the cloud to the edge, especially when dealing with large amounts of data, it is possible to lower the network traffic and decrease the latency for the end-user. It can also bring some security advantages as the edge can be managed by the organization itself (and not the cloud providers), and certain security policies can be applied.

1.2 Motivation

Current systems and applications that use the Cloud for storage and computation will always have its benefits but there are also ways to improve overall latency and user experience by taking advantage of edge resources.

As mentioned above, clients that are far away from the Cloud datacenter location will experience a larger latency than if they were closer to the datacenter. Cloud computing tries to minimize these issues by replicating everything in multiple locations around the world, hoping to have them closer to each user or client.

With the increasing amount of data being produced and transmitted, mostly by the emergence of IoT (Internet of Things) devices, the bandwidth of the connections between the data points and the Cloud will eventually become saturated [51]. This issue can also affect user applications and games where the user needs to wait for a response to their request to the Cloud datacenter.

With the use of Edge computing, applications can avoid unnecessary communications with the Cloud, by communicating with an intermediate node [47], executing most of the computations there. As this is mainly still a concept, there is a need to improve this kind of architectures as their current design also presents some flaws and challenges that must be addressed. We want to help developers and organizations during the development and implementation of new edge applications that are constantly receiving and processing data from multiple sources by simplifying their process.

1.3 Objective

In this thesis we have analysed how cloud computing can affect the latency of Internet of Things (IoT) devices and the data they produce. We set Edge computing as a possible solution for this problem, by improving latency and costs.

We also studied different ways to approach computation processes and how we can split monolithic tasks into smaller ones, checking if it is more efficient than the traditional way. There is also a need to improve the way that edge applications use their resources. With this in mind, we have developed a framework with the following goals:

- Help developers and organizations to create or improve edge applications by taking advantage of all the available resources with less human intervention.

- Provide mechanisms for declaring and introducing new devices at any time.

- Optimize all the resources available, taking into account their capabilities and the latency between nodes.
- Allow developers to compute data with their custom code within a set of possible functions that may include summing, subtracting, minimizing, etc.
- Minimize the amount of data that is sent across the network by filtering it directly at the source.

1.4 Document Structure

The rest of this document is organized as follows:

- **Chapter 2:** Presents Cloud and Edge Computing in more detail with the current solutions available and why there is a need to evolve even more into the concept of Edge computing.
- **Chapter 3:** Describes our Solution with the details and the approach for achieving it and evaluating accordingly.
- **Chapter 4:** Evaluates our solution, comparing it with current approaches.
- **Chapter 5:** Presents our conclusions and how we intend to continue working on our Framework.

2 Related Work

In this chapter we will address cloud computing and its limitations as well as the emerging concept of edge computing, explaining its properties and how it can help to mitigate some of the increasing limitations of using remote data centers far from the end users and devices.

We will discuss some technologies and services that are currently available for public use, including the major Cloud providers like Amazon, Microsoft and Google. We will also study and discover different aggregation methods and data processing techniques in distributed computing.

2.1 Cloud Computing

The notion of network based computing started in the decade of 1960, with the use of mainframe computers which were used by large organizations as a way of allowing their users to access data and other resources. As the years progressed, the internet evolved and the need for wireless devices increased. In 2006 big companies like Google and Amazon introduced what we now call the modern cloud computing, a way of delivering computing services like servers, storage, databases, networking, software, analytics and intelligence over the internet. According to The National Institute of Standards and Technology (NIST) [19] cloud computing *"is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"*.

Cloud computing also allows companies to pay only for the resources they consume, unlike for example the mainframe computers that usually had expensive licenses and had to be maintained manually. With this in mind, from a business perspective, cloud computing is showing as the way to go as it allows to save a lot of costs and time by eliminating the need to plan ahead for provisioning and eliminating the use of unnecessary resources.

2.1.1 Concepts

Cloud computing was built in a way that can simulate our own devices and allows for applications to be deployed by using virtualization [38]. When a new service is deployed in the Cloud, it is provided with a virtual ecosystem of hardware and software. This allows users from different devices to access the same hardware in the Cloud. Virtualization comes with different features [38]:

1. **Flexibility:** Companies and organizations have the ability to share resources without sharing critical information and data across the system
2. **Security:** The risk of multiple attacks in case of a vulnerability is reduced by the ability of isolating hardware and software by virtualization
3. **Data protection:** Virtualization allows companies to move whole applications and services to several locations in a very simple way because it consolidates servers and resources into files.
4. **Cost effective:** Reduces hardware requirements and the need for maintenance
5. **Access control:** Virtualized hardware infrastructure is secured by custom rules

Having a virtualized system means that the Cloud providers can easily and automatically scale the virtual machines that are provided for the applications. An application owner can configure the provisioning in a way that it is scaled automatically in case of an increase of service demands and at the same time not being worried about maintenance expenses.

2.1.1.1 Types of Cloud Environments

Although the tendency is to move everything into the Cloud, there are some aspects that need to be considered for different types of usages and needs. Some companies, like in the banking industry, need to keep some information and services running on their own premises. This type of necessity has created 4 main types of cloud computing services in the Cloud environment, each one with its own benefits and disadvantages that need to be considered [36]:

1. **Public Cloud:** Provisioning of services and infrastructure for open use by the general public. This type of cloud provides their users benefits such as no up-front capital costs and the ability to scale on demand, making it a pay-as-you-go service. However, all the infrastructure is under full control by the Cloud provider and customers must accept their terms and conditions.
2. **Private Cloud:** This type of cloud environment is usually secured by a firewall and used by a single organization. It can be a full on-premises service managed by the

organization itself or by a cloud provider. Private cloud solutions offer more control and security making it easier to restrict access to valuable resources. However, the organization is responsible for the management and maintenance of the software and infrastructure, making it less cost effective.

3. **Hybrid Cloud:** A combination of both public and private cloud, that tries to eliminate the limitations of each approach. This design is scalable but still possible to secure important information.

4. **Community Cloud:** Not as common as the other solutions, being a collaborative platform used by different organizations and companies, allowing them to easily share and collaborate at a lower cost. However, this approach may not be the right choice for every organization as it raises some concerns in terms of security, compliance, and performance.

We will later discuss about edge computing and how it is connected to the Cloud. edge computing is a very broad concept that can be integrated in any of these 4 types of Cloud services. Refer to Figure 2.1 for more information.

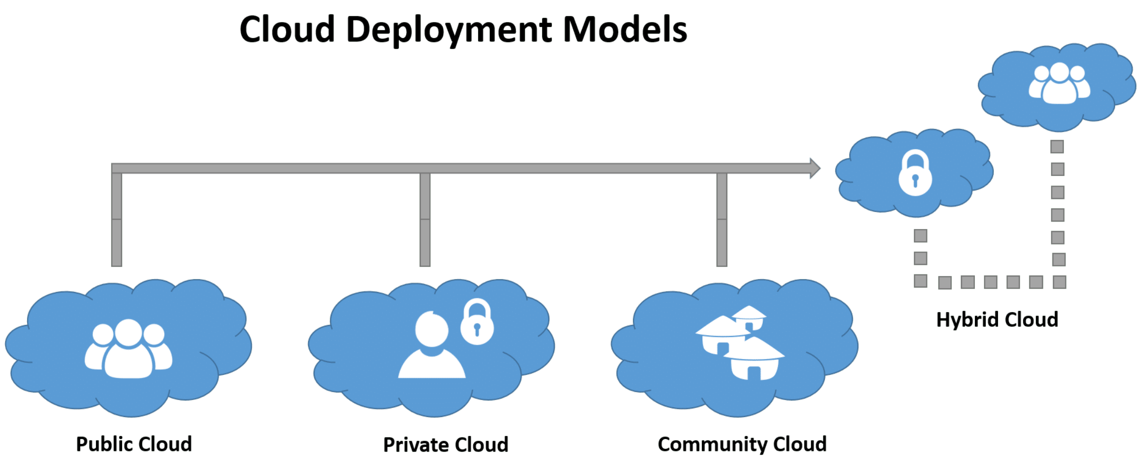


Figure 2.1: Cloud Deployment Models

2.1.1.2 Service Model

Organizations that chose to use cloud services are served with hardware and resources as a service on demand. As customers have the ability to scale when necessary, cloud providers can take part in managing some of their resources. This situation divides the service model into 3 separate types [22]:

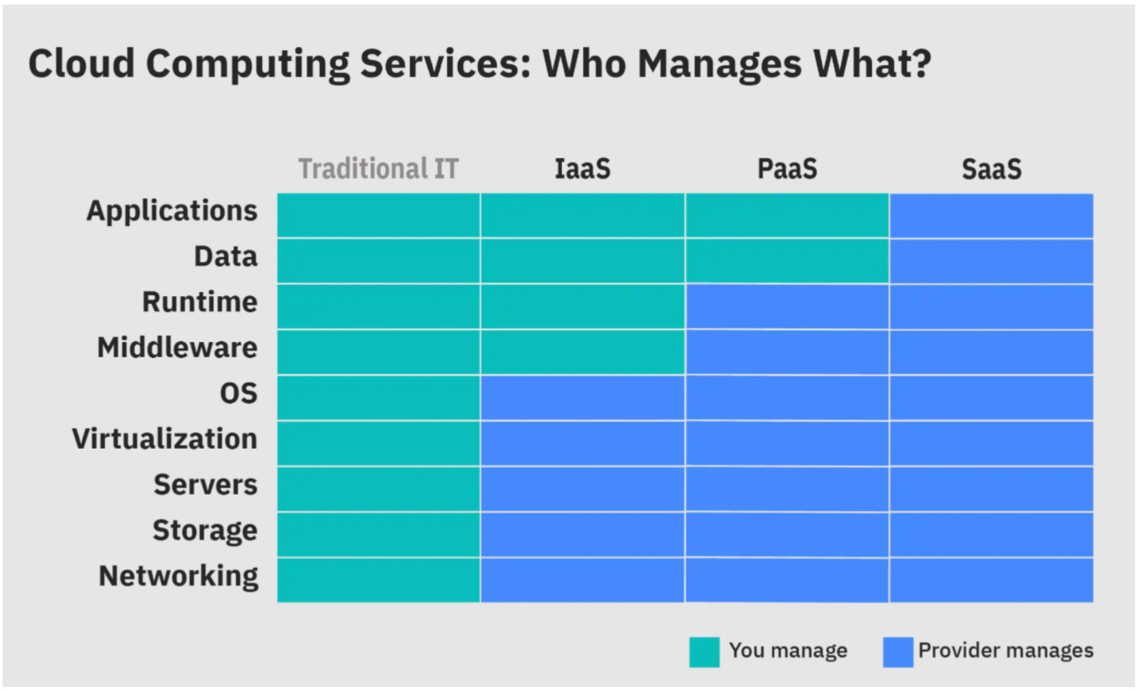
1. **Infrastructure as a Service (IaaS):** provides access to a complete infrastructure with servers, storage capacity and networking resources that customers can use as

they would do with their on premises hardware. Customers use the hardware with an internet connection but it is still maintained by the Cloud provider.

2. **Platform as a Service (PaaS):** provides platform level resources for developing, administering and running applications. The provider manages and maintains the hardware and software included in the platform. This is an easier and more cost effective solution then IaaS, if the intention is to run applications without having to administer the OS.

3. **Software as a Service (SaaS):** provides on-demand applications, like Email, social media, and cloud file storage solutions (such as Dropbox or Google Drive) that run on cloud environments and is fully managed by the Cloud provider.

The type of service model chosen by each organization or individual is very important as it can help in the development of their service or application. Later on this document we will also introduce another service called Function as a Service (FaaS) that can be seen as a solution for splitting computations into smaller units, being then processed by independent functions. Refer to Figure 2.4 for more information.



Cloud Computing Services: Who Manages What?

	Traditional IT	IaaS	PaaS	SaaS
Applications	🟩	🟩	🟩	🟦
Data	🟩	🟩	🟩	🟦
Runtime	🟩	🟩	🟦	🟦
Middleware	🟩	🟩	🟦	🟦
OS	🟩	🟦	🟦	🟦
Virtualization	🟩	🟦	🟦	🟦
Servers	🟩	🟦	🟦	🟦
Storage	🟩	🟦	🟦	🟦
Networking	🟩	🟦	🟦	🟦

🟩 You manage 🟦 Provider manages

Figure 2.2: Cloud Computing Services: Who manages what?

2.1.2 Examples

There are a vast amount of cloud service providers but by far the most known and used in the industry are Microsoft Azure (Azure Cloud Services), Amazon AWS (Elastic Compute Cloud) and Google Cloud (Google App Engine) [20]. Although they all provide storage, servers, virtualization and middleware, it is provided in different ways [38]. In this topic we will refer to these computing services and compare their differences:

1. **Google App Engine:** is a Paas (Platform as a Service) [20] that is used all over the world to host and manage web applications. With this type of service, Google deals with the scaling of the resources automatically so that developers focus only on the development itself. This service works together with Google Compute Engine that runs workloads on virtual machines running at physical data centers.

2. **Microsoft Azure Platform:** Microsoft offers Azure Cloud Services as an opponent to other web applications hosting services, with the same characteristics (provisioning, load balancing, auto scaling) [44].

3. **Amazon EC2:** is the oldest of the 3 and has bare metal instances that allows applications to access memory and the processor of the hosting server directly.

Later on in this document we will also refer to other products offered by these cloud providers that are also very helpful in our study to improve latency and user experience.

2.1.3 Discussion

Now that most applications and services work in the Cloud, they are accessible by almost any device that has an internet connection. This also means that data is being stored on remote servers (far from the end-user) and that computations are also being addressed remotely. These cloud servers are usually more powerful than our day to day devices but we are dependent on the latency created by the wireless connections.

What we are trying to understand is that even with lower computational capacity, the devices closer to the source can still provide a better response in dealing with some computation. The problem is that this situation isn't very common nowadays because there isn't much information available or frameworks and products that could help.

With cloud computing, new challenges are emerging with the growth of IoT (Internet Of Things) applications, increasing the amount of data produced and manipulated. Although the Cloud platforms and services are easily scalable, the amount of time it takes to process and communicate the data will increase. Networking connections are increasing exponentially, as shown in figure 2.3, but the bandwidth speed is not following up as it should, becoming a bottleneck.

The cloud will always be part of almost every application and system but it will begin to have a different role as the years and technologies evolve. The Cloud will always be essential for data storage or backups and for any type of computation that doesn't require a real time response.

In the next topic we will discuss Edge Computing, a concept that is trying to help mitigate these issues.

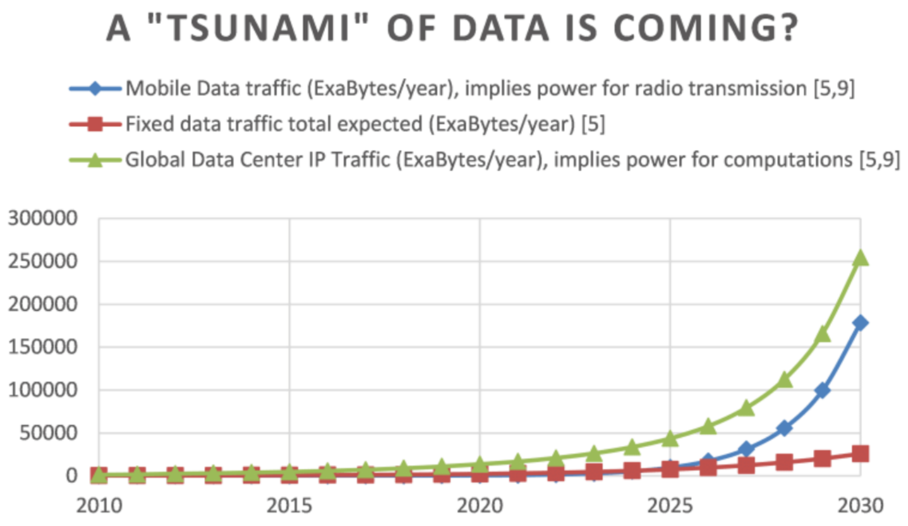


Figure 2.3: A "tsunami"of data is coming

2.2 Edge Computing

In this topic we will address the source of Edge Computing [29], how it is related with the Cloud and how this concept can directly affect cloud applications and their end-users. We will also address some of the flaws and issues associated with this type of architecture and how our framework can improve the integration of edge devices into existing or new applications.

2.2.1 Concept

As the years went by, more applications were transferred and created on Cloud environments. Although this has brought a lot of benefits, there is no way of escaping the fact that the servers hosting the applications are (relatively) far from their end-users. In most cases, latency is a very important factor to take into consideration and with the amount of IoT devices that are transferring and receiving data from the Cloud, it is inevitable that there is an increase in latency. Edge computing is trying to mitigate this by complementing the Cloud architecture with computational nodes closer to the data (users or IoT devices) [48], allowing for some (or all) computation to be made outside the Cloud.

This concept can provide faster, more stable and at a lower cost services. Users should experience a faster and more consistent experience while companies and providers will also benefit from low latency and the possibility of real time monitoring. With this, it is possible to reduce network costs by avoiding possible bottlenecks on the bandwidth, limiting service failures and a better control on sensitive data (it is possible to manage specific data on private networks using edge nodes, without reaching the Cloud), allowing companies to enforce specific security policies.

With a faster and more stable service, it becomes possible to conduct big data analytics in real time, allowing to make changes and decisions as the data is being processed. Since the data is being handled close to the origin, it isn't compromised by transference latency. Having the possibility to collect and analyze data in real time, contributes to the appearance of AI/ML services and applications, like text translation (with an image) or autonomous driving. We are going to go more in depth with some examples on the next topic. Applications and services can take advantage of edge computing when taking data from many data points and using them to create new information and make predictions and decisions.

Big data applications (with a lot of data processing) are split into several different stages that are performed possibly in different locations (physically). The first stage is the data ingestion, in which the data is gathered by the several producers and transported. The Edge concept [29] takes part at the ingestion stage. After this, data is analyzed (used for machine learning for example), usually at the public cloud, or if there are any security constraints, at the private cloud (or mainframe). After this stage, the machine learning models are sent back to the Edge for runtime analysis. This is only possible because of the proximity between the Edge and the data producers.

Edge computing is a very important step to introduce a hybrid cloud vision that offers a good overall operation and in some cases a better user experience. Currently there aren't many services that help with introducing an Edge architecture into our day to day applications. Developers and organizations are "on their own" when it comes to this topic, meaning that they have to create and use their own devices or techniques to make this possible.

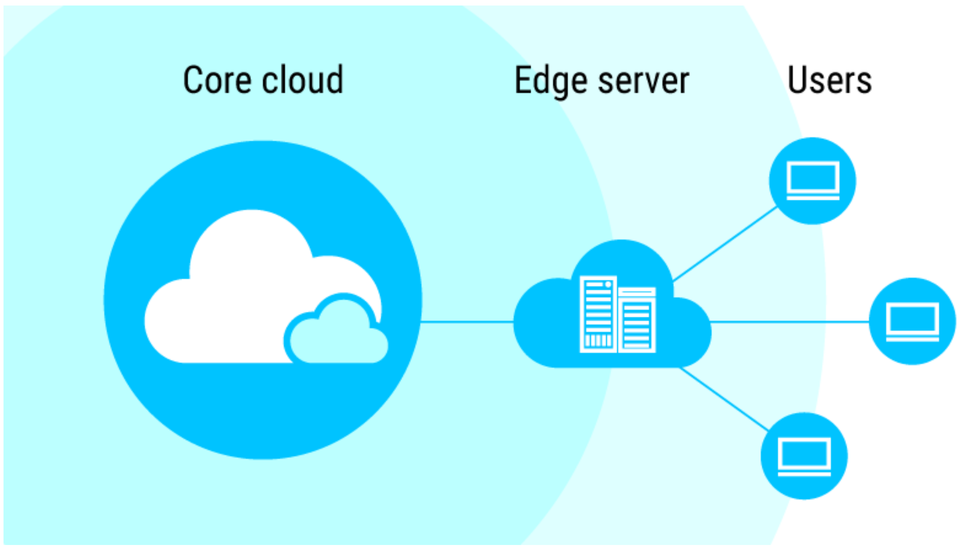


Figure 2.4: Edge and Cloud relation

2.2.2 Edge relevant use cases

As we stated before, many edge use cases exist because of the need to process data locally in real time, when transmitting information and data to a cloud datacenter causes unacceptable levels of latency and possibly higher costs. Every day, more and more companies and services are starting to use edge computing on their daily basis:

- We can take as an example a modern factory that produces cars, in their plant there are hundreds of Internet of Things (IoT) sensors that produce data that needs to be analyzed to prevent any malfunction or for quality control. These sensors generate a big stream of data that we can easily calculate: If each sensor produces 500 bytes per second, having a total of 1000 sensors it produces 43.2 GB of data per day, 302.4 GB per week, 1.34 TB per month and 15.76 TB per year. It's faster and less expensive to process that amount of data closer to the equipment, rather than transmitting it to a remote datacenter first. But it's still desirable for the equipment to be linked through a centralized data platform. That way, for example, equipment can receive standardized software updates and share filtered data that can help improve operations in other factory locations. It is also important to create a remote backup of the data produced (either being computed or still raw).

- Another example is the need for real-time data analysis on autonomous vehicles. Being a real health hazard for pedestrians and other cars, this type of vehicles are suited with dozens of sensors that are analyzing the surroundings of the car in real time, producing different types of data that need to be analyzed to make decisions as the car is moving. In this situation, all of the data processing is done inside the car by a local infrastructure with its own operating system but this system is also connected to the Cloud or the Edge to receive software updates and, for example, real-time traffic information.

- With the appearance of 5G communication, the telecommunication providers are running their networks on virtual machines closer to the users. An edge computing architecture allows these providers to keep their software running in several remote locations with their custom security policies. This software running closer to the end-users reduces latency and the overall experience.

- A CDN (Content delivery network) can also be an example of edge computing [49] as a server that is storing content closer to the user. This can be applied in general websites by allowing their users to get the website content in a faster way. The CDN provider will have servers in many locations and they can work as a connection point between different networks at the Edge [42]. This will make it more efficient and faster for data to travel from the initial source to the final destination.

2.2.3 Fog Computing

The meaning of the word Fog, refers to [28] *a mass of cloud consisting of small drops of water near the surface of the earth*. As a distributed computing concept, similar to real life, Fog computing takes place between user or IoT (Internet of things) devices in an infrastructure that connects end devices with cloud data centers. Just like edge computing, Fog is a concept that tries to bring computation closer to the users or devices that are producing data, but in this case, by using the network connection between the devices and the Cloud as a computational force.

Fog nodes [48] are essential for this type of computing and they can be any device with computing power, storage capability and can be placed anywhere with a network connection (5G, Wi-Fi, wired...). These nodes act as a gateway that can receive different kinds of data and address it accordingly. For example, if the service is dealing with real-time data that needs to be analyzed immediately, it should be connected to the nearest node available and on the other hand, if the data isn't time sensitive or doesn't need to be analyzed in real time, it can be sent directly to the Cloud for further analysis or just for storing.

Fog data stream architecture has multiple layers that are used in combination with each other [51]. The **Application layer** is responsible for defining the logic of streaming jobs, the **Processing layer** carries the processing jobs and hosts stream processing engines that will be discussed later like Apache Spark and Apache Flink. The **Data layer** holds data in databases, caches or warehouses and they work together with the processing layer. The **Resource management layer** holds the virtualized system focusing on the utilization of the network, CPU, GPU, memory, storage, etc. and the **Physical layer** that is this situation holds the physical network infrastructure. This infrastructure is likely to be managed by the organization that owns the devices and the software.

As we seen, and to resume, Fog computing has some benefits:

1. **Better response time (latency):** As the initial analysis and processing of data occurs near the source, the latency is reduced and the end-user should experience a more responsive service.

2. **Bandwidth preservation:** The Fog computing concept reduces the amount of data that is sent to the Cloud, thereby reducing bandwidth consumption and related costs. This also improves the latency of the service as the computing doesn't depend on the transference stage.

3. **Multiple networks:** While edge computing generally processes data at a device level, Fog computing computes resources at LAN level (5G, Wi-Fi, wired...)

But also some drawbacks:

1. **Costs:** On one side, Fog computing uses extra infrastructure that is not required in a cloud computing solution and this may add up additional costs. On the other hand, it can also save money by reducing latency and the bandwidth usage.

2. **Physical location:** Similarly to edge computing, it usually depends on a physical location on premises, as opposed to cloud computing.

3. **Uncertainty:** This concept has been idealized many years ago but there are still different approaches on Fog computing by different organizations and providers.

Fog computing can be very helpful if used with our framework to optimize the Edge nodes because its resources can be very dynamic.

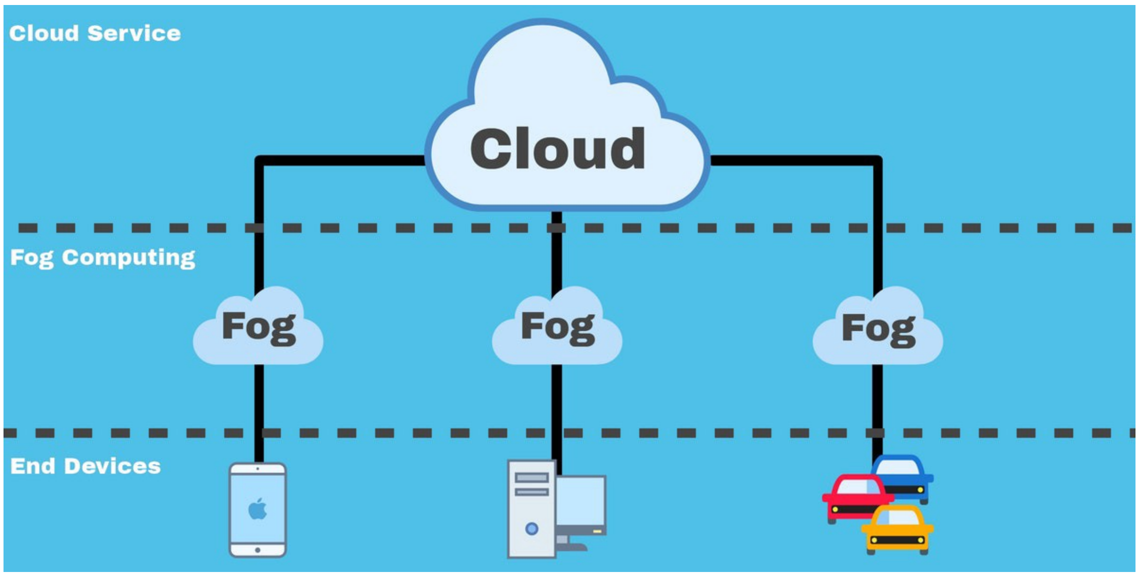


Figure 2.5: Fog Computing

2.2.4 Discussion

In the previous topic we discussed cloud computing and its architecture, uncovering some limitations that could be improved. In this topic we introduced the edge computing concept that can be seen and used around multiple scenarios. Edge computing was born from the need of a faster and better computing experience, meaning that it can be used to connect users or devices and the Cloud. We discovered that in some scenarios where a lot of data is being produced, in environments like warehouses or even web applications where the communication time has a big importance on the response latency, edge computing can help reduce this bottleneck by reducing the length that the data has to travel to reach its destination.

Edge nodes can have a lot of different shapes and sizes or can even be the device that produces the data for further processing. Knowing how the Edge works allows us to find ways to improve even further the latency of data processing and computation. We decided to focus on edge nodes with little computational power that can't host and run the main processing softwares and with this situation, find a way to split the data into smaller units and then aggregate the final results to reach a final output.

Although these technologies and architectures can help and improve modern applications and solutions, they also have some issues and flaws. Fog computing for example requires specialized hardware that can be costly and doesn't take advantage of the devices that are really part of the Edge. Currently there is no software that helps in decomposing monolithic computations into smaller units to remove pressure from network links. Edge computing can make a big difference on applications if we are able to split computation throughout multiple locations on the Edge, meaning that the data transfer times should become irrelevant.

To better understand computation techniques we will then study and discuss some of the most used software in distributed computing platforms that are offered by the main cloud providers. We also need to understand aggregation protocols and how they can be applied to our context, where we intend to increase the amount of applications and services that use edge computing.

2.3 Kubernetes at the Edge

With the increasing number of edge devices that are able to process data there is a need to improve and unify the way that the data is transmitted and processed.

Kubernetes [50] is an open-source platform that manages workloads and services in the form of containers. It is portable and extensible, meaning that it can be automated. Kubernetes containers [23] are similar to Virtual Machines but they are able to share the same Operating System among applications. Containers, unlike Virtual Machines, are considered lightweight and this is why they can be very helpful in the concept of edge computing. Each container has their own memory, processes and filesystem but they are very portable and can be used across multiple OS's.

Being lightweight, containers have become very popular and they bring a lot of benefits to edge applications if installed properly:

1. Easy to develop and integrate new updates providing a reliable image build and deployment.

2. Allows to get information and metrics about the work state of each container and application.

3. Provides consistency across all stages of development, testing and production. Kubernetes runs on multiple machines in a similar way.

4. Runs on most operating systems independently of the device being on the Cloud or on premises.

5. Containers raise the level of abstraction and split applications into smaller pieces that are deployed and managed dynamically.

Kubernetes is a system that's responsible for managing these containers that run the applications and make sure that there is no downtime. If a container is down in a Edge node, Kubernetes is responsible for starting another one. It provides developers a framework to run the distributed systems and a way to make containers communicate with each other:

1. Containers are exposed using the DNS name or their own IP address. It also has a load balancing mechanism to distribute network traffic across the network. This is very important because Edge nodes need to communicate with each other and with a master node to perform computations.

2. Kubernetes allows developers to choose their intended storage system. On edge nodes it is possible to use a local system for faster processing. In other alternatives it is also possible to connect to a remote cloud storage system.

3. If there is a necessity to change the way that the data is processed, it is possible to change the state of the container to a new one.

4. Developers can inform Kubernetes how much memory and CPU it's required for each container and each node will be fitted with enough containers for an optimal performance.

5. Kubernetes performs a self check on containers to guarantee that they are performing as they should. If they are not, Kubernetes has the ability to delete and create new containers.

6. Kubernetes has the ability to store important information and secrets in a way that they aren't exposed to the outside.

Kubernetes is a very good platform to use at the Edge. It is possible to install containers on devices closer to the data source [37], even if they have low capacity in terms of memory and CPU. Kubernetes is a way to extend the benefits of the Cloud to the Edge with a very flexible architecture.

There are already some open source projects that make use of Kubernetes in a more improved way to manage workloads at the Edge:

1. Microsoft released an open source project called Akri with the intention of connecting small edge devices like cameras, sensors, microcontrollers and making them part of the Kubernetes cluster. Akri collects the output from the devices, extending the Kubernetes framework. This is one of the possible approaches for devices that are unable to run Kubernetes containers on their own [43]. Akri has multiple

controllers that can be configured in order to search for certain devices inside the Edge. The containers will look for the devices and if they have permission they will connect and start to receive the produced data which will then be analyzed and processed.

2. Krustlet is another open source product released by Microsoft that takes advantage of WebAssembly modules [39] which are binaries and not Operating Systems environments. These modules are very small and can be placed in edge devices or networks. Developers are able to compile code from most of the known programming languages. This code will then be executed by the Edge devices.

3. Cloud providers like Amazon, Google and Microsoft are also developing their own solutions for building edge applications. We will detail some of their products in the next topics.

2.4 Aggregation

Common computations that usually occur on the Cloud can also be performed at the Edge, depending on the Edge resources availability and capability. This leads to a need to build aggregation protocols [3] for the Edge devices, specifically for cases where these nodes can't sustain big computational challenges. Aggregation also helps in decreasing the amount of data that is transferred at once, which usually floods the network links between nodes. By splitting data into smaller units at the same time that we reduce the distance between source and destination (using edge instead of cloud), the overall efficiency will improve significantly.

In mathematics, aggregation functions are a type of computation that involves a range of values that are calculated independently and then results in a final output taking into account all of the different values [3]. Some examples of aggregation functions are the Average, Count, Maximum, Minimum, Range, Sum, etc. but each of these functions have different characteristics that need to be taken into consideration:

- Functions that have commutative and associative properties, like Maximum or Minimum are simpler to aggregate as they don't need to use other functions to reach a final result. On the other hand, the Average is an example of a decomposable aggregation function because to reach the final output we need more information from each unit like the number of values being calculated for each unit and apply another function. For the Average function, the output from each node should be a pair of (x, y) where x is the average for that set and y is the count of values.

- Another property that's important to take into account is if the aggregation function is sensitive to duplicate values. Mathematical operations like Sum are directly affected by the presence of duplicate values but on the other hand, Maximizing a set of values can ignore duplicates of a value that was already processed.

2.4.1 Aggregation on the Edge

As we mentioned before, computations at the Edge tend to provide a lower latency than in the Cloud. If we introduce aggregation into this concept it will improve the performance even more [4]. As we learned, it is better to deal and process data closer to the source (or on the source itself). For most IoT (Internet of Things) applications, what really matters is the final output, for example, analyze in real-time the risk of something collapsing or going wrong in a factory, and it is not necessary to store data. If we have sensors working in different machines inside a factory that are constantly producing logs that need to be analyzed in real-time, we can have a small device with little computational resources attached to the sensors that are summing the logs. Then, from time to time this data is transmitted to a central edge node (possibly inside the factory) that aggregates the results from each sensor device and creates a final output to be analyzed immediately by the factory workers. For compliance purposes, this data can then be stored in the Cloud as a backlog for future references or to do some machine learning. As this information doesn't affect the factory it can be calculated in the Cloud as the latency is irrelevant.

Usually each edge node holds a calculated value in a static way, meaning that it won't change for a certain amount of time or until it is transferred to the main node for analysis. If it is required an even faster analysis or processing of the data we need to start thinking about continuous aggregation protocols. This means that every time that data is produced it changes the value that was calculated in each node. The main aggregator [27] needs to be able to follow these changes to produce a real-time output.

In our work we will have to take into account the type of aggregation function that is going to be used for the data that is provided. There are some kinds of functions that require more processing capabilities than others but everything will be part of our framework. We will discuss later a way of removing unnecessary data that would only delay the processing of data.

On the next topic we will discuss some aggregation protocols that follow batch processing (data is sent from time to time in batches) and stream processing models (data is continuously being analyzed).

2.5 Distributed Computing Platforms

In this topic we will address some examples of Distributed Computing platforms that operate and manipulate data produced by users or IoT (Internet of things) devices. Cloud and edge computing are part of the Distributed Systems concept, in which multiple machines in different locations operate together to form an unified system.

Artificial intelligence and machine learning are shifting from batch processing to real time and event driven applications that require the data to be analyzed in real time at the Edge nodes near the data sources.

We will discuss and detail some products like Hadoop Map-Reduce, Apache Flink, Apache Spark, etc. that are currently used to process large amounts of data with multiple machines and devices executing simultaneously. However, none of these existing solutions take into consideration the type of action being executed as we will conclude finally in the discussion.

2.5.1 Hadoop Map-Reduce

Map-reduce [21] was introduced because of the need to process large amounts of data, doing it in parallel by dividing the work into smaller and different independent tasks. This model was inspired by functional programming languages such as Lisp, Python, Erlang, Haskell, Clojure, etc. and targets specially data intensive computations. Data can have any format, being specified by the user or programmer, but the output is always a set of <Key, Value> pais. The algorithm is specified with the use of a Map and a Reduce function, as well with a sorting stage:

1. **Map:** The data provided as input is split into smaller sections. Depending on the size of the data and the memory available on each mapper server, the Hadoop [21] framework defines the number of mappers. After being split, each block is assigned to its mapper for further processing. These mappers are called worker nodes and are managed by the master node that checks each worker for a correct execution.

2. **Sorting:** After the Mapping function is over, the keys produced are then sorted (the values can be in any order).

3. **Reduce:** After all the worker nodes complete the mapping function, the Reduce can start. It processes and produces pairs of <Key, Value>, keeping the data type as in the beginning. All values that share the same key are assigned to a single reducer that will then process it.

We can consider that the execution of a Map-reduce function is called a job. Each job has a map, sorting and reduce phase. As an example (that will be discussed later), we can consider a job that counts the number of times each word appears in a set of documents. The map function will execute for each document the main computation (counting the occurence of each word). The input data is splitted automatically to run in parallel across the Hadoop cluster and it is received from the Hadoop Distributed File System (HDFS). HDFS enables streaming access to file system data in Apache Hadoop, its fault tolerant and stores data across multiple machines. The reduce stage will receive the result from the map tasks and form a set of parallel reduce tasks, aggregating the output of the map stage for all the documents, providing the final output. The final result is usually stored in HDFS. Before the reduce stage all of the output values from the map stage there is a shuffle process that creates a single <Key, Value> pair, where the value becomes a list of all elements sharing the same key.

As an example, we can consider that we have data splitted into 2 separate documents (doc1 and doc2). These two documents are considered the input data and are divided as chunks called splits, during the job configuration [2.6].

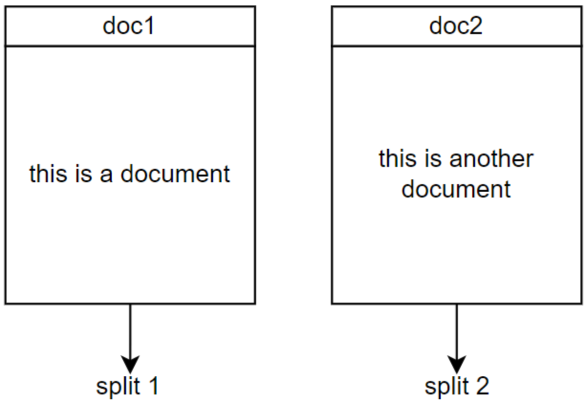


Figure 2.6: Document to split

These splits are included in the job information that is used by the job tracker. The tracker manages the tasks in each node on the cluster, creating a task for each split, transforming it into <Key, Value> pairs. In this case, each occurrence of a word represents a pair where key = word and value = 1 and it is possible to have repeated keys. Between the map and the reduce occurs the sorting and shuffling functions, where the pairs are ordered and split. The reduce stage will then receive the ordered values and produce an aggregated word count, accumulating the initial splits. The final output won't have any repeated keys [2.7].

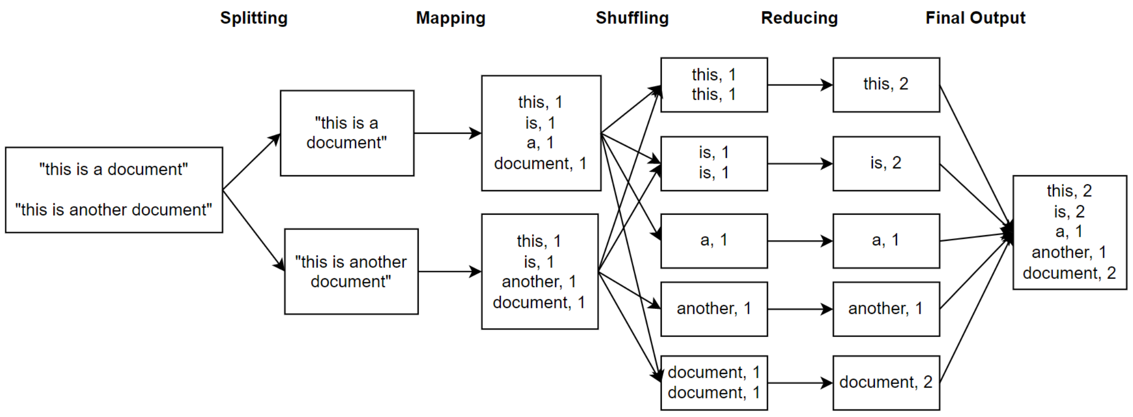


Figure 2.7: Map-Reduce phases

Hadoop Map-Reduce [21] may be very helpful in processing big data but it does it in batches (batch processing). This means that the data is analyzed all at once and it is finite. Only after the process of reducing is completed can the output be analyzed.

On the next topic we will provide an example of a stream processing data analysis product.

2.5.2 Apache Flink

Apache Flink is a data streaming processing framework [5] means that it is able to analyze data in real-time, as opposed to Hadoop MapReduce that receives data in batches. Data streams are a flow of information or data that is being transmitted from the source to the destination in a continuous way, ordered by the time it was produced. These streams can be heterogeneous, meaning that they are produced by different devices (for example sensors).

For our case study it is important to understand how we can split data streams into multiple smaller units. If the situation is that each device produces a data stream there is a possibility to have an edge node for each data producer to analyze individually the data. After this step, the output will be aggregated by a master node.

Data streams can be divided into two types:

1. **Bounded streams:** Data with a defined start and an end, meaning that it has a fixed size and the computations come to an end.
2. **Unbounded streams:** Data with a defined start but without any defined end. It is unknown the size of the data and when the computations will be over.

Although there are two different types of data streams, flink as a unified framework for their processing.

A very common use case for stream processing is the use of risk control systems. While the data is being processed in real-time (if possible on the Edge), a set of rules can be applied to check if there is any issue with the data and set an alarm for example.

With the implementation of consistent checkpoints, Flink has 24/7 availability. Checkpoints are responsible for handling faults, by consistently recording the state of operators during computations and generating snapshots for persistent storage. These states are data information that is generated in the middle of computations and it's very important to guarantee a recovery after a crash or failure. As stream computing can be considered as incremental (computation keeps increasing over time), it requires the system to query the data continuously. When some job fails within the Flink normal computation, it allows the system administrator to restore the job from a previous state checkpoint, making sure that no data is lost during the process.

2.5.3 Apache Spark

Similarly to Apache Flink, Spark [34] offers a scalable and fault tolerant data processing framework that can analyze and receive data in batches or streams. It provides the same framework for both data types and this is one of the reasons why it is used worldwide.

Opposite to Flink, Apache Spark simulates data streams by using micro batches [34] meaning that it can only provide near real time processing. Although the data is being analyzed continuously, it still suffers from the division into micro batches and aggregation. However, this increased latency can be acceptable for some applications. The throughput (amount of data as input) is directly proportional to the latency in this situation. This means that developers must find a way to optimize their Spark configuration as it is not improved on its own.

It is also possible to query data using SQL. Spark framework has SQL support to make it easier to retrieve data in a homogeneous way, independently of the data source, that may include Hive, Avro, Parquet, ORC, JSON or JDBC. There is also an API for querying data.

2.5.4 Functions as a Service (FaaS)

In this chapter we will discuss Functions as a Service computing and how it can be related to edge computing.

2.5.4.1 Concept

Functions as a Service (FaaS) is a computing service based on the Cloud environment [46] that allows to execute code when necessary (by triggers or timers for example) but without the need for the complex infrastructure that is usually required for deploying and managing micro services applications.

As we mentioned above, for deploying and hosting a web application on the Cloud, it is usually required to provision and manage infrastructure like servers and storage or even the operating system. With the FaaS concept, all of the infrastructure is provided and scaled when necessary by the Cloud provider in which you choose to deploy your FaaS service. This makes it easier to develop functions independently and in a more simple way [18]. This concept is very useful if an organization is migrating from the mainframe to the Cloud for example, allowing it to build independent functions with some benefits:

1. **Pay when you use:** FaaS computing works as an action. When the function is called, instances are deployed and the action starts. Users and organizations will only pay for the time that the action is occurring. This makes it very cost effective.

2. **Benefits of cloud infrastructure:** As this service is offered by cloud providers, it has high availability as it can be deployed on multiple availability zones in multiple regions across the world.

3. **Automatic scaling:** As it happens with cloud computing services, FaaS can be scaled automatically when necessary. When demand increases, instances will be scaled for a better performance.

4. **Focus only on code:** As we mentioned above, with FaaS there is no necessity to manage infrastructure and this allows users and organizations to focus only on the development of the code, reducing the time to produce new services.

As it enables functions to be isolated and scaled, FaaS is being used for high-volume computations in parallel. It can be used for data processing, aggregation and other forms of computing. FaaS can be used as a backend for Internet of Things (IoT) devices, approaching the concept of edge computing.

2.5.4.2 Amazon Lambda @Edge

Amazon Lambda is an example of a FaaS (Function as a Service) [46] from Amazon. Lambda functions run code pieces on a high-availability and scalable infrastructure, totally administered by Amazon. Similarly to other FaaS options, this service is only paid for the time it's being used (in 100ms increments). Lambda functions are also monitored automatically for performance logging, reporting metrics through Amazon CloudWatch.

The use of FaaS in close proximity to the users and devices brings a lot of benefits in terms of performance:

1. If there are some values that are used more frequently, they can be stored and accessed on the Edge by improving the cache hit ratio.

2. We have the possibility to create models with machine learning that will predict the requests that are made to the Edge at any specific time of the day.

3. By increasing the amount of requests that receive a response from cached data, the overall performance will increase significantly.

4. Amazon has infrastructure spreaded around the whole world and with the help of Amazon CloudFront content delivery network, users will be served by the closest location available.

We can take as an example of a smart light (connected to a smartphone) that can be turned on and off from anywhere in the world with an internet connection. When the user presses the button, it triggers an event to a function in the Cloud that switches the light. This is a great example of using FaaS as a single function that should be close to the user, it is simple to connect and it's very easy to scale to thousands of users and devices if necessary.

With Amazon Lambda @Edge it's possible to move some computation to the Edge, providing a better user experience and lower latency for client and data processing applications. Removing these kinds of events from centralized servers will allow for a more structured and stable architecture.

AWS Greengrass Amazon developed another product that can work in conjunction with Lambda functions called Greengrass [24]. It is an open source IoT (Internet of Things) edge service that helps in developing and using IoT applications and devices. AWS Greengrass can be deployed on devices to process the data locally that is being generated. With this service, developers can use edge devices to easily analyze and react to data with custom rules, making it also possible to communicate with other devices to make better decisions:

- Being an AWS service, it provides support for transferring data (processed or not) to the Amazon cloud storage services. The Greengrass software runs on multiple platforms including Windows and Linux distributions.
- It is possible to import modules that are available for public use and that are called components. Greengrass can run on Docker containers, native OS (operating systems) or custom runtimes.
- Components [9] are basically building blocks that help developers in creating more complex workloads. These components can help with machine learning, local data processing, data management, communication, etc.
- InfluxDB is an example of a component that provides a database for Greengrass edge devices and can analyze and process data in real time, monitoring operations at the Edge.
- LoRaWan protocol adapter is another component that helps in real time analysis by ingesting data from wireless IoT devices with a specific communication protocol, avoiding unnecessary communication with the Cloud.
- AWS Greengrass supports Lambda functions [8], importing them as components. This can be very useful if there are some functions deployed on Lambda that were previously used with another architecture to be transferred to an edge based architecture.
- Greengrass requires only 1GHz of computing power (Arm or x86 processors) and 96 MB of RAM and supports most programming languages like Python, Java or C.

Greengrass can be a good solution for running computations on the Edge nodes that are being managed by our framework. Functions can be used in combination with Greengrass running inside containers to perform data processing.

2.5.4.3 Apache OpenWhisk

Apache OpenWhisk [31] is another example of a Function as a Service, allowing developers to run application computing and logic from HTTP triggers or directly from any backend or web application. OpenWhisk is open source so it is not dependent on any service like Amazon Lambda and it can be triggered using an API, allowing companies and organizations to have serverless computation capabilities.

It was also developed an attempt to improve OpenWhisk for edge usage - Lean OpenWhisk [25]. This software is intended for edge nodes with smaller and more limited nodes that can't sustain OpenWhisk. It removes some components like Kafka that are very heavy weighted by using a simple queue. As we mentioned before, in this thesis we want to take advantage of smaller edge devices with lower capability. If we divide computations into smaller units we don't need a single node to process the whole data. OpenWhisk being open sourced [7], allows us to study in more detail how we can achieve this final goal.

Similar to Greengrass, OpenWhisk is also available to run inside containers. This is very helpful because it can be triggered by a web application [11]. The master node can send information to the Edge devices that are hosting OpenWhisk, which will then perform the selected computations.

Microsoft Azure and Amazon are also starting to bring their FaaS platforms closer to the users and into the Edge with Azure Functions and AWS Greengrass, that connect to the Cloud ecosystem in a seamless way [26]. The only problem is that in order to use these services, the Edge nodes are controlled by the respective cloud provider. With the use of OpenWhisk, developers aren't affected by a possible vendor lock-in as they only depend on themselves.

2.5.5 Discussion: Lack of Computation Classifications

As we mentioned above, there are a vast amount of solutions and systems that can analyze and process large amounts of data while using multiple machines to perform computations simultaneously across the system. However, we also noticed that none of the current solutions take into account the different nature of computations when assigning different computational units to machines. This is also explained by the fact that these solutions assume a (mostly) homogeneous execution environment where all machines have similar computational resources. This means that, independently of the type of data or desired output, the way that the data processing is executed is the same.

Our solution, since focusing on the highly heterogeneous edge environment, aims to address this limitation by identifying different computational units of an application across a set of well defined classes with different properties and characteristics. If we take as an example a summing computation, it is very simple to decompose it equally into multiple units that will have a similar workload and execution, with a single output, this decomposition is sensitive to duplicate values. On the other hand, if the desired computation was to find the maximum or minimum value of the data, we will also have a single output but there is no sensitivity to duplicate values as they represent the same and would only increase the execution time and costs related with data transfers. This type of execution where the number of output values is lower than the initial input values, should probably be executed near the data sources. Another option is to have multiple functions executing at the same time which should be taken into consideration when our framework selects the ideal node or nodes for execution.

In another scenario, there are certain types of computations in which the output of the execution is larger than the initial input. For example, in a machine learning algorithm, the model is created with a large amount of data and if this model is used to perform a prediction, the output can be larger than the input. In a weather prediction, the input can be represented as a full year of real data and the output can be two or three years of forecasts. This type of scenario where the output is larger than the input is likely to be more efficient if performed at the Cloud as to avoid increasing the data to be shipped from the Edge to the Cloud. This should also be represented in our framework as a class of computation with a different annotation for improving performance.

Different nodes have different resources and it is reflected in their throughput. Current products usually wait for all workers to finish their execution to perform final aggregation. This means that in a heterogeneous environment where the nodes have different resource power, if the amount of data being sent for processing is the same, some workers will finish their execution faster than others. We will discuss in the next chapter how we plan to fix this problem and provide a better experience.

2.6 Summary

In this chapter we addressed Cloud and Edge computing. Although the Edge seems to be a very good complement for current Cloud applications and solutions, it still has a lot of flaws that can be fixed and improved. We decided to study ways to improve the use of edge computing on devices like sensors, cameras etc. where the computational resources are low and ways to minimize the amount of data that is unnecessary to be transferred from node to node as well as choosing the correct nodes for each type of processing function:

- We studied protocols and solutions that are designed for small devices using Function as a Service to compute the data, including some open source projects. This could be a good solution for splitting the computations into smaller blocks. If each edge device performs computations directly to the data that they are producing, it reduces the amount of data being transmitted from one side to the other. After the data is processed on each device, a final output can be transmitted from time to time to a master node for further processing or investigation.

- Another way could be to add another device with more power attached to the data producer and then perform computations on the secondary edge device. This could be a way of fixing the problem but it is costly to add new hardware to the infrastructure and the data would have to move from one node to another. If a lot of data is being produced and the Edge node doesn't have enough capabilities it can be a solution. Because of this we studied how the data is transmitted when it is regularly being produced: in batches or in streams. Batches have a certain amount of time between transmissions and the data is grouped and sent together. In most cases there is no problem with this situation. On the other hand, stream processing allows for faster and smoother processing because the data is always being sent and analyzed.

- In cases where the network has some edge devices it can be considered as a way of fog computing. In this situation, our framework will help in optimizing the whole service by minimizing the latency taking into consideration all of the resources available on the network itself.

- We studied how computations can be split into different classes with specific characteristics and properties that require certain resources. These different classes will be represented in our framework as annotations and will take a big part in choosing the correct node for execution. In some specific cases it can be optimal to perform the computations in the Cloud instead of the Edge.

FaaS or serverless computing was one of the focuses on this document because resources can be used as functions instead of being constantly running. Currently there aren't many lightweight FaaS solutions for Edge architectures and environments. To improve even further our solution, we can use FaaS as a simple operation where, for example, duplicate values are removed before being sent for processing. This allows to reduce the amount of data being transferred between nodes and therefore the bandwidth usage and overall latency and associated costs. We also studied containers as an alternative for virtual machines because these containers can be lightweight, scalable and portable. Kubernetes is a very good tool for managing these containers, including ways to improve communication between nodes. These containers can be hosted by the Edge devices and the master node, where the framework will be running and managing everything.

Since our intention is to divide the computations throughout the Edge nodes, we also studied aggregation as a way to reach a final output. Some aggregation protocols are already built for distributed systems and can be helpful to reach our objective.

Throughout the next chapter, we will detail how each of these factors will complement each other to build a new solution that will improve big data applications.

3 SMEdge - Self Managed Edge

During the first Chapters, we presented the Cloud as the current method for hosting applications and processing information. Although it has some limitations and downsides (as everything does), it is, and will always be very useful to perform some types of operations due to its high capacity and availability. This means that our solution will not fully replace the Cloud, but use it in a different way, cooperating with the Edge to create a better and more efficient system.

Taking the above information into account, in this Chapter we will present SMEdge - Self Managed Edge, a solution that takes into consideration the location of where the data is produced and executes specific functions across the Edge and the Cloud and can be applied in multiple scenarios. Our solution will deal with the data produced from multiple origins and process it accordingly, managing everything required to perform different types of computations efficiently. We will focus on handling data that is generated sequentially across time intervals, such as every second, minute, or hour. We understand the importance of aggregating this data according to the specific needs of possible customers. Whether it's aggregating data for real-time monitoring, analyzing trends over hours or days, or generating reports at regular intervals, SMEdge provides the flexibility to aggregate data according to the customer's necessity. By efficiently managing and processing data from multiple origins, our solution ensures that various types of computations are performed efficiently, resulting in a more optimal and effective system.

SMEdge, is a solution designed to revolutionize how companies handling big data, such as Uber, manage and process their data. By leveraging the power of the Edge and the Cloud, SMEdge optimizes data processing and transforms the way these companies operate, ultimately leading to enhanced efficiency, cost-effectiveness, and faster user experiences.

Traditionally, companies like Uber rely on the Cloud for hosting applications and processing data. While the Cloud offers high capacity and availability, it also comes with limitations and potential downsides. SMEdge recognizes that the Cloud cannot be entirely replaced, but instead proposes a collaborative approach that combines the strengths of the Cloud and the Edge to create a more efficient and effective system.

The fundamental idea behind SMEdge is to consider the location where data is produced and to execute specific functions across both the Edge and the Cloud. This approach enables companies like Uber to process data closer to its source, minimizing latency and improving response times. For instance, instead of transmitting all the data generated by Uber's operations to the Cloud for processing, SMEdge empowers the Edge devices (e.g., smartphones, vehicles) to perform certain computations locally, thereby reducing reliance on distant servers.

By implementing SMEdge, companies handling big data, such as Uber, can achieve several significant benefits. Firstly, SMEdge optimizes the utilization of computational resources by distributing processing tasks between the Edge and the Cloud based on their respective strengths. This results in improved efficiency as data processing is performed closer to the source, reducing the burden on the Cloud infrastructure.

Moreover, SMEdge offers cost-effectiveness by reducing the amount of data transferred over networks and minimizing the usage of expensive Cloud resources. By leveraging local processing capabilities, companies can reduce bandwidth costs and decrease their reliance on Cloud service providers for every computational task.

Additionally, SMEdge enhances the speed and responsiveness of applications for users. By minimizing data round trips to the Cloud, latency is reduced, leading to faster response times and a seamless user experience. For services like Uber, this means quicker updates on ride availability, real-time tracking, and overall improved reliability.

One of the key advantages of SMEdge is its adaptability and versatility as a framework. While it is initially presented as a solution for companies handling big data, like Uber, SMEdge can easily be adapted to various other scenarios and industries such as warehouses, healthcare industries, etc.

The goal of our solution is to collect, aggregate and store information provided in a decentralized manner, in the form of a time-series database. SMEdge utilizes a time-series database to collect, aggregate, and store data in a decentralized manner. The solution includes real-time monitoring and aggregates data during specific time periods. This enables companies to efficiently analyze trends, make informed decisions, and enhance system performance. By leveraging the capabilities of a time-series database, SMEdge improves data handling for better efficiency, cost-effectiveness, and faster user experiences. We believe that our project should have the following requirements:

- As we have multiple nodes spreaded across separate geographic locations, each node should only interact and communicate with the ones closer to itself. For this to be possible, each node must be aware of the others, and this will happen everytime a new node is created or deleted. Latency should be directly related with distance and this will be the metric that we use to have awareness of the other node's distance. With this in mind, it isn't necessary to have knowledge about the true location but only the relative location.

- It is also very important to be able to query data stored across the edge nodes and the cloud. The data produced by the ridesharing vehicles (or taxis) have different types (string, date, integer, etc.) and we want to be flexible to allow any type of data to be analyzed and queried. We also want this solution to be easily adapted to new companies and customers, meaning that we allow the developers to introduce their data sources and this data will be aggregated and analyzed in a uniformed way, preventing the need to create workarounds that would possibly affect the overall performance of the application.

- Each node should be aware of the performance from itself and from the closest available nodes. This allows the decision making regarding the API usage to be decentralized. For example, if a node is suffering with high memory or CPU usage, it will be saturated. The closest nodes will be aware of this situation and help with the incoming requests.

- In the case of a total node loss, we should have replicas of all raw data available in each node. In this case, a replacement node should be able to resume the work without any major issues.

- Have a simple and efficient way to communicate and propagate with other nodes in the system. As this solution aims to improve the performance, we should implement an easy way to transfer information.

- As we implemented a dashboard to allow clients and users to check statistics and information from all the nodes, we should also implement alerts and notifications that will inform the administrator about important system events. These notifications can be sent by email or text message and the logs will also be available and stored for future analysis. We allow the developers to set custom triggers and alarms, either regarding the system performance (time it takes for a message or event to travel between nodes) or even to control and manage the CPU and memory usage (the system should adapt itself but it is important for the administrator to be aware of these actions).

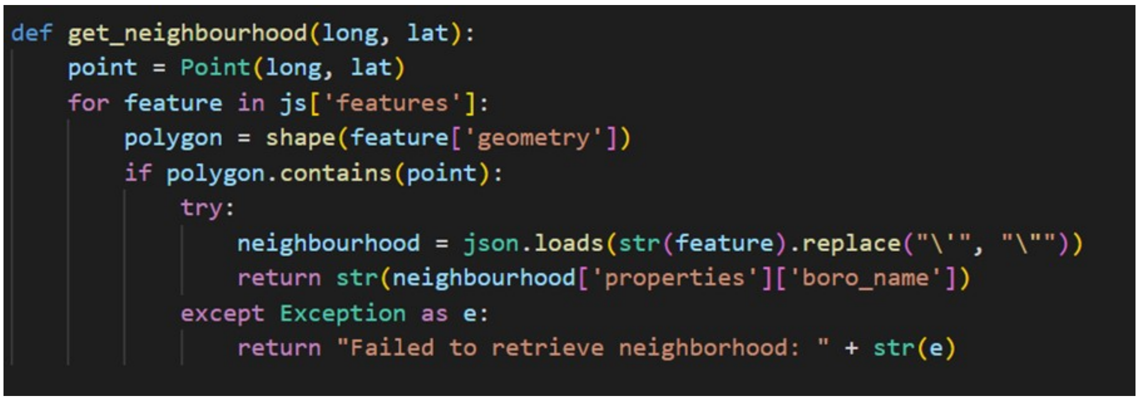
- Different types of computations should be possible to achieve between the Edge nodes and the Cloud. Some computations are performed in a de-centralized way (directly on the Edge nodes) and the aggregation of these computations is made in the Cloud (possibly using another type of computation/aggregation).

We will begin by presenting the system model and its assumptions (Section 3.1), followed by an overview of all the main components (Section 3.2), including the Tracking Module (Section 3.3), the Aggregate Computation protocol (Section 3.4), the API (Section 3.5), the Dashboard (Section 3.6) and addressing the whole design and how the system should be implemented (Section 3.7).

3.1 System Model and Assumptions

We will now present and discuss our system model which has a set of assumptions that will justify and motivate the way we design and implement our solution.

Our proposed solution is a system that processes data from multiple devices (in this case taxis or moving vehicles) and deals with it accordingly, resorting to the Edge and the Cloud, in the most efficient way. Each taxi is represented by a medallion (unique ID) and alerts the most appropriate Edge node whenever a new trip starts and ends. The taxi has a GPS and is aware of its own location in the beginning and the end of each trip. The GPS device provides the taxi with its latitude and longitude. With this information, we resort to the function getNeighborhood(long, lat) which uses the Python package shapely. The function creates a Point and checks if this point is present inside a specific set of polygons. The polygons are represented in a .geojson file which contains information about the borders of each neighborhood inside New York City. Depending on their location, the taxi will resort to a .geojson file to calculate the correct neighborhood 3.1 and send an HTTP request to the closest Edge node, which will handle the request.



```python
def get_neighbourhood(long, lat):
    point = Point(long, lat)
    for feature in js['features']:
        polygon = shape(feature['geometry'])
        if polygon.contains(point):
            try:
                neighbourhood = json.loads(str(feature).replace("\'", "\""))
                return str(neighbourhood['properties']['boro_name'])
            except Exception as e:
                return "Failed to retrieve neighborhood: " + str(e)
```

Figure 3.1: Code snippet for neighborhood finder function

Each node that will be representing the Edge has its own separate resources such as CPU, memory, storage, etc. and in this scenario they will be available to communicate with each other over the public internet (with a public IP address). Each node is responsible by itself but is also aware of the current status and usage of the other nodes.

The Cloud service is deployed as an Azure App Service (Linux) due to the simplicity and adaptability. We will approach this choice during the next topics, showing how we implemented this solution and its main benefits. For testing purposes, the Cloud will be hosted in a Docker container on our local machine, however, latency will be properly applied so it's possible to compare it to a real life scenario.

Our solution will optimize the way that companies like Uber, Lyft, Bolt and other ridesharing services deal with all the information received from the multiple vehicles registered in their platforms.

In order to design our solution, we have to follow a set of system assumptions:

- The edge nodes aren't always at the same distance from the user's location but the latency only varies from neighborhood to neighborhood - meaning that, for example, if the user is located in Manhattan, he will always face the same latency no matter of the street he is located. We follow this rule as the real difference in latency in real life is irrelevant.
- In real environments, nodes experience different latency's to every other node. Our system model assumes that, with a high probability, nodes that are closer to each other have better latency between themselves compared with nodes further away.
- It is assumed that all nodes that participate in the network are reachable by every other node, so that they can receive and reply to requests, unless they have a failure or they are simply unavailable.
- It is assumed that the location of the edge nodes is closer to the user's than the location of the cloud data-center and that the edge nodes are closer to each other than to the cloud.

Besides decreasing the length between the data and the server, we will avoid saturating the bandwidth, by using multiple edge nodes that take the load split according to their capability. Additionally, as some data is processed locally on the Edge nodes, the amount of requests to the Cloud (which has the most impact in terms of latency due to its distance) is reduced.

There are currently multiple advances in technology that support our solution such as the use of CDN's, Fog Computing and Mobile edge devices. These technologies have been arising mostly due to the expansion of 5G networking.

3.2 Overview

We decided to split our solution into 3 different modules with different responsibilities: the API, the Tracking module and the Aggregate Computation protocol. Additionally, we developed a Dashboard that allows us to visualize and test the system as a whole. During this topic and in the following pages we will describe the importance and responsibility of each module, which combined represent SMEdge - The Self Managed Edge. The modules and overall architecture can be seen in the following image (3.2).

The first component is the API and it will expose the different operations performed by other components to the multiple clients and devices. After receiving the requests, it will also handle the output (if there is any) to the appropriate handler.

With this in mind, the API has the following main objectives:

1. Perform managing operations on the Edge nodes
2. Receive messages and information from multiple devices and clients

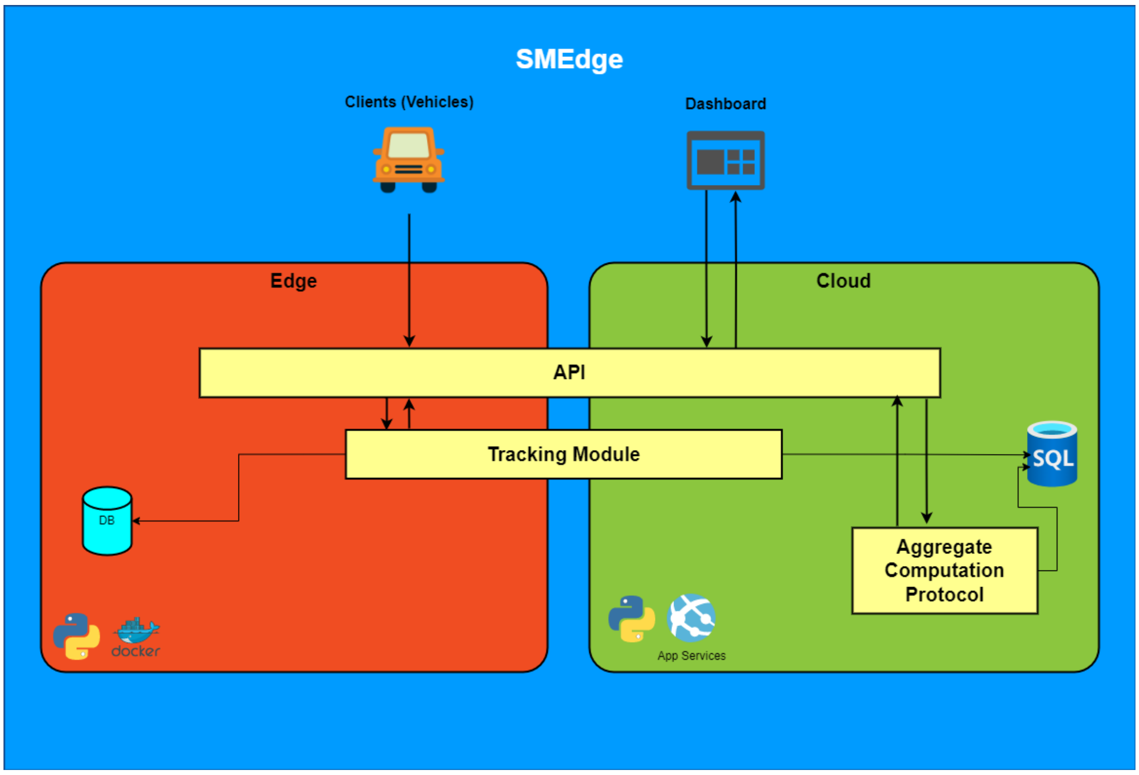


Figure 3.2: Modules and system architecture

3. Query data across multiple locations

Another important component is the Tracking Module which is responsible for managing the whole system in terms of infrastructure and trip metrics. This module has the capability of communicating periodically with all the nodes that compose the whole system and check their status, health and tracked metrics by collecting and performing small computations to it. Additionally, the Tracking Module receives requests to start tracking the metrics and makes all the necessary changes to the nodes, creating the required tables in the databases. After the data is computed in the nodes, the module is capable of performing pre-set actions set by the users.

The last component is the Computation Protocol and it is responsible for handling the data that is sent to the API by the clients and the Edge nodes to the Cloud. There are multiple types of computations which we will detail in the next sections and these are configured by the application developer. The data is collected by the multiple nodes and needs to be aggregated or de-aggregated, depending on the type of computation. This component is located in the Cloud and is responsible for picking the data from the correct databases and produce new databases with the computed data, allowing users to analyze it.

Additionally, we created a simple Dashboard where it is possible to see the whole system in action. This dashboard gets the data from the API mentioned above and can be configured by the system administrator.

In the next sections we will detail each of the modules presented above with an explanation and demonstration of its design and implementation. Starting with the Tracking Module (3.3), followed by the computation protocol (3.4) the API (3.5) and the Dashboard (3.6).

3.3 Tracking module

The Tracking Module is a component that was designed to collect and analyze the metrics related to the service that is using our solution (in this case, related with car trips). These metrics can include information such as distance, time, cost, etc. One way to monitor these metrics is to perform small computations such as calculating the maximum or the sum of its values and leading them to the Cloud.

The Tracking module performs computations on a small set of data that is stored for a specific period of time or on a set of data with a specific size, before sending it to the Cloud. This module is composed by two different components as illustrated in Figure 3.3.

1. Metric Analyzer: Each time that the API receives a request to start tracking a new metric, it routes the request to the Metric Analyzer whose main responsibility is to read each trip and store each tracked metric in a separate list. The analyzed metrics will be stored during a certain periodicity or for a specific length and sent to the Edge after the specified computation is performed.

2. Threshold Manager: As the data is sent to the Cloud, it is validated by the threshold manager. The user may specify a set of rules and conditions to make sure the data is correct. This data also includes performance metrics and if a condition is met, an action will be performed.

3.3.1 Metric Analyzer

Our system was designed to deal with data that is created over time and it is important to keep a sense of time to be able to query it accordingly. Each Edge node will store in their local memory a simple database with all the records of trips with the exact time frame of when it was created. This will be used to replicate all the information across the nodes and in case if it is necessary to perform a backup.

During the development of the system and as mentioned in the previous Chapter, it is possible to send data to the database using two different approaches: streaming and batching. Although streaming has some benefits, when dealing with large amounts of data (as our scenario) it's better to use batches to propagate the information as it brings the following benefits:

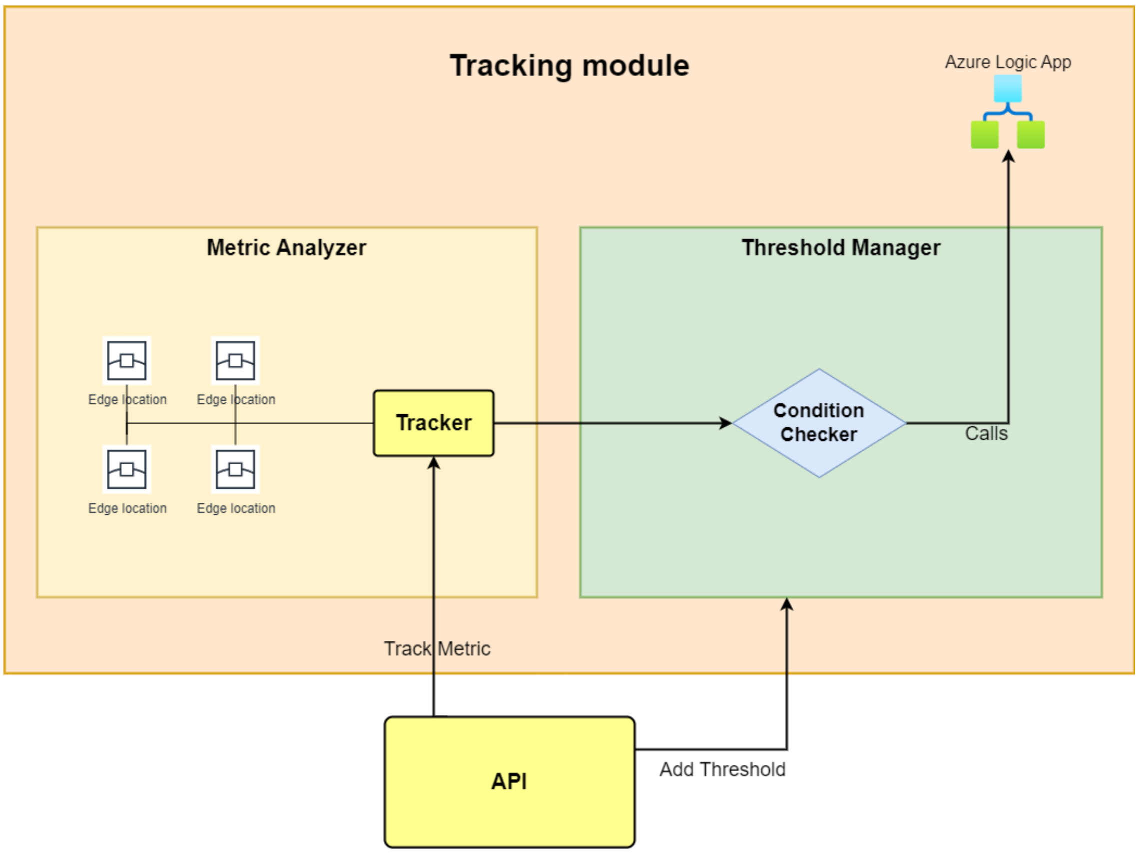
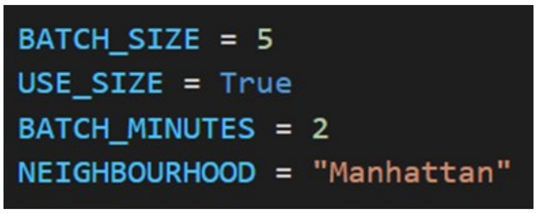


Figure 3.3: Tracking Module Architecture

- Reduced network overhead: Sending data to a database in batches reduces the network overhead, as there are fewer requests to the database. This can significantly reduce the amount of bandwidth and CPU resources required to process the data, directly reducing the related costs.
- Improved performance: Batching data can improve the performance of the database by reducing the number of small transactions, which can cause a lot of overhead. Batching allows for larger transactions, which are more efficient to process.
- More efficient use of resources: Batching data allows for more efficient use of resources, as it reduces the amount of time that the database is idle between transactions. This can help to reduce the overall resource requirements of the system.

We allow the user to define the way that the batch is used, giving two options:

- Period: Define a specific period of time in minutes. Whenever the period is reached, the batch is sent to the database and a new batch starts getting gathered.
- Size: Define a specific batch size. The length of the batch is measured every time that a new trip is added. Whenever the batch size is reached, it is sent to the database.



```
BATCH_SIZE = 5
USE_SIZE = True
BATCH_MINUTES = 2
NEIGHBOURHOOD = "Manhattan"
```

Figure 3.4: Configuration file

These settings are defined in a configuration file as shown in Figure 3.4.

If `USE_SIZE` is set as True, the periodicity will be disregarded and the policy will follow the `BATCH_SIZE` parameter.

At the same time that the main database is created, the nodes have lists inside a Dictionary that store specific data in the individual nodes. These are requested by the developer/system admin whenever a new metric starts being tracked as it follows:

- Each trip has multiple parameters that can be analyzed such as date, time, distance, cost, pickup location, drop off location, payment type, tip amount, etc.
- Each parameter can be monitored with a certain periodicity which can be manually defined (it has a default value of 5 minutes). In this case we don't allow the user to select a batch size as the intention is to use equal periods of time and this is more efficient if we deal with data produced for specific periods.
- During the periodicity value, the parameters that are being analyzed are added to a Dictionary and each metric contains a list of values.
- After the specified periodicity is reached, the values are aggregated accordingly and some type of computation is performed (such as average, count, max or minimum).

During the time set defined in the periodicity variable, the list will be filled with the values of its specific metric. For example, if the periodicity is 5 minutes and the user wants to track the trip distance, the list created to track the distance will be filled during this period. When the 5 minutes are over, the Edge node performs the aggregate computation on the data and sends it to the Cloud, to be handled by the Data Handler module which we will detail in the next topics.

Each Edge node requires a separate Time Series Database - this means that it should be created whenever it is necessary. Our system provides a very simple dashboard and an API endpoint where the responsible person can select the metrics that will be analyzed. After inserting a new metric (the user should specify the metric name, the periodicity and the type of computation), the API is called and a new database is created in the Cloud. The node will continue to run this background task that will handle the trips accordingly.

Additionally, the component detailed above is also used to monitor some performance metrics such as:

- CPU
- Memory
- Number of incoming requests
- Number of outgoing requests
- Number of 5xx server errors

These metrics are tracked directly on the Edge nodes and checked every minute. Each minute, the node performs a health check to CPU and memory. Similar to the trip metrics, the user should specify a periodicity for when the data is computed and sent to the cloud. As mentioned previously, it is possible to use the query handler to visualize this data with a bigger periodicity by aggregating the available data. This data can be visualized from the Dashboard, as shown in Figure 3.5.

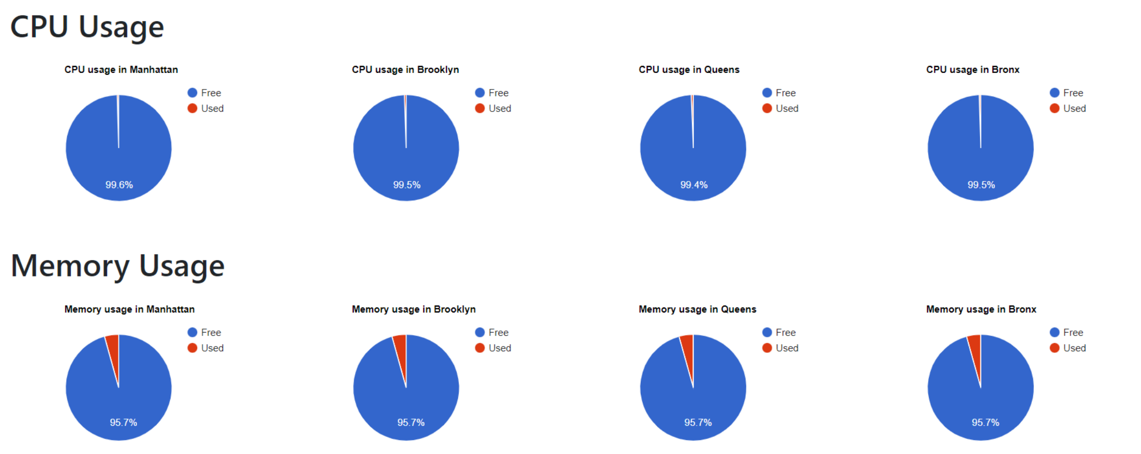


Figure 3.5: CPU and Memory usage visualization

To save memory on the Edge nodes, the data that is transferred to the Cloud from time to time is then deleted, clearing each list in the Dictionary and making the previously allocated space available again. By deleting the information on the Edge nodes, we reduce the necessity of using nodes with high capabilities, reducing the initial investment on the infrastructure.

3.3.2 Threshold manager

The Threshold Manager is a very important component for our system as it is responsible for alerting the developers or system administrators about the health of their infrastructure or some actions/events on the data being received on the nodes.

These parameters are monitored from time to time (the developer must select the periodicity that will be used to monitor the system). For example, if the periodicity is set to 5 minutes, everything will be monitored every 5 minutes (some conditions are aggregated but this is done automatically) and if the threshold is exceeded (for example, 90 percent of memory usage) a notification or alarm is sent to the specified user. In this scenario, the CPU is checked every minute and when the periodicity of 5 minutes is reached, the data is aggregated using Average or Max, depending on the type of situation. After the aggregation, the data is sent to the Cloud and goes through the Threshold Manager, which will check if there is any condition pre-defined for the metric.

Regarding the metrics related to the trips, this module is also very helpful to detect certain patterns. For example, it is possible to track the number of trips that occur during a certain period of time and create an alert that sends an email or adds some information to a database. If the number of trips exceeds a certain value, it is possible to send more drivers to a specific location. We will then detail how these actions are configured.

In order to use the Threshold Manager, we are using the same module mentioned above (Metric Analyzer) which is responsible for analyzing the possible queries that are necessary to query the data mentioned above. All the information regarding requests, server errors, CPU and memory are stored in a Time Series Database and it's possible to use our Data Handler to perform a set of predefined computations. In this specific case we are able to use Count, Max, Min and Average. The Data Handler is a module present in the Aggregate Computation Protocol and will be detailed in the next Section.

Each metric that is tracked by the threshold manager consists of 4 simple items:

1. Condition: The condition that will be checked, such as max, min, count and average. During a specific timeframe (periodicity), the values are aggregated according to this condition.

2. Value: The value of the threshold which will be compared to the aggregated values.

3. Periodicity: Period of time where the data is aggregated.

4. Action: URL of an Azure Function App or Logic App that will be called using a POST request with the information about the event. The action will only be performed if the condition is met. The user can also use this method to call any personalized/custom endpoint.

The condition is checked using a simple a conditional "if" as explained below:

- if (condition(value)): `perform_action()`

If a condition is met, the notification is shown in the Dashboard. If an action is specified to occur after the condition, it will be executed. These actions are implemented using Azure Function Apps and Logic Apps - these apps are hosted on Microsoft's Cloud platform and don't require any management on the infrastructure side. In this case, the user should provide the URL of an Application (Azure Function Apps allows us to create multiple types of triggers such as Timer, Blob Storage, Azure SQL, etc). The HTTP trigger will be called by our system and the action will be performed automatically.

Logic Apps allows users to create and run workflows that integrate with various Azure and third-party services. One common use case for Logic Apps is sending email notifications. The Logic App is triggered using an HTTP request with a JSON payload, containing all the necessary information to write a custom email.

As an example, we created a sample that sends an email to notify a user that the specific action occurred in 3.6.

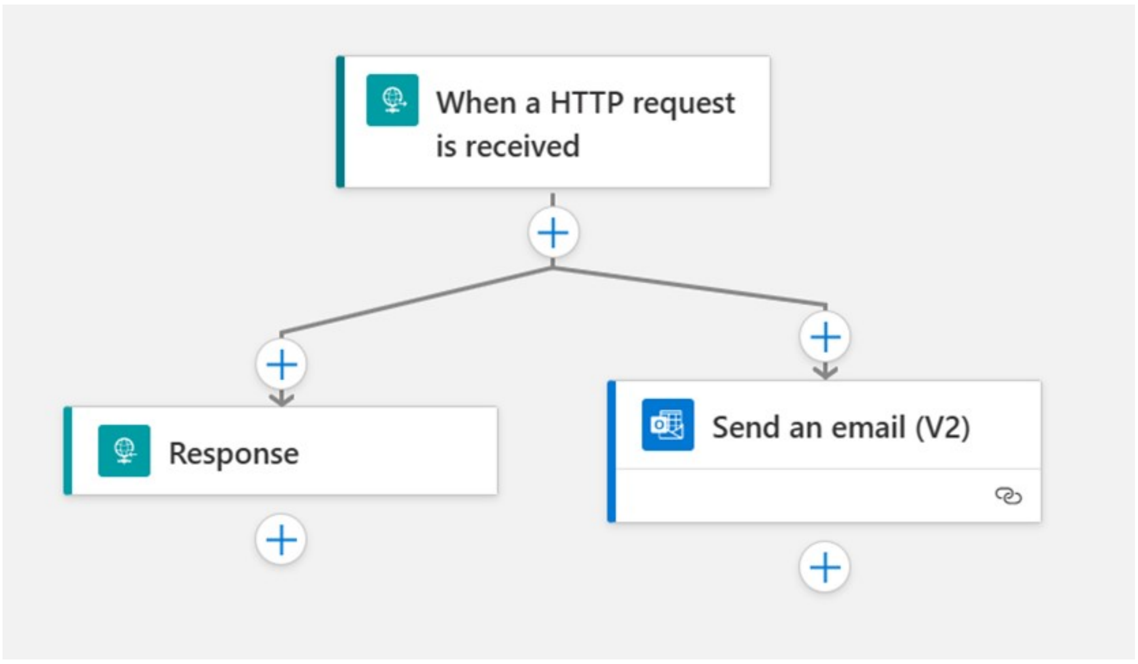


Figure 3.6: Logic App designer

3.4 Aggregate Computation protocol

A computation protocol is responsible for following a set of rules and procedures that manage the way that the data is processed and consumed in a distributed system. Our protocol ensures that the data is processed efficiently and that the results of each computation are consistent and have the same format.

The Aggregate Computation Protocol (ACP) is a distributed protocol that handles data produced by multiple Edge nodes across the network. One of its main benefits it's the ability to perform computation of separate datasets without requiring the need of a single dataset to be stored or transferred across multiple locations. We apply this protocol to our scenario where the idea is to split the computational load across the network of nodes and the cloud.

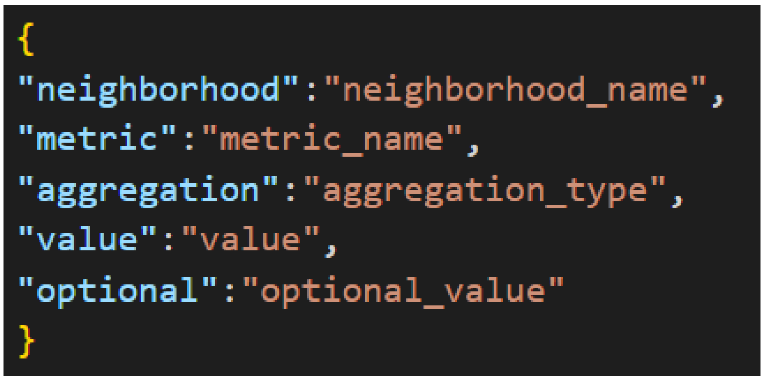
The ACP works by allowing nodes in the network to exchange and combine partial results of the computation, rather than transmitting the entire dataset. This allows for the computation to be performed in a decentralized manner, with each node contributing its own resources and capabilities to the process. Another benefit of this protocol is that it is independent of the number of nodes in the system, making it flexible to changes in the network such as adding or removing nodes.

We will follow by detailing the components of this protocol. Starting with the Data Handler, which receives data from the Edge nodes and the Query handler, which performs specific computations on existing databases, producing new ones.

3.4.1 Data Handler

The data handler is allocated in the Cloud and is responsible for multiple tasks related with handling the data that is sent by the Edge. This component is responsible for receiving aggregated data from the Edge nodes and placing it in a specific time series database for storage and further processing.

As mentioned previously, the Tracking module, which is present in the Edge nodes, is responsible for taking the data produced over time and aggregating it locally during the periodicity set by the user. When the periodicity is reached, the data is aggregated after a GET request is sent from the Cloud to the Edge, which returns a JSON response in a simplified way, using the following structure as shown in Figure 3.7.



```
{
"neighborhood":"neighborhood_name",
"metric":"metric_name",
"aggregation":"aggregation_type",
"value":"value",
"optional":"optional_value"
}
```

Figure 3.7: JSON example

The “value” should contain the value of the metric that was tracked during the set periodicity. Some aggregation types require an additional value which will be present in the “optional” placeholder. The average, for example, is a value that depends on the number of events if we want to perform additional aggregations to it.

This way, the Edge nodes don’t need to be aware of the current time as the requests are performed by the Cloud. After receiving the information from each Edge node, the aggregated value is complemented with the current time (using the datetime module for Python) and added to the database.

This module has knowledge about the existing Time Series Databases and tables created by the monitoring module and will then add the received aggregated data into the correct TSDB. During this stage, each node (that represents a different region) has its own separate TSDB. This means that the Data Handler doesn't perform any type of computation on the data, it simply routes it to the correct place.

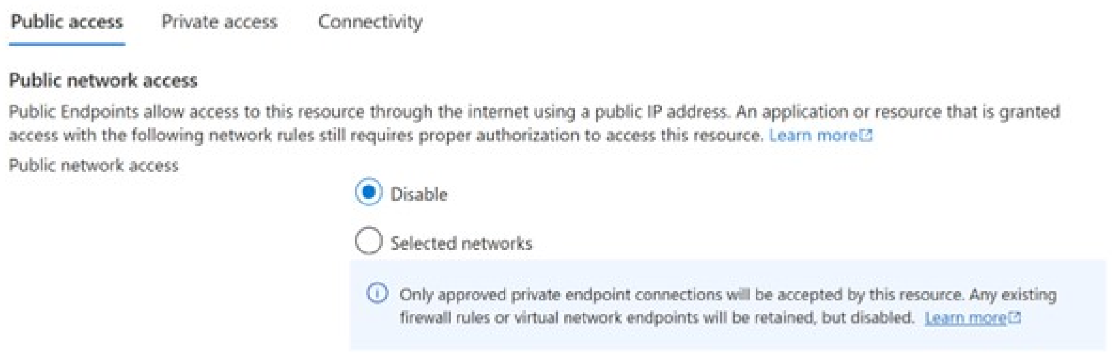
Although it is possible to aggregate the data in the monitoring module in different ways (such as max, min, count, average), each TSDB that is created is named using the neighborhood name as bellow:

- neighbourhood, for example: Manhattan

Each database stores all the aggregated data for the specific node and it is possible to query it using multiple conditions. The connection to the Database is made using the Pyodbc Python module. This module creates a connection to the DB using its server address, the database name, a username and a password. Additionally, this module requires that the hosting machine has the necessary drivers (in this case ODBC Driver 17 for SQL Server):

- conn = pyodbc.connect(DRIVER;SERVER;PORT;DATABASE;UID;PWD)
- cursor = conn.cursor()

For testing purposes, that will be explained in Chapter 4, for the Cloud, the database is hosted on Azure using SQL Server and is protected using network restrictions shown in Figure 3.8.



Public network access

Public Endpoints allow access to this resource through the internet using a public IP address. An application or resource that is granted access with the following network rules still requires proper authorization to access this resource. Learn more

Public network access

[✓ Disable]

Selected networks

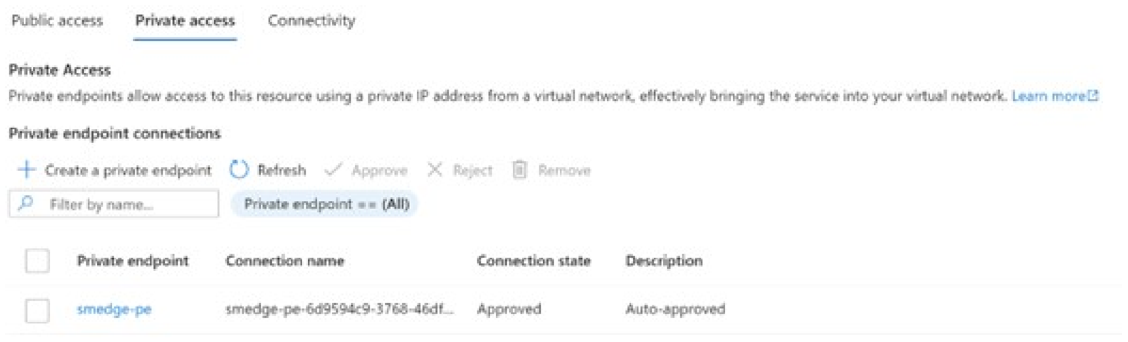
Only approved private endpoint connections will be accepted by this resource. Any existing firewall rules or virtual network endpoints will be retained, but disabled. Learn more

Figure 3.8: Network restrictions enabled in the Portal

By disabling public access, the service that connects to the database needs to be inside a Vnet peered with a Private Endpoint on the database as shown in Figure 3.9.

For the simulation of our solution, the database was deployed under the Standard Pricing Tier which is very cost efficient (there are pricing tiers that provide better performance) and has a total storage space of 250GB.

As we will compare the use of Centralized and de-Centralized systems, we have also created a local database, that will be hosted and located inside the respective container. This file is named Cloud.db and we use the database management system SQLite to query



Private Access

Private endpoints allow access to this resource using a private IP address from a virtual network, effectively bringing the service into your virtual network. Learn more

Private endpoint connections

Private endpoint	Connection name	Connection state	Description
smedge-pe	smedge-pe-6d9594c9-3768-46df...	Approved	Auto-approved

Figure 3.9: Private endpoint creation

it. The connection to this database is much simpler as it only required the use of sqlite3 Python module and the name of the file for connection. As the file is in the same location as the Python script, there are no network or security restrictions and the connection should be immediate.

The data handler module is responsible for receiving and storing data from Edge nodes in a time series database. It ensures that the data is organized, accurate, and ready for further processing and analysis. This allows for efficient and effective use of the data collected by the edge nodes.

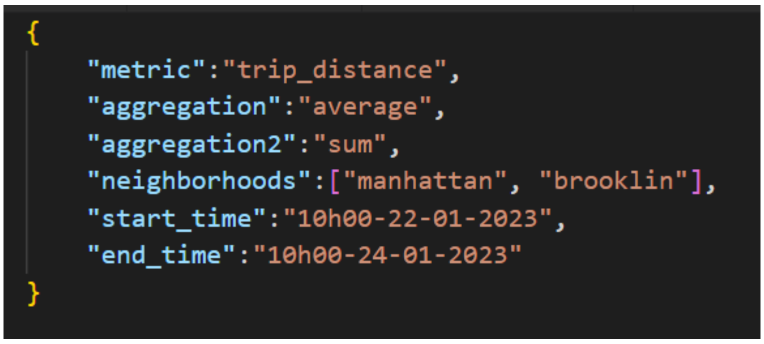
In the next Section we will detail how this data is queried and handled, optimizing each resource in the system.

3.4.2 Query Handler

The Query Handler is a software component that is responsible for querying time series databases and producing new ones with aggregated or computed values. It allows users to access and analyze the data stored in the time series databases in a flexible and efficient manner. As we previously mentioned, this system was designed to analyze data produced by multiple vehicles across multiple regions that are represented by the different Edge nodes. It is possible to have thousands of requests per day which are handled separately. However, it is important to be able to query the data by aggregating it. In this Section we will explain how the Query Handler receives requests and returns a result.

Requests to the query handler are initiated by the user from the Dashboard or directly by making a GET request, this means that our API will receive the request and then make the request to the query handler. This request contains the name of the metric, the type of aggregation, the second aggregation, the selected neighborhoods and the time frame, as shown in the example shown in Figure 3.10.

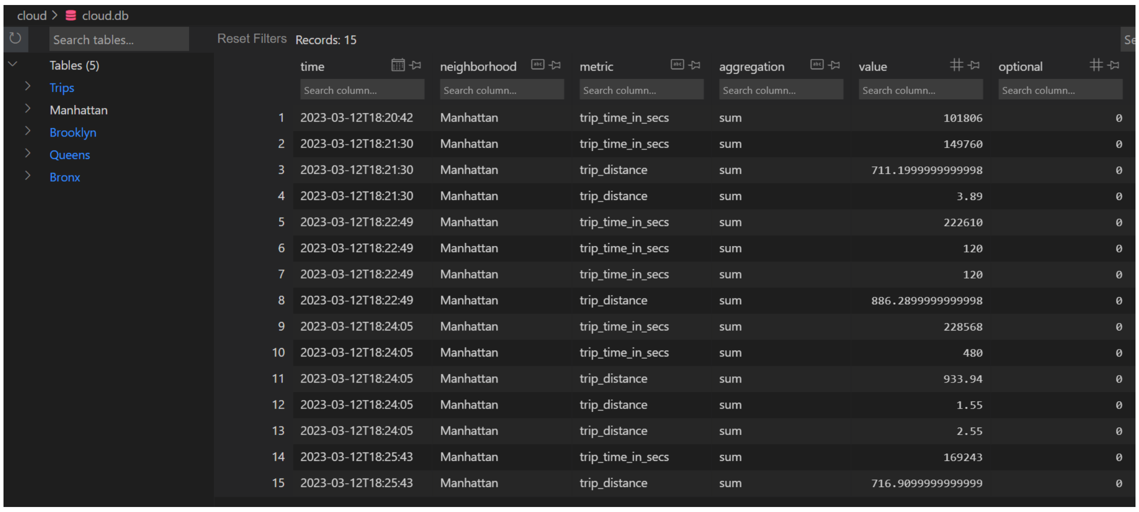
The query handler will then look at the existing TSDB used by the Data Handler, retrieving the appropriate database(s). If the requested information isn't stored in the system (metric not being tracked or time frame not existent) an error will be returned to



```
{
    "metric":"trip_distance",
    "aggregation":"average",
    "aggregation2":"sum",
    "neighborhoods":["manhattan", "brooklin"],
    "start_time":"10h00-22-01-2023",
    "end_time":"10h00-24-01-2023"
}
```

Figure 3.10: JSON payload example

the API with that information. It is important to remember that the first aggregation selected must be pre-selected for tracking and will be computed directly on the Edge nodes, before reaching the cloud. In Figure 3.11 we can visualize an example of the different Tables present in the Cloud database, each representing a different neighborhood. In this example, we can see that the metrics trip_time_in_seconds and trip_distance are being tracked with the aggregation "sum". This data was previously computed on the Edge nodes during the specified periodicity.



	time	neighborhood	metric	aggregation	value	optional
1	2023-03-12T18:20:42	Manhattan	trip_time_in_secs	sum	101806	0
2	2023-03-12T18:21:30	Manhattan	trip_time_in_secs	sum	149760	0
3	2023-03-12T18:21:30	Manhattan	trip_distance	sum	711.1999999999998	0
4	2023-03-12T18:21:30	Manhattan	trip_distance	sum	3.89	0
5	2023-03-12T18:22:49	Manhattan	trip_time_in_secs	sum	222610	0
6	2023-03-12T18:22:49	Manhattan	trip_time_in_secs	sum	120	0
7	2023-03-12T18:22:49	Manhattan	trip_time_in_secs	sum	120	0
8	2023-03-12T18:22:49	Manhattan	trip_distance	sum	886.2899999999998	0
9	2023-03-12T18:24:05	Manhattan	trip_time_in_secs	sum	228568	0
10	2023-03-12T18:24:05	Manhattan	trip_time_in_secs	sum	480	0
11	2023-03-12T18:24:05	Manhattan	trip_distance	sum	933.94	0
12	2023-03-12T18:24:05	Manhattan	trip_distance	sum	1.55	0
13	2023-03-12T18:24:05	Manhattan	trip_distance	sum	2.55	0
14	2023-03-12T18:25:43	Manhattan	trip_time_in_secs	sum	169243	0
15	2023-03-12T18:25:43	Manhattan	trip_distance	sum	716.9099999999999	0

Figure 3.11: Cloud database with different Tables representing the Edge nodes

After the correct Time Series Databases/Tables were selected (the ones representing each Edge node), a new aggregation process will start as follows:

1. Each table that is queried represents a different Edge node (neighborhood). The query contains the metric, aggregation, `start_time` and `end_time`.

2. As the periodicity for each metric with a specific aggregation type is the same for every node, the result of the query will have the same length.

3. The results of each table are sorted according to their introduction time and added to a separate list.
4. A new aggregation is performed to each index of the list (max, min, count, etc) and sent to the final list to be displayed.
5. By following the same procedure, we are able to uniform the data that is sent to the user.

Uniforming the output data of a system is an important step in ensuring that the data is accurate, consistent, and ready for further analysis. This process involves standardizing the format, structure, and content of the data to make it easier to work with and understand. Another important benefit of uniforming the output data is that it improves the accuracy of the data. Data that is not uniformed can contain errors, inconsistencies, or missing values, which can lead to incorrect conclusions or poor decision making. By standardizing the data, it is possible to identify and correct these errors, improving the overall accuracy of the data.

The new data is generated following the same pattern as the input data and sent back to the client. In our example, this data is displayed in a linear graph, representing the different time spans.

The query handler module is a powerful tool for querying and analyzing time series databases. It allows the system to access and analyze the data stored in the time series databases in a flexible and efficient manner and also to produce new time series databases with aggregated and computed values. This allows for more in-depth analysis and insights of the stored data.

3.4.3 Summary

In this Section we presented the Aggregate Computation Protocol as a method for performing calculations on a group of data, such as averaging or summing the values. This can be useful in a variety of contexts and an aggregation can be performed on other aggregations, splitting the computational load across multiple nodes and devices. The use of aggregate computation protocols allows for efficient and accurate processing of large amounts of data in parallel.

The Data Handler module is responsible for receiving aggregated data from edge nodes and placing it in a time series database. This module selects the appropriate time series database according to their names.

The Query Handler module is responsible for querying time series databases and producing new ones with aggregated values. It allows users to access and analyze the data stored in the time series databases in a flexible and efficient manner. This module allows users to visualize data with a different periodicity and for a specific time frame.

The Aggregate Computation Protocol allows for the efficient and accurate processing of large amounts of data, making it possible to perform complex calculations, detect anomalies and trigger alerts, monitor and analyze time series data, and provide a more efficient and effective way to monitor and analyze systems. The use of data handler and query handler modules allows for the efficient storage, retrieval and analysis of the data.

3.5 API

The API is one of the most important parts of our solution and its purpose is to expose all methods and functionalities of the remaining parts, which works combined to provide a final result. The API will connect the client to the server (in this case, our modules) using a set of operations that were modeled in a flexible way.

In this Section, we will detail the API, summarizing its operations and functionalities.

3.5.1 Overview

Our API combines a set of protocols and works as a point of contact between the data source and the data target. As we aimed to design a flexible API for future developers and we designed a visual interactable page, we decided to follow a RESTful (Representational state transfer) architectural style.

The API was developed using the Python language (version 3.9.11) due to its simplicity and the support provided by the overall community. We integrated Flask (2.2.2), a simple framework that was chosen for being lightweight, easy to adapt and very well documented.

The communication is performed using the HTTP protocol as this is one of the most common client-server protocols and can be easily adapted to day-to-day devices. This application layer protocol is sent over TCP (Transmission Control Protocol) which allows two hosts to connect and exchange data. In our scenario, the requests are sent by three types of entities:

- Taxis or vehicles that are part of the platform: These vehicles have a device with some computational power and send information to the API exposed by the Edge nodes.
- The Cloud: Although the Cloud has different functionalities from the Edge nodes, it will also contact and exchange information with the mentioned API.
- Other Edge nodes: Each node has the same API deployed and as the whole system works as a combined system, they must communicate with each other to guarantee the reliability of the whole environment.

Each client that uses the API is aware that it exists and that it is exposed by each specific Edge node. Whenever a new node is added, the developer must introduce its location (either by selecting a neighborhood or providing coordinates with latitude and longitude).

After selecting the most appropriate node, each client will exchange data with the servers and vice versa using the JSON data format. JSON (JavaScript Object Notation) was chosen due to its simplicity and because it is very lightweight, allowing us to transfer multiple types of objects with the same format. After the server receives the requests from the different types of clients, it will return a response. Depending on the response (it may be successful or not) the data will be processed or the monitoring module will manage the different servers.

Each time that an endpoint is called, either it is successful or not, it will be logged into a list for further analysis if necessary. Each log contains the endpoint name, the status code (HTTP) and a simple description of the event.

The API is implemented and deployed throughout the Edge nodes and the Cloud but there are certain operations that are specific to each of these locations due to its characteristics and functionalities.

With the above information in mind, we will then detail and give an overview of the different API operations.

3.5.2 API operations

As mentioned in the previous Section, we will now detail the different operations supported by our API. This API is deployed both on the Edge and the Cloud and some functionalities are specific for its location:

Available in the Cloud

1. Add and remove Edge nodes: This is the first and one of the most important operations that is exposed by the API. Every time that a new system is created, the developer should collaborate with the team responsible for the infrastructure of the Edge nodes and use the API to add them to the system by specifying its IP address and the location. The location should be generalized as an area such as a neighborhood or a city. This operation can also be used to remove existing Edge nodes if necessary.

2. Add a new metric analyzer: This endpoint requires the following input: metric name (trip_distance, trip_time, etc), computation type (max, min, count, average) and periodicity (number of minutes). After sending a POST request to the endpoint, all nodes will start to monitor the specified metric and aggregate the values accordingly - this endpoint contacts each node to inform about this situation.

3. Search for existing metric analysis: After a metric is inserted, it will be added to the list of tracked metrics. This endpoint is used mostly by the dashboard interface (which will be explained in detail in the following sections) and returns the list of tracked metrics.

4. Get for metric analysis: After adding some metrics for analysis, it is possible to query it accordingly by using this endpoint. The user should use as input the name of the metric and, as a result, the data will be returned to the GET request and used, for example, to fill a linear chart. As this data is an aggregation of all the data available throughout the nodes, it isn't necessary to contact them. As the metric analysis contains a periodicity value, it isn't possible to query it immediately as it isn't always available in the cloud.

5. Receive metric analysis: Each metric analyzer is inserted with a specific periodicity which means that only after some minutes it will be propagated from the nodes to the cloud. This endpoint is used by the nodes to send their computed analisis for a specific metric during the specified periodicity.

6. Set threshold: Using the threshold component mentioned in the previous Section, it is possible to use this operation to manage all types of alerts. It's required to specify the metric to be analyzed (such as CPU or memory), as well as its threshold value (in percentage). Additionally, as each node is independent and has different types of loads, the user should select the node for which the threshold manager will be applied. This endpoint will then contact the node directly using an endpoint mentioned in the next Section.

7. Query data (SQL): as data is replicated to the cloud, we allow users to query it using SQL queries. This action is performed directly in the main database, not the Time Series DBs.

8. Get trip flows: This endpoint was designed specifically for this example where ride sharing data is produced. It queries the database for a specific time frame set by the user and returns the trip flows between the different neighborhoods.

9. Get logs: All requests made to the API are logged into a simple list that can be converted to text. This list contains the time of each event and a simple description with the HTTP status code.

Available on the Edge Nodes

1. Start a new trip: This operation is used by the clients (in this case taxis or ridesharing vehicles) and it is used everytime a new trip is started. This operation requires the location of the vehicle, a medallion (vehicle ID, similar to a license plate which is unique for each vehicle) and the current time. The API will automatically register the trip, adding it to the local database to be handled in the future. When a new trip starts, the taxi is registered as unavailable for other trips.

2. Finish a trip: Similar to the previous operation, this is also performed by the clients but everytime a trip is ended (reaches the final location). This operation requires some additional information from the client such as:

- Trip ID that was created previously
- Medallion
- Pickup latitude
- Pickup longitude
- Drop Off latitude
- Drop Off longitude
- Trip time in seconds
- Trip distance
- Total amount in dollars

After the trip is over, a new object is created with all the information and saved in the database for future processing. It is possible to perform specific computations at this step that can be implemented by the developer - this operation is mentioned above in the Cloud operations Section.

3. Add a new metric analyzer: this endpoint requires the metric name, the aggregation type and periodicity. When the endpoint is called using a POST request, the specified metric will start being tracked. The endpoint is called by the cloud simultaneously to all the edge nodes.

4. Check CPU usage: Each node has its own separate hardware composed of independent memory(RAM), CPU and storage. With this endpoint, it's possible to check at any time what is the current CPU usage in percentage for a specific node.

5. Check memory usage: Similar to the previous operation, it is also possible to check the current memory usage in percentage for a specific node.

6. Health Check: As we mentioned previously in the monitoring module, each node can be tracked using the threshold manager. Whenever a condition is met, it will be stored on a list. This endpoint can be used with a GET request to query this list. The user doesn't have to add any parameter.

7. Ping: Simple GET request to the root of the node to check if it is up and running. Returns HTTP 200 if operating normally.

8. Get logs: All requests made to the API are logged into a simple list that can be converted to text. This list contains the time of each event and a simple description with the HTTP status code.

3.5.3 Summary

In this Section we presented an overview of our API, which is one of the modules that are used to run our solution. We presented a list of the several available operations that are exposed to the clients and the other nodes in the system.

3.6 Dashboard

As the idea of this whole system was to make data processing and analyzing more efficient, easy to use and user friendly, we decided to develop a simple interface which contains a dashboard that is capable of managing all the possible settings and configurations of the nodes, as well as visualizing graphs and flow maps.

This dashboard was designed using simple HTML and Javascript and it communicates with the API mentioned previously to receive and send all the necessary information. This dashboard can be used by the system administrator or anyone who wants to analyze the data that is being processed. By using a dashboard to call an API, users can access and visualize data from different sources in a unified and user-friendly way. This allows for more efficient and effective decision making, as well as a better understanding of the data.

In the next topic we will provide a brief overview of all the functionalities and the technologies used.

3.6.1 Overview

Our Dashboard allows users to monitor, visualize and interact with operations available from the API. One important task is to make sure that the Edge nodes are available, as well as their current status, including the CPU and Memory usage. We decided to implement circular charts representing the usage percentage and simple circles with different colors representing the node (the API pings each node and expects to receive an HTTP 200 response), as shown in Figure 3.12.

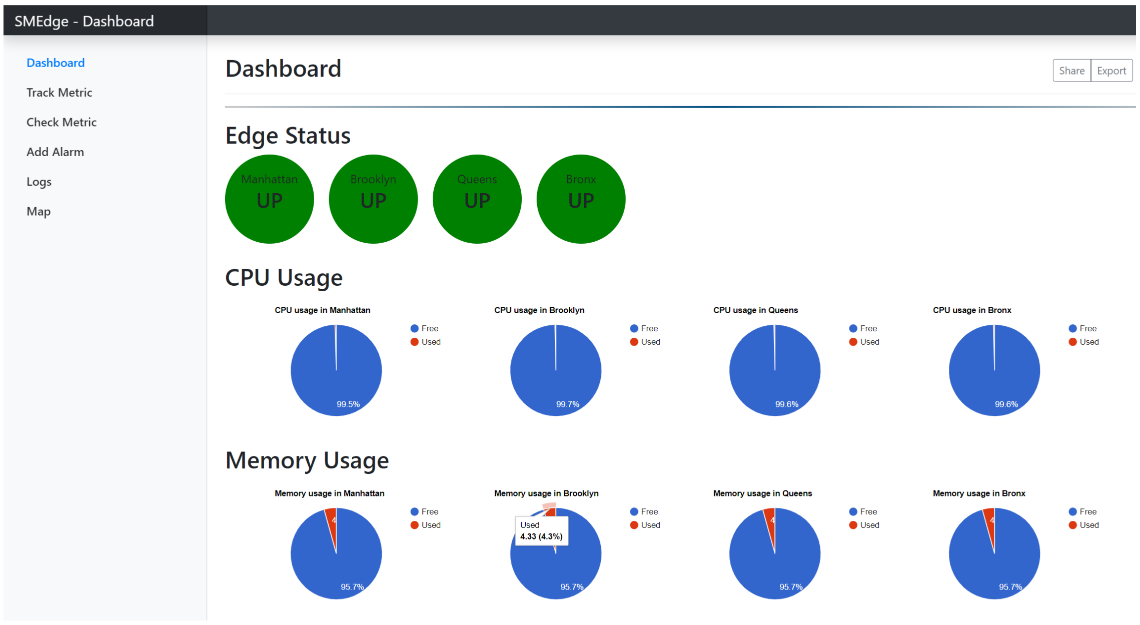
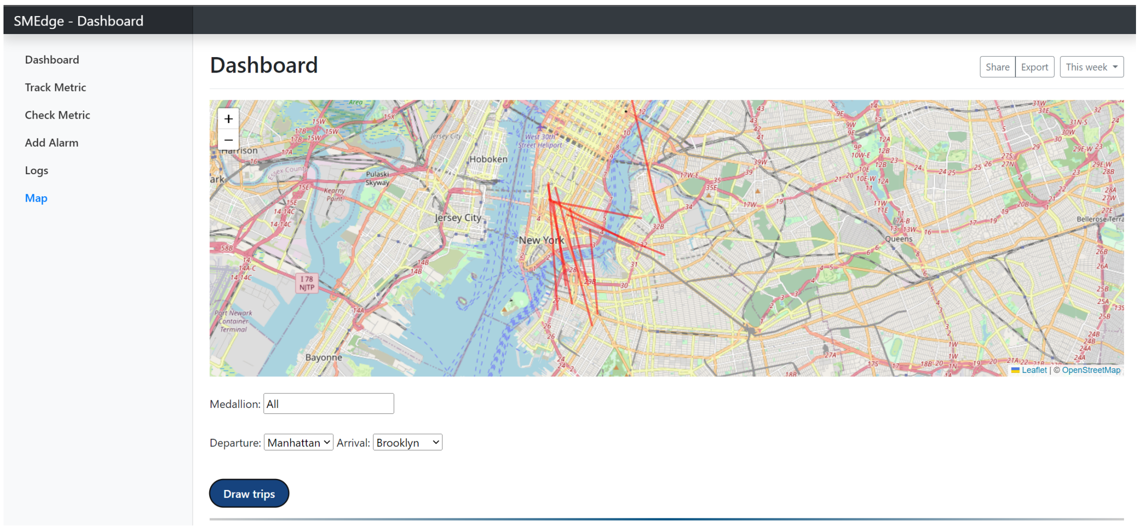


Figure 3.12: Edge nodes status in the Dashboard

This system was designed for analyzing data produced by ride sharing vehicles that produce data with a sense of location (using latitude and longitude). One of the important tasks is to be able to visualize the flows of trips (meaning, for a specific timeframe, where the passengers are traveling from and to). This was possible to achieve by implementing an open-source JavaScript library called Leaflet - it is very important to mention that the whole library only takes 42 KB of memory, making it very lightweight. Leaflet is used for mobile and desktop and has a lot of examples available on their documentation, making it very easy to use and implement.

With Leaflet, the user should select a timeframe in the dashboard and it will be presented with a map with the trips started and finished during that period across the whole geo-location (this applies to all the nodes in the system). Additionally, the user can select a departure and/or arrival neighborhood, as well as filtering the trips provided by a specific medallion as shown in Figure 3.13.



Medallion: All

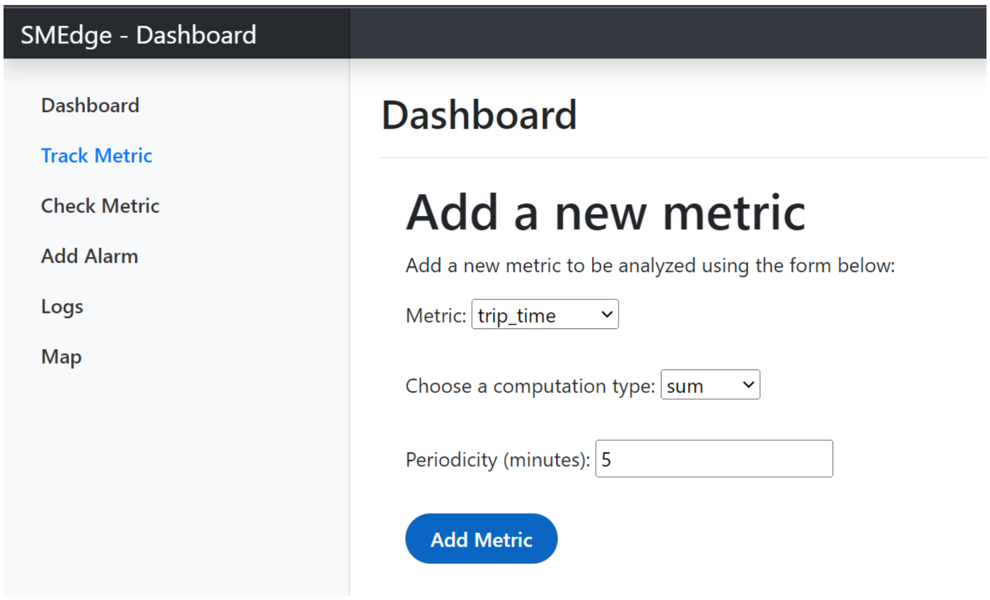
Departure: Manhattan Arrival: Brooklyn

Figure 3.13: Trips displayed in the Map

This feature makes a GET request to the API using the /get_trips endpoint, using the input provided by the user. This endpoint will query the internal database for existing Trips and return the list to the frontend.

Another feature of the dashboard is the possibility to manually add new metrics to be analyzed. The dashboard uses the API to query all the available metrics (excluding the ones that are already being tracked). After selecting the metric, the user should choose the computation type (count, max, min, average) and the periodicity (in minutes), as shown in Figure 3.14. This feature uses the endpoint /global_track_metric which pings each node inside the network, marking this metric as traceable and setting its periodicity.

After the metrics start being tracked, as mentioned in the previous sections, an initial computation will be performed directly on the Edge nodes. This computed metric will be sent to the Cloud according to its periodicity. Using the tab shown in Figure 3.15 , the users should select a metric, the initial aggregation, the seconds aggregation (to be



Dashboard

Add a new metric

Add a new metric to be analyzed using the form below:

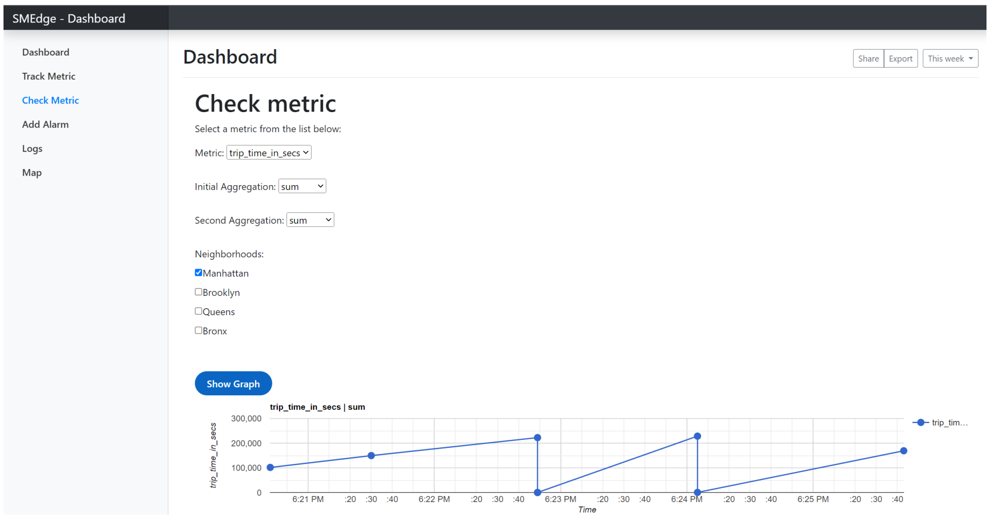
Metric: trip_time

Choose a computation type: sum

Periodicity (minutes): 5

Figure 3.14: Start tracking new metric from the Dashboard

performed in the cloud to the already computed metric) and the neighborhoods. This feature calls the endpoint /get_analyzed_metric that collects the metrics from the different selected neighborhoods and performs an additional aggregation to this data.



Dashboard

Check metric

Select a metric from the list below:

Metric: trip_time_in_secs

Initial Aggregation: sum

Second Aggregation: sum

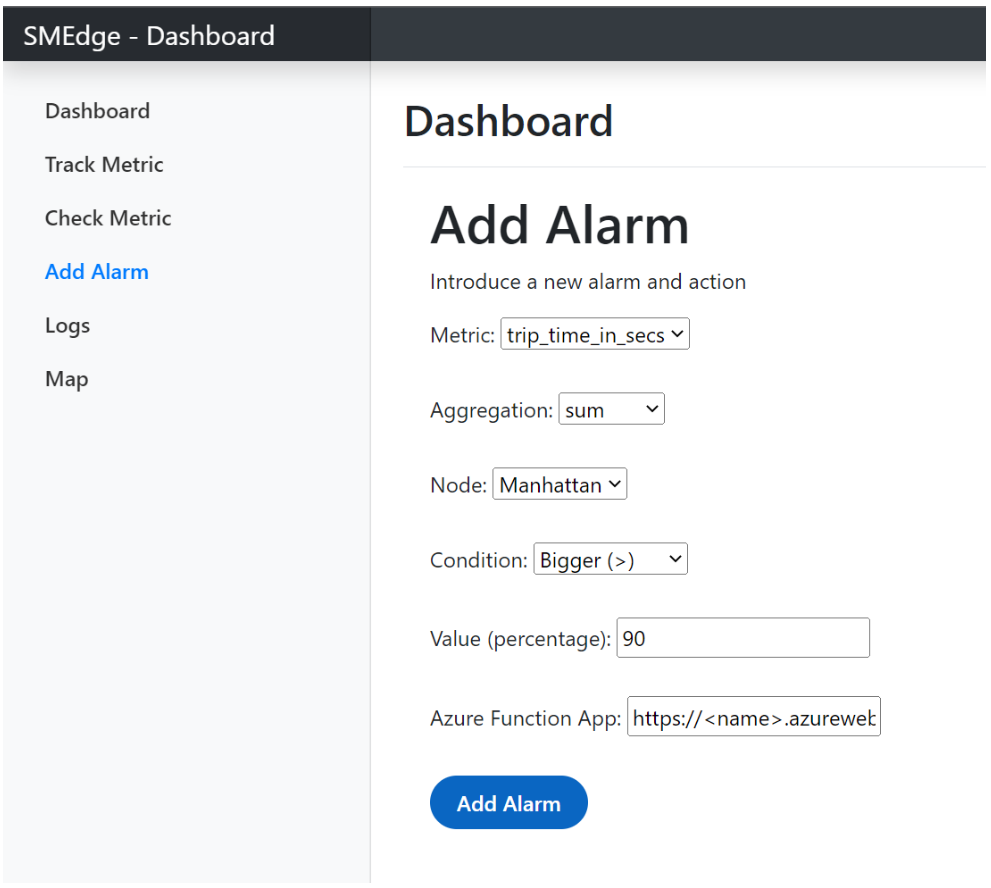
Neighborhoods:

☑Manhattan
☐Brooklyn
☐Queens
☐Bronx

Figure 3.15: View tracked metric from the Dashboard

The Graphs were implemented using Google Chart tools - this feature is free to use and allows us to customize multiple types of data visualizers and implement them in HTML pages with cross-browser compatibility without the need of any type of plugins.

As mentioned in previous sections, the system allows users to monitor and track system metrics such as CPU and memory so we implemented an interface for this feature as well. It is possible to set an threshold limit for a specific node and metric, setting a custom value and an action to be performed in case of the event being triggered, as shown in Figure 3.16.



Introduce a new alarm and action

Metric: trip_time_in_secs

Aggregation: sum

Node: Manhattan

Condition: Bigger (>)

Value (percentage): 90

Azure Function App: https://<name>.azureweb

Figure 3.16: Define a new action from the Dashboard

3.7 Design and Implementation

The prototype of our solution was developed using the Python programming language. This was mainly a choice motivated by personal preference, but a valid argument would also be that Python has a lot of benefits that makes it useful when handling big amounts of data:

1. Easy to use libraries: Python has a big collection of libraries that can be used to implement new features and functionalities, including data analysis, machine learning, web development, and more. In our system we resorted to libraries such as pyodbc, docker, sqlite, requests, etc.

2. Speed and performance: Python is a fast and efficient programming language, and it provides several tools and libraries that can be used to optimize performance. This makes it a good choice for developing a system that needs to perform well, even when working with large amounts of data.

All the services were implemented using HTTP backends and we chose Flask for the development of our API as it is very lightweight, scalable and flexible, allowing us to define custom endpoints and manage the requests and responses however we desire.

Docker was used as the container runtime for the Edge nodes specially for its portability: Docker containers can run on any system that has Docker installed, regardless of the underlying infrastructure. This makes it easy to move applications from development to production, and to deploy them on different environments, such as physical servers, virtual machines, or cloud-based platforms. This makes it possible to use different types of machines to host our Edge nodes across the city.

To store the data we used time series databases as our data is time stamped, meaning that we perform queries based on the time of each event. Time series databases are specifically designed to store time-stamped data, which means that they are optimized for handling large amounts of time-stamped data efficiently. This can lead to faster query times and reduced storage requirements compared to traditional databases.

As each metric is added to the database specific for each Edge node location, it is important to name and store the databases correctly. This means that whenever it is necessary to query a specific database, the system will be aware of its exact name.

The prototype includes the features and functionalities described previously, including discovery and track of other neighborhood nodes, aggregation and de-aggregation protocols and data transfer across the edge and the cloud. Additionally, we implemented some tools that will help us troubleshoot and benchmark all the different features and tasks.

3.8 Summary

In this Chapter we discussed the implementation of SMEdge, a solution designed to handle large amounts of data and perform complex operations in a fast and efficient manner between the Edge and the Cloud.

We began by detailing our system that consists of 4 main different components:

1. Aggregate computation protocol: This protocol performs aggregations on time series databases and produces new time series databases. It uses data handler and query handler modules to manage the data and perform operations on it.

2. Tracking Module: This module monitors different metrics and performs actions under certain conditions set by the users.

3. API: This component provides an interface to access the data and perform operations on it. A dashboard can be used to call the API and retrieve the data.

4. Dashboard: A simple interface to visualize data and make requests to the API.

We then followed by a small overview of its Design and Implementation, explaining our choices along the creation of SMEdge.

In the next Chapter, we will detail and present the experimental work done to evaluate our system and compare it to current scenarios.

4 Evaluation

In this chapter, we present the results of the experimental work carried out to evaluate the performance of our solution in comparison to traditional scenarios. The primary objective of this evaluation is to assess whether our approach can deliver better results compared to existing solutions. We conduct various experiments to measure the performance metrics such as latency, throughput, and resource utilization. These metrics are crucial in assessing the effectiveness and efficiency of our system. By analyzing the experimental results, we aim to provide insights into the behavior and capabilities of our system and identify areas for improvement. Ultimately, the goal of this evaluation is to validate the effectiveness of our solution and demonstrate its potential benefits.

We begin by describing the considerations behind the choice of model for our underlying network (Section 4.1), followed by the experimental setup (Section 4.2), and finally a discussion of the obtained results (Section 4.3).

4.1 Network

In order to simulate the real world environment, which has an undetermined and varying latency between the multiple nodes that work together, we implemented a middle-ware that will simulate the latency during the communication between the nodes while running in our local environment. By using this middle-ware, we can test and evaluate the performance of our solution under more realistic conditions, and make more informed decisions on how to optimize it for real-world deployment.

Latency is the delay or time lag between the transmission of a request and the receipt of a response. It is a critical factor in the performance of networked systems, particularly those that involve distributed computing across geographically dispersed locations. Latency is directly related to distance, as the time taken for a signal to travel between two points increases with the distance between those points. During the next paragraphs, we will calculate latencies taking into consideration as an example a simple request that doesn't require any type of serious computation on the server.

In the context of distributed computing, latency can have a significant impact on the

Location	IP	RTT (ms)	Distance (km)	Time per 100km
New York	157.230.229.24	100	5384	1.857355126
Singapore	188.166.213.141	263	11888	2.212314939
Moscow	31.177.80.4	75	3906	1.920122888
Amsterdam	51.15.53.77	42	1847	2.273957769

Table 4.1: Ping tests across the world

performance of the system. As data is transmitted between nodes in the network, the delay introduced by latency can slow down the processing of requests and the transmission of results. This can result in a decrease in the overall throughput and efficiency of the system.

As we will developed in the past sections, there are multiple edge nodes, each responsible for a specific zone. Our latency will be calculated taking into consideration the distance between the data producer and the edge node and the average speed. This distance should have an almost direct relation with the latency. Bellow is a table with the results of pinging IP addresses from across the world (always from the same location - Lisbon), as shown in Table 4.1.

Taking the table shown in 4.1 into consideration, we can perform an average to the time per 100km (in milliseconds) as shown in Figure 4.1.

$$\frac{1{,}86 + 2{,}21 + 1{,}92 + 2{,}27}{4} = 2,065\ ms$$

Figure 4.1: Average on calculated latency's per 100km

Although latency is a very important topic, it can only be achieved if there is sufficient bandwidth between the edge nodes to enable the transmission of large amounts of data.

If the bandwidth between edge nodes is insufficient, it can lead to delays in data transmission, resulting in increased latency and slower response times. This can have a significant impact on the overall performance of the system and can even render it ineffective.

For testing purposes, we used Docker to host multiple edge nodes in different containers and configure them to use different ports to communicate with each other. This approach allows us to simulate a distributed environment with multiple edge nodes running in different containers and communicating with each other over different ports.

We will detail in the next section how the latency and bandwidth will be applied in our experimental setup.

4.2 Experimental Setup

In order to test the performance and functionality of the SMEdge framework, we conducted experiments using a local setup. Our experimental setup consisted of a single machine acting as a central server and multiple edge nodes that simulated a distributed network. All nodes were hosted on the same machine to but we manually added some latency to simulate a real network environment.

To test the system, we created a Python script that simulates the different vehicles traveling around the map, picking and dropping passengers. This script has information about the trips start and end and will contact the Edge API for the closest node to its location. This script sends the data in real time, meaning that the system will be evaluated with the same load as in real life.

We used Docker containers to simulate the edge nodes, and each container had an instance of the SMEdge framework running on it. The central server also had an instance of the SMEdge framework running on it, which acted as the query handler and data handler module. We used Flask to develop the API and sqlite to connect to the time series database hosted on the same container as the nodes. We created several Docker containers on a single machine, each running the same software stack that would run on an actual edge device. By configuring the containers with different network settings, we can simulate the latency and bandwidth limitations that would be present in a real-world edge network. We will detail later in this section how we added latency to each container.

To simulate a real-world scenario, we have a dataset of real taxi trips taken in New York City during the year of 2013. This dataset was gathered from the NYC OpenData library [30] but we decided to shorten it to contain only 200,000 entries (which represents around 20 continuous days). This dataset contains metrics such as speed, distance, and cost, which were used to test the tracking module and the Aggregation Protocol. We also tested the threshold manager module by setting up alerts for when certain conditions were met.

To test the performance of the SMEdge framework, we measured the response time of the API when querying the time series database. We varied the number of edge nodes in the network and changed the type and size of the batches used to send the data to observe the impact on response time.

As mentioned previously, each node represents a neighborhood. For our experimental setup, we will take into consideration that all vehicles in a specific neighborhood are at the same distance from their respective node and from the nodes in the other neighborhoods. In the following image, we will detail how the distances and latencies will be applied to each vehicle connecting to a specific node. For this example, we will take into consideration that the average latency per 100km is 2,065ms as calculated in the previous section.

We gathered the distance between each node and created a simple table which is shown in Figure 4.2 and a map in Figure 4.3 to display this information.

Neighborhoods	**Manhattan**	**Brooklyn**	**Queens**
Brooklyn	13,20 km		
Queens	16,77 km	14,59 km	
Bronx	10,87 km	21,86 km	16,11 km

Figure 4.2: Distances between Edge nodes

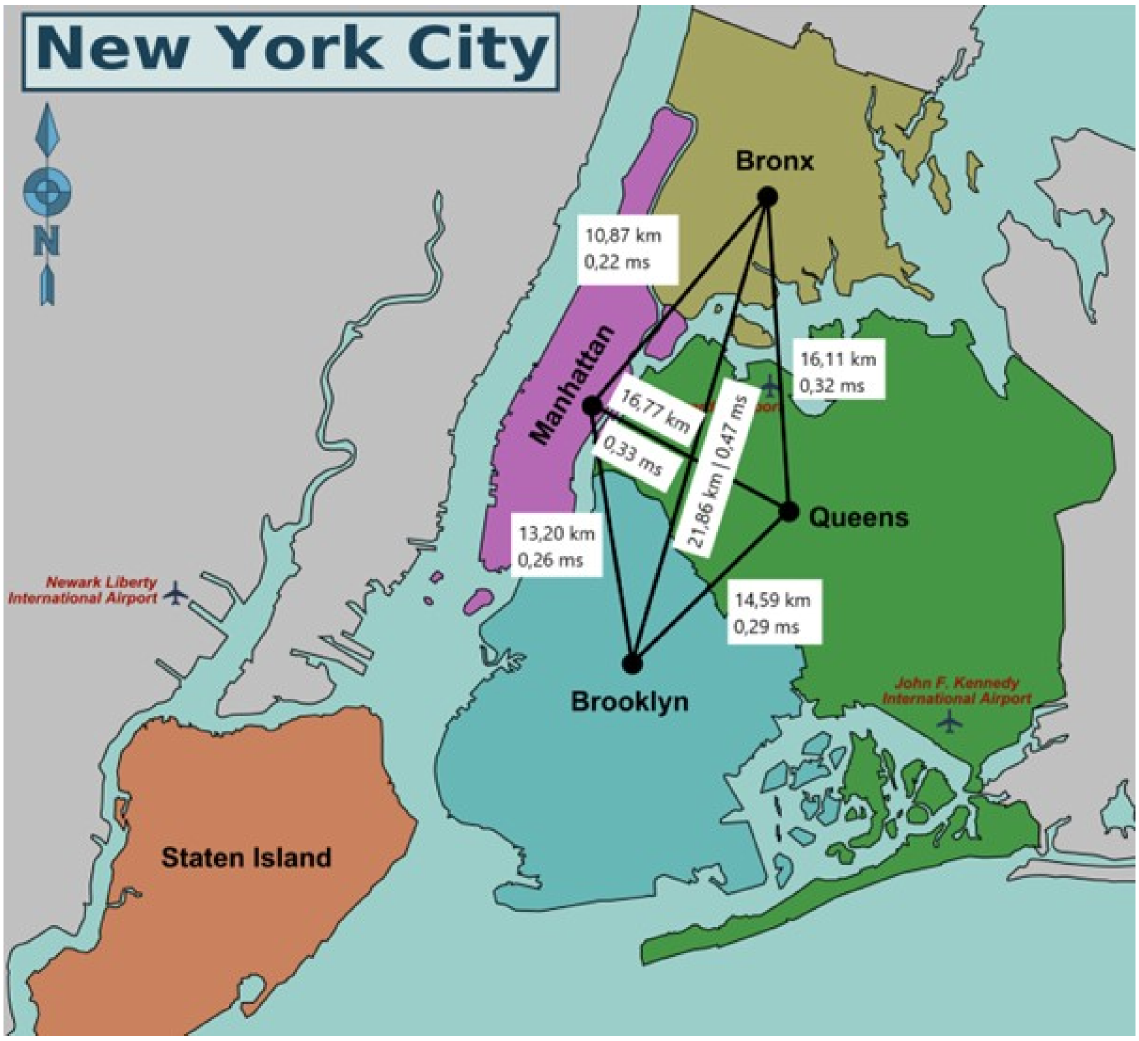


Figure 4.3: Distances and latency's between Edge nodes displayed in a Map

Additionally, we consider that the Cloud server is hosted in Azure East US datacenter, which is located in Virginia, Nevada, roughly 480 km away. These 480 km represent a latency of 9,6 ms (the average of latency between the Edge nodes is 0,31 ms, 3,23 percent of the one to the Cloud). It's important to note that the latency that we are calculating corresponds only to the travel time, not taking into consideration the processing time.

For our experimental setup, the Cloud server is physically hosted in a Docker container with the following characteristics:

- 4 Cores
- 2 GB of RAM
- 5 GB of storage

As the Cloud API is hosted inside our local machine, we manually added some latency to the network connections, as we will detail in the next paragraph.

To add latency to our experimental setup we used the Traffic Control (tc) tool to simulate latency on each of the Edge nodes whenever a vehicle is trying to connect to it. Traffic Control (TC) is a tool in Linux systems that enables the simulation of different network conditions, such as bandwidth limits and latency, among others. In a Docker environment, it is possible to use TC to simulate latency in containers.

To execute Linux commands directly to our containers, we use the following input:

- docker exec <containername> tc qdisc add dev eth0 root netem delay 100ms

It is also possible to limit the bandwidth of the connection using:

- docker exec <containername> tc qdisc add dev eth0 root tbf rate 1mbit

As previously mentioned, another variable that can affect positively and negatively the overall performance of the system is the use of batches to send information to the cloud. We discussed that these batches can be configured to follow time or length. During our tests, we evaluated the system using the following configurations:

- 5 minute batch
- 5 trips batch
- 4 Edge nodes
- 2 Edge nodes

These different configurations will be compared to a normal configuration used by most platforms, a single datacenter which performs computations in a centralized manner. The results and comparisons will be performed in the next section.

Our experimental setup allowed us to test the functionality and performance of the SMEdge framework in a simulated distributed network environment. The results of these experiments helped us identify areas for improvement and optimization to ensure that the system can be applied in multiple scenarios and configurations.

4.3 Experimental Results

The experimental results section presents the findings of our investigation on the effectiveness of the Edge computing architecture as a solution to decentralized computation. In this section, we provide a detailed analysis of the performance of our Edge computing system compared to a centralized Cloud architecture. We conducted a series of experiments to evaluate the system's performance in terms of latency, bandwidth, and computational resources utilization.

The results obtained from these experiments are presented and analyzed to determine the viability of the Edge computing system as a practical solution for decentralized computation. The findings presented in this section provide valuable insights into the effectiveness of Edge computing and contribute to the ongoing research in this area.

In addition to the analysis of performance in terms of latency, bandwidth, and computational resource utilization, we will also evaluate the effectiveness of our Edge computing system using the following metrics:

1. **Number of requests to any database:** This parameter indicates the number of times a node had to access a database to fetch data required for computation. We measured this parameter to evaluate the efficiency of the edge nodes in fetching data from a local database compared to a centralized database.

2. **Time taken to analyze a metric:** This parameter represents the time taken by the edge nodes to analyze a specific metric and generate a response. We measured this parameter to evaluate the performance of the edge nodes in processing data and generating responses compared to a centralized cloud solution.

3. **Total number of operations to the Cloud:** This parameter indicates the number of times a node had to communicate with the cloud to perform a certain operation. We measured this parameter to evaluate the efficiency of the edge nodes in performing operations locally.

4. **Total number of requests to the API, including those made to both the Edge and the Cloud:** This parameter represents the total number of requests made to the API, including requests made to the edge nodes and the cloud. We measured this parameter to evaluate the overall performance of the system and to compare the efficiency of our edge computing solution with a cloud-only solution.

These parameters will provide us with a comprehensive understanding of the performance of our Edge computing system in comparison to a centralized Cloud architecture.

We will analyze the total waiting time for requests as another key performance indicator. This will help us understand how efficiently our Edge computing system can handle requests while maintaining low latency and high throughput. By comparing the total waiting time for requests in our Edge computing system to that in a centralized Cloud architecture, we can determine the effectiveness of our solution in a real-world scenario. The results of this analysis will be presented and discussed in this section, providing insights into the practicality of Edge computing for decentralized computation.

To evaluate the parameters mentioned above, we will create a text file where the system can write the operation that were performed (such as accessing a database). To evaluate the total execution time, we will have a separate text file where we will input the execution time for each call.

For both scenarios (Centralized and De-Centralized) we will have the system running for exactly 1 hour, receiving trips in real time, which represents a volume of approximately 15,000 trips. We will test the following scenarios:

- **De-Centralized (SMEdge):**
  1. 1 Edge node for each Neighborhood
  2. 1 Edge node for each 2 Neighborhoods
  3. 5 minute batch
  4. 20 trips batch
- **Centralized (Cloud):**
  1. Trips are sent in streams and processed when necessary

For all the scenarios mentioned next, we will track 3 metrics but using different computation (max, sum and average) with a 2 minutes periodicity.

4.3.1 De-Centralized scenario (SMEdge)

We begin by presenting the test performed on the De-Centralized scenario using our Framework (SMEdge). As our dataset is composed by trips that occurred in four main neighborhoods (Manhattan, Brooklyn, Queens and Bronx), we deployed the SMEdge framework (Edge version) on **4 separate containers (Edge nodes)**. Additionally, each container has it's separate network with different latency and using a different port. To replicate the Trips information in the Cloud, we are using a **batch size of 5 Trips**.

After running the system for 1 hour, we got the following results:

- 16182 Trips were received by the Edge nodes (4.495 operations per second)

- The Cloud database was accessed 3302 times (0.917 operations per second)
- The vehicles waited an average of 40.64 ms and a total of 657637 ms (during 18.2% of the simulation time, the vehicles waited for a response)
- The max latency observed was 2457 ms

It's important to remember that the access to the Cloud database is made separately from the process that receives the trips. This means that the time waited for the Vehicles includes only the initial storing in the Edge node and the RTT.

Additionally to the metrics above, we performed some queries to our Database, to the separate tables storing the tracked metrics and representing the different nodes:

- **49 ms:** SELECT time, value, optional FROM Manhattan WHERE metric = "trip_time_in_secs"AND aggregation = "sum"
- **48 ms:** SELECT time, value, optional FROM Manhattan WHERE metric = "trip_distance"AND aggregation = "max"
- **50 ms:** SELECT time, value, optional FROM Manhattan WHERE metric = "total_amount"AND aggregation = "average"
- **1205 ms:** SELECT * FROM Trips

These queries were performed using the API to reach the database directly, without any processing in between. The tables representing a neighborhood (Edge node) have much less data when compared with the global Trips table, this will reduce the processing time and remove the need to perform computations on the raw data. With our results, we can confirm that the querying time for a tracked metric is irrelevant.

We've also tested using the aggregator endpoint available to the users from the API (/get_analyzed_metric) which returns the tracked metrics aggregated according to it's periodicity. We've sent multiple requests to this endpoint using the payloads described below (and the average response time):

- **41 ms:** /get_analyzed_metric {"metric": "trip_time_in_secs", "aggregation":"sum", "aggregation2":"sum", "neighborhoods":["Manhattan"]}
- **42 ms:** /get_analyzed_metric {"metric": "trip_time_in_secs", "aggregation":"sum", "aggregation2":"max", "neighborhoods":["Manhattan", "Brooklyn", "Queens", "Bronx"]}
- **51 ms:** /get_analyzed_metric {"metric": "trip_distance", "aggregation":"max", "aggregation2":"max", "neighborhoods":["Manhattan"]}
- **51 ms:** /get_analyzed_metric {"metric": "trip_distance", "aggregation":"max", "aggregation2":"sum", "neighborhoods":["Manhattan", "Brooklyn", "Queens", "Bronx"]}

- **42 ms:** /get_analyzed_metric {"metric": "total_amount", "aggregation":"average", "aggregation2":"average", "neighborhoods":["Manhattan"]}

- **47 ms:** /get_analyzed_metric {"metric": "total_amount", "aggregation":"average", "aggregation2":"sum", "neighborhoods":["Manhattan", "Brooklyn", "Queens", "Bronx"]}

As a comparison, the average response time for our /ping endpoint (this function just returns HTTP 200 response code to show that it is operational and doesn't perform any processing) was of 39 ms.

For our second test, we followed the same scenario as before (tracking the same trips and using a batch size of 5 Trips). However, we reduced the number of available nodes, routing all the requests from the Vehicles to Manhattan and Brooklyn (Edge nodes representing their neighborhood).

As expected, the number of Trips received was the same (16182) and the Cloud was also accessed 3302 times. However, we found that the waiting period for the Vehicles changed, as we observed the following results:

- The vehicles waited an average of 44.45 ms and a total of 719289 ms (during 19.9% of the simulation time, the vehicles waited for a response)

- The max latency observed was 4369 ms

The average response time increased by 9.38% and this increase was expected as the same load was processed by half of the resources, as we can observe in Figure 4.4:

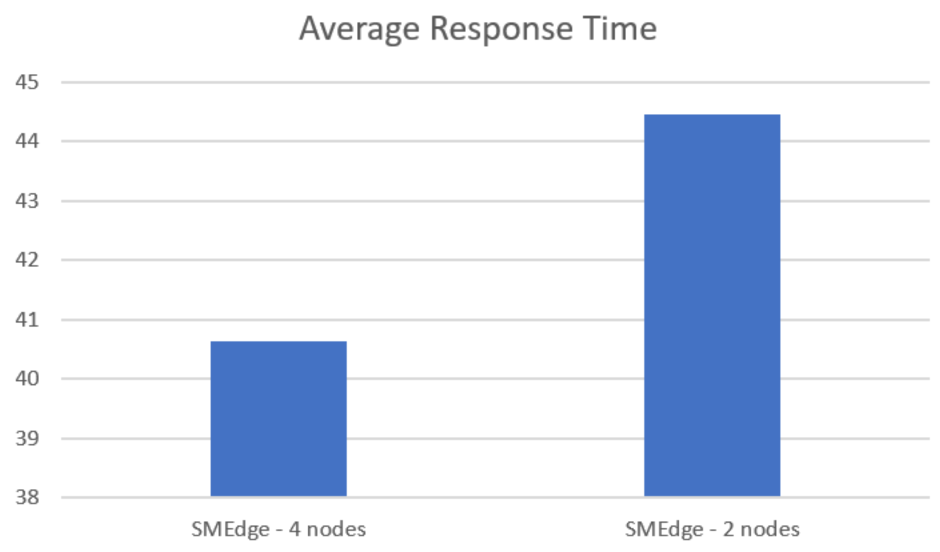


Figure 4.4: Average response time using 2 or 4 nodes

During this test, we checked once again the performance of our API endpoint /get_analyzed_metric and got the following results:

- **47 ms:** /get_analyzed_metric {"metric": "trip_time_in_secs", "aggregation":"sum", "aggregation2":"sum", "neighborhoods":["Manhattan"]}
- **52 ms:** /get_analyzed_metric {"metric": "trip_time_in_secs", "aggregation":"sum", "aggregation2":"max", "neighborhoods":["Manhattan", "Brooklyn"]}
- **63 ms:** /get_analyzed_metric {"metric": "trip_distance", "aggregation":"max", "aggregation2":"max", "neighborhoods":["Manhattan"]}
- **62 ms:** /get_analyzed_metric {"metric": "trip_distance", "aggregation":"max", "aggregation2":"sum", "neighborhoods":["Manhattan", "Brooklyn"]}
- **57 ms:** /get_analyzed_metric {"metric": "total_amount", "aggregation":"average", "aggregation2":"average", "neighborhoods":["Manhattan"]}
- **61 ms:** /get_analyzed_metric {"metric": "total_amount", "aggregation":"average", "aggregation2":"sum", "neighborhoods":["Manhattan", "Brooklyn"]}

Comparing the average response time of this endpoint between the 2 nodes and 4 nodes scenarios we faced an increase of 24.83%.

Additionally, we performed another test using 4 Edge nodes but a 5 minute batch (the batch used to replicate the data from the nodes to the Cloud). As expected, the average response time for the clients (Vehicles) was similar to the other scenario with 4 nodes as well (40.23 ms). Where we found an improvement was on the amount of accesses to the Cloud database, which was reduced to 534 (comparing to 3302), as shown in Figure 4.5:

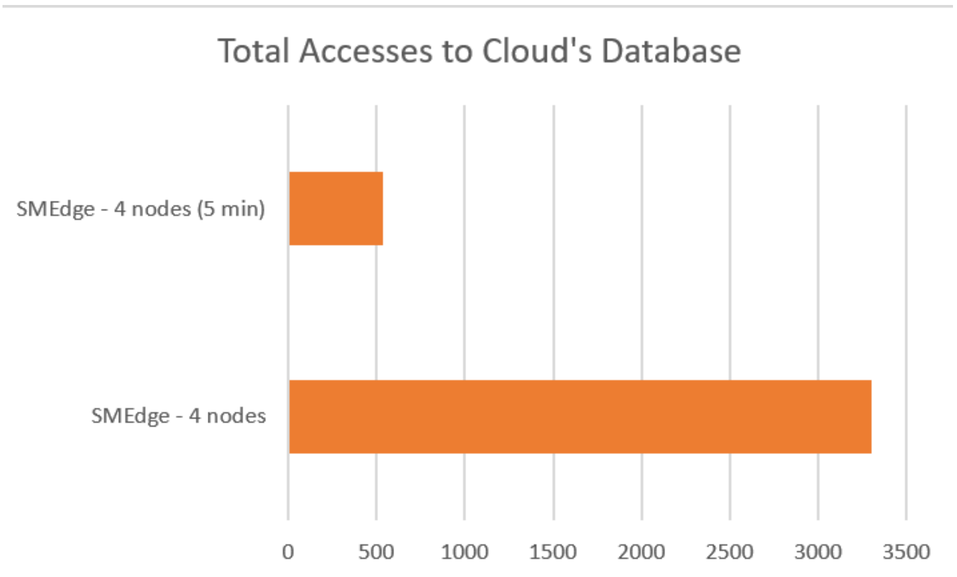


Figure 4.5: Number of total accesses to Cloud's Database

After performing the tests, we were able to visualize the tracked metrics using the Dashboard. In the x-axis we can see the timeline of the events (this time represents the

Cloud server local time and the events were logged every 2 minutes as set initially). In the y-axis we can see the result of both aggregations performed. The first aggregation is the one performed on the Edge node and the second aggregation is performed on the Cloud to the data received by each node. Below we can observe the results of a "sum"aggregation on trip_time_in_seconds (Figure 4.6), the "max"aggregation on trip_distance (Figure 4.7 and the "average"aggregation to the total_amount (Figure 4.8).

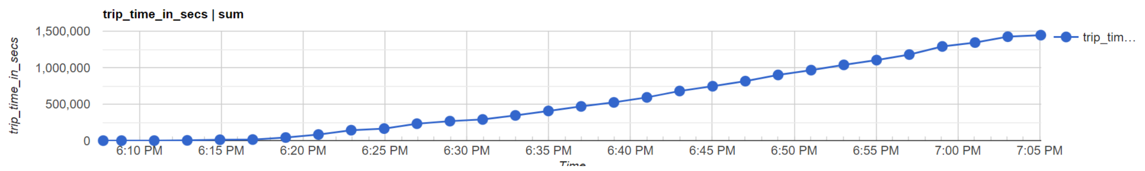


Figure 4.6: "sum"Aggregation on trip_time_in_seconds

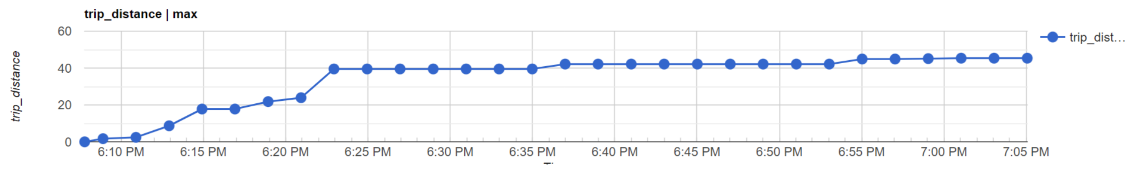


Figure 4.7: "max"Aggregation on trip_distance

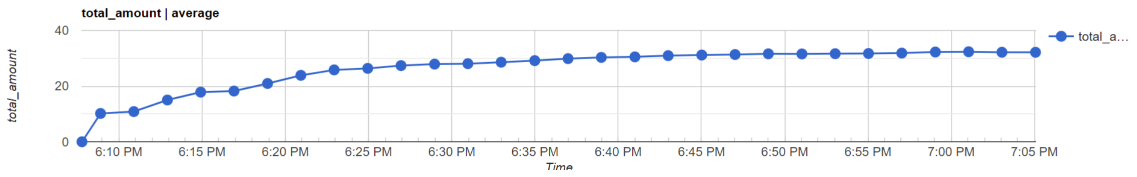


Figure 4.8: "average"Aggregation to the total_amount

We will then detail our Experimental results on a Cloud-only solution.

4.3.2 Centralized scenario (Cloud-only)

For this scenario, all the endpoints of our API were deployed to a single container representing the Cloud. This means that all clients (Vehicles) communicate with a single node relatively far away from them. This is the current scenario for most modern systems and we got the following results:

- 16182 Trips were received by the Cloud (4,495 operations per second)
- The Cloud database was accessed 16247 times (4,513 operations per second)
- The vehicles waited an average of 95,88 ms and a total of 1551530 ms (during 43,1% of the simulation time, the vehicles waited for a response)
- The max latency observed was 5719 ms

Additionally, we created an SQL query that simulates the work performed by our framework to track the metrics by querying directly the "Trips"table:

- SELECT strftime("%Y-%m-%d %H:", datetime(dropoff_datetime)) || CASE WHEN CAST(strftime("%M", datetime(dropoff_datetime)) AS INTEGER) % 2 = 0 THEN CAST(strftime("%M", datetime(dropoff_datetime)) AS INTEGER) ELSE CAST(strftime("%M", datetime(dropoff_datetime)) AS INTEGER) - 1 END || ":00"AS interval, SUM(trip_time_in_secs) AS "sum_value"FROM Trips WHERE dropoff_location = "Manhattan"GROUP BY interval

This query performs a "sum"to the data grouping it every 2 minutes. It's important to mention that this example only performs one aggregation (our framework is able to perform a second aggregation between the Edge nodes). As result, we experienced the following latency's for 10 consecutive executions:

- (1): 220ms; (2): 180ms; (3): 179ms; (4): 205ms; (5): 177ms; (6): 187ms; (7): 196ms; (8): 187ms; (9): 172ms; (10): 203ms.

This results in an average latency of 228.9 ms, an increase of 301.57% comparing to our 2 Edge nodes scenario and 401.31% to our 4 Edge node scenario, as shown in figure 4.9.

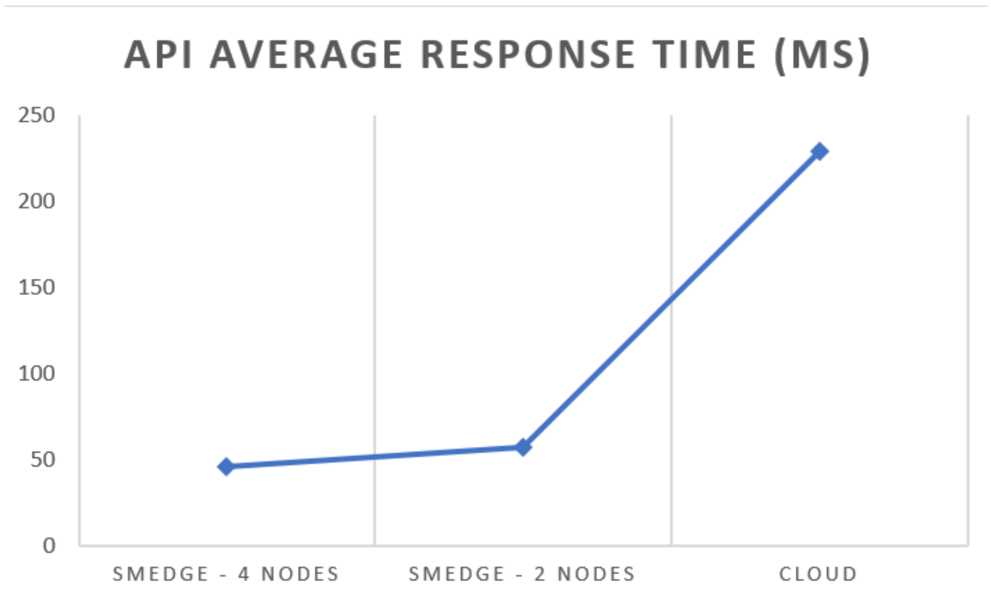


Figure 4.9: API (/get_analyzed_metric) Average Response Time in milliseconds

We have also observed that the maximum response time increased when the number of nodes decreased. This makes sense as the amount of trips per second isn't stable and the higher response times were observed when the load increased and, as python doesn't support multi-threading, the trips were processed in a queue. This problem is mitigated with the use of multiple nodes that split the traffic accordingly, as it occurs on the SMEdge framework (Figure 4.10).

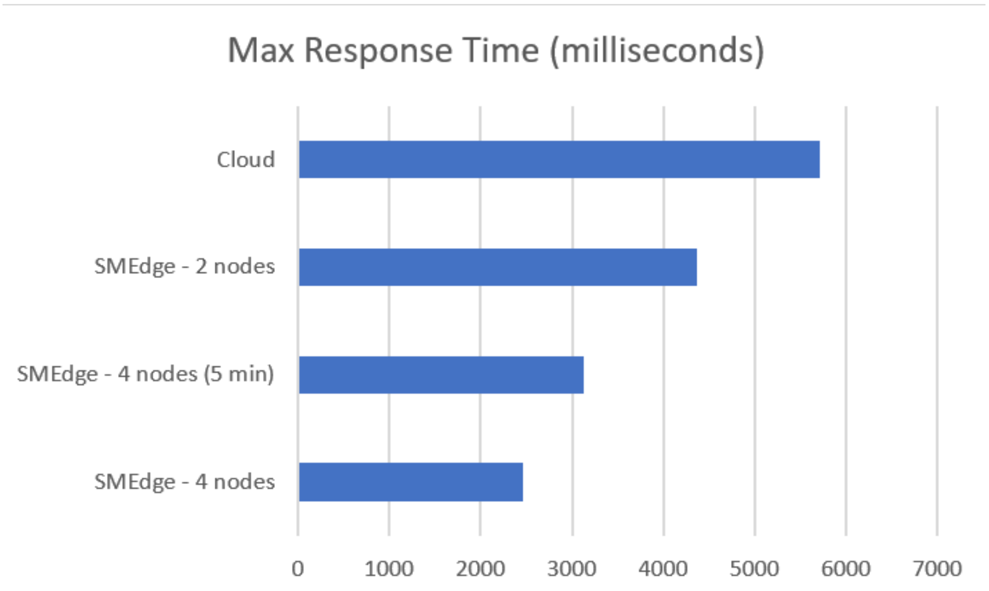


Figure 4.10: Maximum observed response time

4.3.3 Conclusion

Based on the experimental results presented in this section, we concluded that SMEdge is a more efficient and effective framework for managing and processing vehicle trips then using a Centralized approach. The use of Edge nodes in the SMEdge framework significantly reduces the number of accesses to the Cloud database, and takes the servers closer to the user, thereby reducing the response time for clients. This reduction in response time translates to improved performance, increased productivity, and a better overall user experience.

One of the key advantages of SMEdge is the significant reduction in the number of database accesses. By processing and aggregating data at the Edge nodes, the framework minimizes the need for frequent interactions with the Cloud database, resulting in improved efficiency and lower resource consumption. This not only improves the overall performance but also leads to cost savings for organizations that rely heavily on Cloud-based solutions. With fewer database queries and more efficient processing, organizations can reduce their operating costs, particularly in scenarios where SQL server systems charge based on processing time. The experiments also showed that SMEdge provides better scalability compared to a centralized approach. By distributing the workload among multiple Edge nodes, the framework is able to handle higher traffic and maintain lower response times, even under increased load. On the other hand, in the centralized scenario, as the number of nodes decreases, the average response time increases significantly. This highlights the scalability limitations of a centralized architecture and emphasizes the benefits of using Edge nodes. Furthermore, the experimental results demonstrated the effectiveness of the SMEdge framework in processing and analyzing tracked metrics. The API endpoints for retrieving analyzed metrics provided quick responses, even with multiple aggregations and large data sets. The performance of these endpoints was consistently better in the SMEdge scenarios compared to the centralized scenario, reinforcing the advantages of Edge processing.

In summary, the experimental results presented in this section demonstrate that SMEdge is a promising framework for managing vehicle trips and other devices that produce data with different parameters across time. The use of Edge nodes provides significant benefits in terms of efficiency and effectiveness, and its implementation can result in cost savings for organizations. Based on these findings, it can be recommended that SMEdge to be further explored and developed for real-world applications.

5 Conclusion and Future Work

5.1 Conclusion

In recent years, there has been a significant shift towards Edge computing, with more and more organizations recognizing the benefits of processing data closer to the source. The rise of the Internet of Things (IoT) and the need for real-time data processing are driving factors behind this trend, as organizations seek to reduce latency, improve data privacy and security, and reduce network bandwidth consumption.

The Edge computing paradigm has the potential to revolutionize the way we think about data processing and analysis. By leveraging the processing power of edge devices such as smartphones, tablets, and IoT sensors, we can reduce the burden on centralized servers and cloud infrastructure, and enable faster, more efficient data processing and analysis.

One key component of our solution is the Aggregate Computation Protocol, which allows us to aggregate data from multiple edge nodes and perform computations on the resulting data set. This protocol enables us to leverage the processing power of multiple edge devices, while minimizing the impact on network bandwidth and reducing latency.

In addition to the Aggregate Computation Protocol, we have explored several other components that are essential for building a scalable, reliable, and efficient edge computing system. These components include the Tracking Module, which monitors and records data from Edge devices, the Data Handler module, which receives and stores aggregated data from edge devices in a time series database, and the query handler module, which enables us to perform queries on the time series database and produce new ones with aggregated values.

We have also explored the benefits of using a dashboard to call an API, and the importance of data validation and cleaning filters to ensure the accuracy and consistency of the data. We have discussed the benefits of using Docker for containerization, Python as a programming language, and Flask as a web framework for building APIs.

Overall, the benefits of using edge computing to de-centralize computations are clear. By processing data closer to the source, we can reduce network bandwidth consumption, improve data privacy and security, and enable real-time data processing and analysis. This has significant implications for a wide range of industries, including transportation, healthcare, and manufacturing.

In particular, we believe that the system we have developed is well-suited for application in current ride-sharing platforms. The ability to process and analyze data in real-time, while minimizing network bandwidth consumption, is critical for ensuring the safety and efficiency of these platforms. By leveraging the processing power of edge devices, we can provide real-time insights into traffic patterns, route optimization, and driver behavior, while ensuring the privacy and security of user data.

In conclusion, the rise of Edge Computing represents a significant opportunity for organizations seeking to improve the efficiency, reliability, and scalability of their data processing and analysis systems. We believe that this system has significant potential for application in a wide range of industries, and we look forward to further exploration and development of this exciting and innovative paradigm. In the next section, we will detail how to plan to continue our work by improving and adapting the current system.

5.2 Future Work

The development of our solution does not end with the implementation of the SMEdge framework, as we understand that it is an ongoing process that requires continuous improvement to meet the needs of our users and stay up-to-date with the latest technologies. In this section, we will outline some of the steps we will take to improve our solution and make it easy to use for potential customers.

One of the first steps we will take is to gather feedback from users who have implemented our solution. We will use this feedback to identify areas for improvement and identify any pain points that users may have encountered. By understanding our users' needs, we can prioritize the features that are most important to them and make necessary changes to our solution.

Another area of focus will be on improving the user interface of the solution. We understand that not all users will have a technical background, and we want to ensure that our solution is easy to use and navigate for all users. We will work to create a clean and intuitive interface that is easy to navigate, with clear instructions and documentation.

In addition to improving the user interface, we will also focus on improving the performance and scalability of our solution. As more users adopt our solution, we want to ensure that it can handle increasing amounts of data and requests without any performance issues. We will monitor the system's performance, identify any bottlenecks, and make necessary changes to optimize performance.

We will also work to integrate our solution with other commonly used platforms and tools. For example, we could integrate with popular ride-sharing platforms such as Uber and Lyft to provide users with real-time data on their rides. By integrating with other tools, we can expand the functionality of our solution and make it more valuable to users.

Besides ride-sharing platforms, we plan on adapting our solution to other businesses and uses, including general IoT (Internet of Things) devices that produce data with specific metrics.

Finally, we will work to make our solution easy to deploy and use for potential customers. We will create detailed documentation and tutorials that guide users through the process of setting up and using our solution. We will also offer technical support and training to ensure that users are able to get the most out of our solution.

In conclusion, we understand that the development of our solution is an ongoing process, and we are committed to continuously improving it to meet the needs of our users. By gathering feedback, improving the user interface, optimizing performance, integrating with other platforms, and making it easy to deploy and use, we are confident that we can create a solution that is valuable to users and easy to use.

Bibliography

[1] *A Comparison Between Edge Containers and Other Edge Computing Solutions*. https://www.medianova.com/en-blog/a-comparison-between-edge-containers-and-other-edge-computing-solutions/. Accessed: 2022-01-27.

[2] P. A. C. A. Rosa and J. Leitão. *Revisiting Broadcast Algorithms for Wireless Edge Networks*. 2019.

[3] *Aggregate function*. https://en.wikipedia.org/wiki/Aggregate_function. Accessed: 2022-02-01 (cit. on p. 16).

[4] *Aggregate Functions*. https://docs.oracle.com/database/121/SQLRF/. Accessed: 2022-02-01 (cit. on p. 17).

[5] *Apache Flink Fundamentals: Basic Concepts*. https://www.alibabacloud.com/blog/apache-flink-fundamentals-basic-concepts_595727. Accessed: 2022-01-09 (cit. on p. 20).

[6] *Apache OpenWhisk*. https://www.techtarget.com/searchapparchitecture/definition/Apache-OpenWhisk. Accessed: 2022-02-03.

[7] *Apache OpenWhisk Provider Documentation*. https://www.serverless.com/framework/docs/providers/openwhisk. Accessed: 2022-02-02 (cit. on p. 24).

[8] *AWS IoT Greengrass Features*. https://aws.amazon.com/greengrass/features/. Accessed: 2022-01-13 (cit. on p. 23).

[9] *AWS IoT Greengrass ML Inference Solution Accelerator*. https://aws.amazon.com/iot/solutions/mli-accelerator/. Accessed: 2022-01-12 (cit. on p. 23).

[10] R. Balasubramanian and M. Aramudhan. *Security issues: public vs private vs hybrid cloud computing*. 2012.

[11] *Build cloud applications using an open source, serverless platform*. https://developer.ibm.com/components/apache-openwhisk/. Accessed: 2022-02-03 (cit. on p. 24).

[12] *Cloud computing*. https://www.techtarget.com/searchcloudcomputing/definition/cloud-computing. Accessed: 2021-12-19.

[13] *Containers at the edge: Why the hold-up?* https://www.techradar.com/news/containers-at-the-edge-why-the-hold-up. Accessed: 2022-01-29.

[14] P. A. Costa and J. Leitão. *Practical Continuous Aggregation in Wireless Edge Environments*. 2018. URL: https://asc.di.fct.unl.pt/~jleitao/pdf/MiRAge.pdf.

[15] M. C. G. D. Mealha N. Preguiça and J. Leitão. *Data Replication on the Cloud/Edge*. 2019.

[16] *Edge Containers as a Service*. https://www.section.io/modules/edge-containers-as-a-service/. Accessed: 2022-01-27.

[17] *Edge containers tutorial*. https://cloud.google.com/vision/automl/docs/containers-gcs-tutorial. Accessed: 2022-01-28.

[18] *FaaS on Edge devices*. https://community.arm.com/arm-research/b/articles/posts/faas-runtimes-on-edge-devices. Accessed: 2022-01-19 (cit. on p. 21).

[19] *Final Version of NIST Cloud Computing Definition Published*. https://www.nist.gov/. Accessed: 2021-12-07 (cit. on p. 4).

[20] *Google Cloud Computing*. https://cloud.google.com/. Accessed: 2021-12-29 (cit. on pp. 7, 8).

[21] *Hadoop MapReduce Concepts*. https://www.javacodegeeks.com/2014/04/hadoop-mapreduce-concepts.html. Accessed: 2022-01-08 (cit. on pp. 18, 19).

[22] *IaaS vs. PaaS vs. SaaS*. https://www.ibm.com/cloud/learn/iaas-paas-saas. Accessed: 2021-12-10 (cit. on p. 6).

[23] *KubeEdge*. https://kubeedge.io/en/. Accessed: 2022-01-29 (cit. on p. 14).

[24] A. Kurniawan. *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. 2018 (cit. on p. 23).

[25] *Lean OpenWhisk: Open Source FaaS for Edge Computing*. https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b. Accessed: 2022-01-13 (cit. on p. 24).

[26] *Learning Apache OpenWhisk*. https://www.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/ch01.html. Accessed: 2022-02-03 (cit. on p. 24).

[27] L. Libkin. *Expressive power of SQL*. 2003 (cit. on p. 17).

[28] *Meaning of fog in English*. https://dictionary.cambridge.org/dictionary/english/fog. Accessed: 2021-12-20 (cit. on p. 12).

[29] P. Miller. *What is edge computing?* 2018 (cit. on pp. 1, 9, 10).

[30] *NYC Open Data*. https://data.cityofnewyork.us/Business/Library/p4pf-fyc4. Accessed: 2022-01-13 (cit. on p. 56).

[31] *OpenWhisk - Github*. https://github.com/apache/openwhisk. Accessed: 2022-02-02 (cit. on p. 23).

[32] *Optimization problems*. https://tutorial.math.lamar.edu/classes/calci/optimization.aspx. Accessed: 2022-02-10.

[33] A. R. P. A. Costa and J. Leitão. *Enabling Wireless Ad Hoc Edge Systems with Yggdrasil*. 2020.

[34] *Spark Overview*. https://spark.apache.org/docs/latest/. Accessed: 2022-01-09 (cit. on pp. 20, 21).

[35] *Stateful Computations over Data Streams*. https://flink.apache.org/. Accessed: 2022-01-23.

[36] *The Different Types of Cloud Computing and How They Differ*. https://www.vxchnge.com/blog/different-types-of-cloud-computing. Accessed: 2021-12-08 (cit. on p. 5).

[37] *The Use Case for Kubernetes at the Edge*. https://thenewstack.io/the-use-case-for-kubernetes-at-the-edge/. Accessed: 2022-01-22 (cit. on p. 15).

[38] *Virtualization Gains Popularity as a Viable Solution for Enhancing Cloud Security*. https://cloudlytics.com/virtualization-gains-popularity-as-a-viable-solution-for-enhancing-cloud-security/. Accessed: 2021-12-08 (cit. on pp. 5, 8).

[39] *WebAssembly meets Kubernetes with Krustlet*. https://cloudblogs.microsoft.com/opensource/2020/04/07/announcing-krustlet-kubernetes-rust-kubelet-webassembly-wasm/. Accessed: 2022-01-21 (cit. on p. 16).

[40] *What are edge containers?* https://www.stackpath.com/edge-academy/edge-containers. Accessed: 2022-01-27.

[41] *What are public, private, and hybrid clouds?* https://azure.microsoft.com/en-us/overview/what-are-private-public-hybrid-clouds/. Accessed: 2021-12-15.

[42] *What are the differences between Cloud, Fog and Edge Computing?* https://www.mjvinnovation.com/blog/what-are-the-differences-between-cloud-fog-and-edge-computing/. Accessed: 2022-01-12 (cit. on p. 11).

[43] *What Does Kubernetes Have to Do with Edge Computing?* https://www.rtinsights.com/what-does-kubernetes-have-to-do-with-edge-computing/. Accessed: 2022-01-29 (cit. on p. 15).

[44] *What is cloud computing?* https://aws.amazon.com/what-is-cloud-computing/. Accessed: 2022-01-02 (cit. on pp. 1, 8).

[45] *What is cloud computing?* https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/. Accessed: 2021-12-19.

[46] *What is FaaS (Function-as-a-Service)?* https://www.ibm.com/cloud/learn/faas. Accessed: 2022-01-14 (cit. on pp. 21, 22).

[47] *What is fog computing?* https://internetofthingsagenda.techtarget.com/definition/fog-computing-fogging. Accessed: 2022-01-11 (cit. on p. 2).

[48] *What is fog computing?* https://www.stackpath.com/edge-academy/what-is-fog-computing. Accessed: 2022-01-11 (cit. on pp. 9, 12).

[49] *What is fog computing? Connecting the cloud to things.* https://www.networkworld.com/article/3243111/what-is-fog-computing-connecting-the-cloud-to-things.html. Accessed: 2022-01-13 (cit. on pp. 1, 11).

[50] *What is Kubernetes?* https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed: 2022-01-29 (cit. on p. 14).

[51] P. Wiener, P. Zehnder, and D. Riemer. *Towards Context-Aware and Dynamic Management of Stream Processing Pipelines for Fog Computing*. 2019. DOI: 10.1109/CFEC.2019.8733145 (cit. on pp. 2, 12).