POLITECNICO DI TORINO

Master's Degree in Computer Engineering - Computer Networks and Cloud Computing

Master's Degree Thesis

Experimental Serverless Architecture for Real-Time Sensor Data Monitoring in Fog Computing Environments

Supervisor

Prof. RICCARDO SISTO

Tutors

Dott. MARCO SCHIAPPARELLI

Dott. DANILO ABBALDO

Candidate

GIULIA BIANCHI

ACADEMIC YEAR 2022/2023

Summary

The rapid growth of IoT (Internet of Things) and Fog Computing technologies has brought about new opportunities and challenges in data monitoring and processing. This research focuses on the feasibility of using a Function-as-a-Service (FaaS) solution, specifically OpenWhisk, for monitoring sensor data in a Fog Computing environment. The study includes an analysis of existing tools, the development of an experimental use case, and evaluation of the solution's real-time properties and scalability. The objectives are to assess the practical viability of the proposed serverless architecture for edge device monitoring and to provide insights into its benefits and challenges. The thesis presents an introduction, background information, analysis of FaaS tools, exploration of OpenWhisk, use case definition, architecture details, experimental deployment, feasibility demonstration, and conclusions. This research contributes to the understanding of efficient sensor data monitoring in Fog Computing environments, offering valuable insights for future studies and applications.

Acknowledgements

To those who have been a constant presence by my side, your unwavering support and belief in me have been the pillars of my success. Your encouragement, guidance, and friendship have provided the strength and motivation needed to overcome obstacles and reach new heights. I am forever grateful for the enduring connections we have forged.

To those whose paths briefly intersected with mine, your influence, though fleeting, has left an indelible impression on my heart. Through our shared moments, conversations, and experiences, you have broadened my horizons and enriched my perspective. I am grateful for the inspiration and insights you have brought into my life.

And to those who have moved on from my journey, whether by choice or circumstance, your impact remains etched within me. The memories we created, the lessons learned, and the growth we fostered together continue to shape my thoughts and actions. I carry your presence as a cherished part of my story.

I would also like to extend my thanks to the Riccardo Sisto, my academic supervisor, and to everyone I have met in Liquid Reply, who have provided valuable insights and support throughout the duration of my thesis work. Your expertise, patience, and willingness to share your knowledge have enriched my research experience. Your guidance and feedback have played an integral role in refining my work and ensuring its quality. I am thankful for the opportunity to learn from your expertise and to benefit from your mentorship.

As I stand here, presenting my master thesis, I extend my deepest gratitude to each and every one of you. Your contributions have been invaluable, and I am forever indebted to you for the role you have played in shaping my academic path. May our paths continue to intersect, and may we inspire and uplift one another as we forge ahead on our respective journeys.

With deepest appreciation,

ありがとうございます*!*
Giulia

Table of Contents

List of Tables

List of Figures

Acronyms

IoT
Internet of Things

FaaS
Function-as-a-Service

AWS
Amazon Web Services

CLI
Command Line Interface

GCP
Google Cloud Platform

CPU
Central Processing Unit

API
Application Programming Interface

S3
Simple Storage Service

SNS
Simple Notification Service

MIT
Massachusetts Institute of Technology

HTTP

Hypertext Transfer Protocol

SSL

Secure Socket Layer

REST

Representational State Transfer

NoSQL

Not Only SQL

URL

Uniform Resource Locator

SDK

Software Development Kit

PHP

Hypertext Preprocessor

NGINX

Engine X

MQTT

Message Queuing Telemetry Transport

LRU

Least Recently Used

SLA

Service Level Agreement

CRUD

Create, Read, Update, Delete

NodeMCU

Node Microcontroller Unit

HTTPS

Hypertext Transfer Protocol Secure

TLS

Transport Layer Security

IDE

Integrated Development Environment

k3s

Kubernetes on Small Scale

RAM

Random Access Memory

Wi-Fi

Wireless Fidelity

DHT11

Digital Humidity and Temperature Sensor 11

SSID

Service Set Identifier

EEPROM

Electrically Erasable Programmable Read-Only Memory

NPM

Node Package Manager

CRD

Custom Resource Definition

R^2

Coefficient of Determination

RMSE

Root Mean Square Error

AIC

Akaike Information Criterion

BIC

Bayesian Information Criterion

SSR

Sum of Squared Residuals

SST

Sum of Squares Total

DevOps

Development and Operations

I/O

Input/Output

PromQL

Prometheus Query Language

JSON

JavaScript Object Notation

YAML

YAML Ain't Markup Language

QoS

Quality of Service

GUI

Graphic User Interface

Chapter 1

Introduction

The rapid growth of IoT (Internet of Things) and Fog Computing technologies has introduced new opportunities and challenges in the field of data monitoring and processing. [1] In this context, one significant problem arises: how to efficiently monitor and analyze sensor data generated by distributed IoT devices in a Fog Computing environment. Efficient monitoring refers to the ability to optimize resource utilization, minimize latency, and ensure timely data analysis [2]. Another challenge to consider in this context is addressing the centralized management of a distributed architecture. These problems are of utmost importance in practical applications, such as smart cities, industrial automation, and environmental monitoring, where real-time data analysis and decision-making are crucial [3].

This research focuses on the architectural experimentation and feasibility assessment of a Function-as-a-Service (FaaS) solution within the context of IoT and Fog Computing, in order to give an architectural basis to enable the future addressing of most of these challenges. Timely data analysis and centralized management will be covered as topics of this work when evaluation the feasibility of the solution, but aspects like resource optimization and latency minimization are left to further studies. The aim is to explore the viability of deploying a FaaS solution, specifically OpenWhisk, for monitoring sensor data in a Fog Computing environment. This approach allows for a thorough investigation of the architectural aspects and the practical implementation of a scalable sensor data monitoring system.

A Function-as-a-Service (FaaS) solution, such as OpenWhisk, shows potential for addressing these challenges and achieving scalability and efficiency in sensor data monitoring. FaaS offers a serverless computing model where functions are executed in response to specific events or triggers. This event-driven architecture allows for automatic scaling of resources based on demand, making it highly suitable for handling dynamic workloads generated by distributed IoT devices. As the

number of sensors and data streams increases, the FaaS solution can seamlessly scale resources, ensuring efficient utilization and responsiveness.

Furthermore, the modular nature of FaaS allows for fine-grained resource allocation. Each function can be individually scaled, optimized, and managed, enabling efficient resource utilization. Additionally, the serverless model eliminates the need for manual provisioning and management of infrastructure, reducing operational overhead and increasing overall system efficiency [4].

The research methodology involved conducting an analysis of existing tools and technologies with a primary focus on functional criteria. By considering the functional aspects, such as ease of use, event-driven capabilities, and compatibility with distributed systems, the research aimed to identify a suitable FaaS platform for the proposed application.

To validate the feasibility of the selected solution, an experimental use case was developed and implemented using OpenWhisk. The use case represents a typical scenario in which sensor data need to be collected, processed, and analyzed in a distributed Fog Computing environment. The development and architecture of the use case were designed to showcase the practical applicability of OpenWhisk in addressing the challenges of sensor data monitoring.

The evaluation of the implemented solution involved conducting experiments to assess its real-time properties and scalability. This evaluation provided insights into the capabilities of OpenWhisk in handling a realistic workload and demonstrated the feasibility of using a FaaS approach for sensor data monitoring in a Fog Computing environment.

However, it is important to note that the specific evaluation of optimized resource utilization and minimization of latencies has not been conducted in this research. The primary goal of this thesis was to assess the feasibility of the proposed solution, in order to give a starting point to demonstrate its potential for addressing challenges of sensor data monitoring that are not explored in this work. While the architecture and implementation of the solution showcase the potential for optimized resource utilization and efficient monitoring, a comprehensive evaluation of these aspects, including performance comparisons with other architectures and benchmarking, is a topic for future studies.

The architectural design employed in this research, utilizing a Function-as-a-Service (FaaS) solution like OpenWhisk within a Fog Computing environment, exhibits the potential for optimized resource utilization and efficient monitoring. The

event-driven scalability of the FaaS model allows for automatic scaling of resources based on demand, ensuring efficient utilization and responsiveness as the number of sensors and data streams increases. Moreover, the fine-grained resource allocation capabilities enable efficient utilization by scaling and managing individual functions. The serverless infrastructure management aspect eliminates the need for manual provisioning and management of infrastructure, reducing operational overhead and increasing overall system efficiency. Additionally, the distributed nature of Fog Computing, combined with the scalability of FaaS, supports efficient monitoring and analysis of sensor data, bringing computation closer to the data source and reducing latency.

To fully assess the optimality of the architecture in terms of resource utilization and latency minimization, further studies involving performance evaluations and comparisons with alternative architectures are necessary. These future evaluations would provide a deeper understanding of the FaaS solution's performance characteristics and its efficiency compared to other approaches.

1.1 Objectives

The research aims to demonstrate the **practical feasibility** of the proposed serverless architecture for edge device monitoring in a fog computing environment, with one of the key objectives being to **define the architecture** itself. To showcase the practical feasibility of the architecture, the initial step involves constructing a proof-of-concept implementation. Subsequently, the focus shifts towards demonstrating its applicability in real-world scenarios by evaluating key factors such as **scalability**, **real-time capabilities**, and **centralized component management**. Additionally, the objective is to evaluate the seamless **integration** of diverse components within the architecture, ensuring their effective collaboration and interoperability.

By achieving these objectives, the research aims to provide valuable insights into the potential benefits and challenges of implementing such an architecture in real-world scenarios. Furthermore, the research strives to ensure a cohesive and well-integrated system for edge device monitoring, contributing to the advancement of fog computing and its practical application in monitoring environments.

1.2 Methodology

To achieve the stated objectives, the following research approach and tools were employed:

1. **Literature Review**: A comprehensive review of existing literature and research papers was conducted to gain a deep understanding of serverless computing, fog computing and their applications in IoT environments. This literature review served as the foundation for identifying key concepts, challenges, and potential solutions related to the implementation of a serverless architecture in a fog computing environment.

2. **Software Selection**: A careful evaluation of available serverless platforms was conducted based on functional requirements specific in a fog computing environment. Criteria such as ease of deployment, compatibility with different programming languages, and support for event-driven architectures were considered. OpenWhisk was selected as the preferred serverless platform for its suitability and alignment with the research objectives.

3. **System Design**: Based on the knowledge gained from the literature review and the chosen serverless platform, a system design was developed to outline the architecture and components of the proposed solution. The design focused on leveraging computing capabilities, confining computation to the edge clusters, and incorporating a centralized management component.

4. **Implementation**: A proof-of-concept system was implemented to demonstrate the feasibility of the proposed architecture. OpenWhisk was deployed on the edge clusters, enabling real-time monitoring capabilities with integrated edge devices. Additionally, OpenWhisk was also deployed in the centralized management component, providing seamless coordination between the edge clusters and centralized operations. The implementation utilized ESP-8266 sensor to collect temperature, gas percentage, and humidity data from each data center, ensuring comprehensive monitoring of their "health" status.

5. **Experimentation and Evaluation**: Experiments were conducted to evaluate the feasibility of the implemented system in the context of serverless architecture and edge computing capabilities. The primary objective was to demonstrate the system's ability to scale and maintain real-time responsiveness as the number of sensors increased, in order to have a proof of its fesibility. To assess these aspects key metrics including average response time, average network troughput, average CPU utilization, and average RAM utilization were measured and analyzed.

6. **Data Analysis and Findings**: The collected data from the experiments were analyzed to draw conclusions and evaluate the feasibility and effectiveness of the proposed solution. The findings were compared against the objectives and discussed.

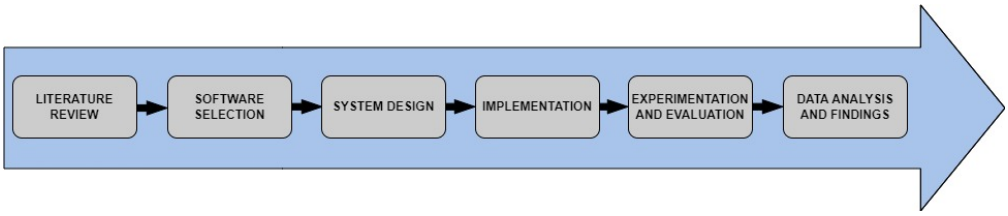

Figure 1.1: Methodology Adopted to Develop the Thesis

1.3 Thesis Organization

The thesis is organized in the following sections:

1. **Introduction**: This chapter provides an introduction to the research topic, presents the objectives of the study, and outlines the methodology used to achieve these objectives.

2. **Background**: In this chapter, the background information related to the research topic is presented. It covers concepts such as Internet of Things (IoT), Fog Computing, and Function as a Service (FaaS).

3. **Analysis of Existing FaaS Tools**: This chapter analyzes various FaaS tools available in the market, including OpenFaaS, OpenWhisk, Knative, AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions. A comparison matrix is provided, and the selection of OpenWhisk as the chosen FaaS tool is explained.

4. **Deep Dive into OpenWhisk**: This chapter provides an in-depth exploration of the OpenWhisk platform, its components, and functionalities. It covers topics such as the OpenWhisk CLI, actions, triggers and rules, packages, and the cache and load balancer mechanisms.

5. **Use Case Definition**: This chapter defines the use case scenario that will be used to evaluate the feasibility of the proposed solution. It discusses the use case scenario and its adaptability.

6. **Architecture and Implementation Details**: This chapter presents the architecture of the proposed solution based on OpenWhisk. It describes the components of the architecture and provides implementation details, including the main action mechanism and component interaction.

7. **Experimental Deployment**: This chapter focuses on the practical implementation of the system, shedding light on the technologies, tools, and processes involved in deploying the edge and centralized instances.

8. **Demonstration of Feasibility**: This chapter focuses on the demonstration of the feasibility of the system, in particular on the details of how scalability and real-time properties are evaluated in the context of the centralized and edge instances.

9. **Conclusions**: This final chapter summarizes the key findings of the research and presents the conclusions drawn from the evaluation. It also discusses the limitations of the study and suggests potential areas for future research.

Chapter 2

Background

In order to comprehend the context and significance of the proposed solution, it is essential to establish a foundation by examining three key components: the **Internet of Things** (IoT), **fog computing**, and **Function-as-a-Service** (FaaS). These technologies play crucial roles in enabling efficient data processing, analysis, and communication, thereby revolutionizing various domains and industries.

2.1 Internet of Things

The Internet of Things (IoT) is a transformative technology that encompasses a vast network of interconnected devices, ranging from everyday objects to complex machinery, embedded with sensors, software, and network connectivity [5] [6]. These devices have the ability to collect, exchange, and analyze data, enabling them to communicate with each other and with centralized systems.

The significance of IoT in today's world lies in its ability to revolutionize numerous industries and sectors. By connecting physical objects and integrating them into digital systems, IoT enables enhanced automation, monitoring, and control of various processes [7] [8]. It has the potential to improve efficiency, productivity, and decision-making across diverse domains, including manufacturing, transportation, healthcare, agriculture, and more [9].

In this solution, IoT devices play a vital role as sensors deployed within the data centers. These devices continuously collect data related to environmental conditions, such as temperature, humidity, and gas levels. These real-time data provide valuable insights into the health and performance of the data centers, facilitating proactive maintenance.

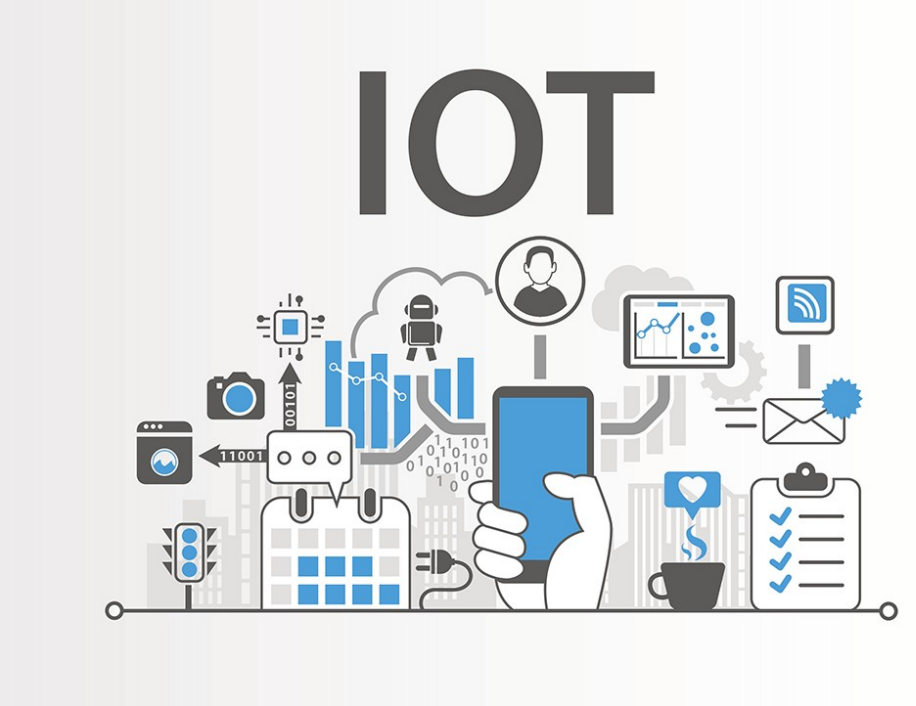

Figure 2.1: IoT

2.2 Fog Computing

Fog computing is a distributed computing paradigm that extends the cloud computing paradigm to the edge of the network [10]. It provides a way to process and store data closer to the source of the data, which can improve performance and reduce latency. Fog computing is often used in applications where real-time response is critical, such as in the Internet of Things (IoT).

The fog computing architecture consists of three layers:

1. The **edge layer** is the closest to the data source. It consists of devices such as sensors, actuators, and gateways.

2. The **fog layer** is in the middle of the network. It consists of servers and routers that can perform some processing and storage.

3. The **cloud layer** is the farthest from the data source. It consists of large data centers that can provide a lot of processing and storage power.

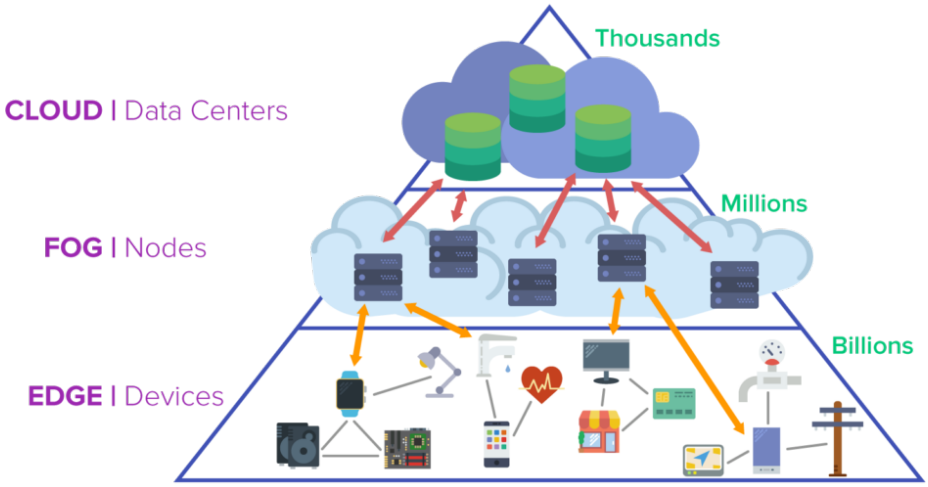

Figure 2.2: Fog Computing Layers

Fog computing has a number of advantages over cloud computing, including:

1. **Improved performance**: Fog computing can improve performance by reducing latency. This is because data does not have to travel as far to be processed.

2. **Reduced bandwidth usage**: Fog computing can reduce bandwidth usage by performing some processing and storage at the edge of the network. This can free up bandwidth for other applications.

3. **Increased security**: Fog computing can increase security by processing data closer to the source. This can make it more difficult for attackers to access the data .

Fog computing is a promising new technology that has the potential to revolutionize the way we use the Internet. It has a number of advantages over cloud computing, and it is well-suited for applications where real-time response is critical.

In this solution, fog computing plays a pivotal role in optimizing the monitoring of multiple data centers in a fog computing environment. By deploying fog nodes within each data center, it is possible to process and analyze sensor data locally, reducing reliance on centralized cloud resources and minimizing network congestion.

2.3 Function-as-a-Service

FaaS (Function-as-a-Service) is a cloud computing model that allows developers to execute small units of code (functions) in a serverless environment. In this model, developers focus on writing and deploying individual functions without the need to manage the underlying infrastructure or worry about server provisioning, scaling, or maintenance [11].

FaaS platforms, such as AWS Lambda, Google Cloud Functions, or Microsoft Azure Functions, provide the necessary runtime environment to execute the functions in response to specific events or triggers. The functions are typically short-lived and stateless, designed to perform specific tasks or computations [11].

The significance of FaaS lies in its ability to provide a highly scalable and cost-effective solution for executing code. By abstracting away the complexities of infrastructure management, FaaS enables developers to focus on writing code and delivering functionality without worrying about the underlying infrastructure. FaaS platforms handle the scaling and management of resources automatically, allowing functions to scale up or down based on demand [11].

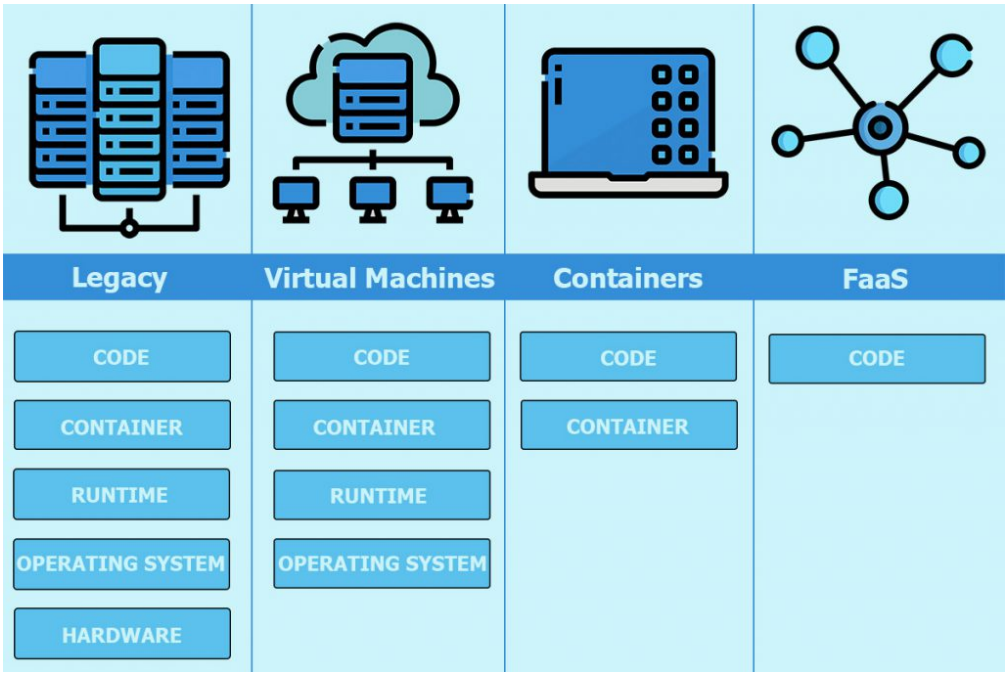

Figure 2.3: Cloud Computing Models

In this solution, FaaS plays a crucial role in processing and analyzing the sensor

data collected from the data centers. The event-driven nature of FaaS allows us to trigger these functions in real-time based on the incoming data streams from the sensors.

By utilizing FaaS, it is possbile to achieve efficient and scalable processing of sensor data without the need to provision and manage dedicated servers. FaaS gives the possibility to dynamically scale the computing resources based on the workload, ensuring optimal performance and cost-effectiveness [11].

Chapter 3

Analysis of Existing FaaS Tools

This chapter analyzes various Function as a Service (FaaS) tools to determine the most suitable framework for the proposed solution. The evaluation focuses on functional criteria such as license, installation type, source code availability, programming language support, interface types, community support, and documentation.

The chapter provides an overview of FaaS tools, including OpenFaaS, OpenWhisk, Knative, AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions.

A comparison matrix summarizes the key features of open-source tools and cloud provider tools, facilitating easy comparison.

After a comprehensive analysis, OpenWhisk is chosen as the preferred framework due to its superiority in aspects such as license, installation type, source code availability, programming language support, interface types, community support, and documentation.

The subsequent sections explore each FaaS tool in detail, providing deeper insights into their capabilities and assisting in the selection process.

3.1 Criteria for Tool Evaluation

In the selection of the most suitable tool or framework in order to reach the primary objective of demonstrating the feasibility of this architectural experimentation, several criteria have been defined. The focus of the evaluation lies primarily on

functional requirements, as the goal is to assess the feasibility and practicality of the tools rather than their optimality. To assess the feasibility of the architecture it should be possible to build it, so the chosen FaaS tool should not limit the creation of the defined architecture. By choosing functional requirements that directly assist in building the architecture, it is possible to ensure that the selected tools effectively support its construction. Additionally, the architecture should be designed to support various integrations with different components, systems, or services. Furthermore, the architecture should provide tools for monitoring, which play a vital role in evaluating scalability, real-time performance, which are some of the objectives of this research, in order to evaluate the feasibility of the system. Lastly, the architecture should serve as a basis for further studies, enabling researchers to explore its optimality and identify potential areas of improvement, so the proposed tool should allow easily extension or integrations of the architecture.

The following functional criteria will be used to evaluate the different tools [12]:

1. **License**: The choice of an open-source license, such as Apache 2.0 or MIT, is beneficial for demonstrating the feasibility of an architecture. Open-source licenses provide more flexibility for customization and collaboration, allowing developers to modify and enhance the FaaS tool to align with the specific requirements of the architecture. By having access to the source code, developers can tailor the tool to better integrate with other components, improve its functionality, and advance the architecture's capabilities. This open nature of the license promotes experimentation, innovation, and optimization, all of which are crucial in assessing the feasibility and practicality of the architecture [13].

2. **Installation Type**: The compatibility of the FaaS tool with various installation environments is crucial in building a feasible architecture. By ensuring that the tool can be installed on preferred target hosts, such as Docker, Kubernetes, Linux, macOS, Windows, or specific cloud platforms, developers can seamlessly deploy and manage the architecture in their desired environment. A smooth installation process and compatibility with the chosen platform reduce deployment complexities, streamline management efforts, and contribute to the overall feasibility of the architecture.

3. **Source Code Availability**: Open-source FaaS tools with accessible source code are particularly valuable when evaluating the effective integration between different architecture components. By allowing developers to access and modify the source code, these tools facilitate customization and integration with other components within the architecture. This flexibility enables developers to experiment with different configurations, optimize interactions between

components, and ensure a well-integrated system. Access to the source code also fosters collaboration and knowledge sharing among developers, further enhancing the feasibility and effectiveness of the architecture [14].

4. **Main Programming Language**: The compatibility of the FaaS tool with multiple programming languages supports flexibility in implementing and evaluating the architecture. By supporting a wide range of languages, such as Node.js, Python, Java, and more, the tool accommodates the preferences and expertise of developers. This flexibility allows for the selection of the most appropriate language for each component, taking into account performance, developer productivity, and integration requirements. Compatibility with multiple programming languages enables a diverse ecosystem of developers to contribute to the architecture's construction and evaluation, enhancing its feasibility [15].

5. **Interface Types**: A FaaS tool that offers a variety of interfaces, including command-line interface (CLI), application programming interface (API), and graphical user interface (GUI), simplifies the usage, development, deployment, and management of architecture components. These interfaces provide different entry points for developers to interact with the tool and the architecture. CLI interfaces enable automation and scripting, API interfaces facilitate programmatic control and integration with other tools, and GUI interfaces offer intuitive visual representations and management capabilities. Having multiple interface types increases the usability and accessibility of the tool, enabling developers to effectively work with the architecture and evaluate its feasibility [16, 17].

6. **Community Support**: Robust community support plays a vital role in demonstrating the feasibility of an architecture. Active communities on platforms like GitHub and Stack Overflow provide valuable resources, knowledge sharing, and troubleshooting assistance. Developers can leverage the collective expertise of the community to address challenges, seek guidance on best practices, and overcome implementation obstacles. This support fosters collaboration, accelerates development, and ensures successful implementation of the architecture. Additionally, community engagement promotes the discovery of potential improvements and optimizations, laying the groundwork for future studies on the architecture's optimality [18].

7. **Documentation**: Comprehensive and well-maintained documentation is crucial in building a feasible architecture and integrating different components effectively. Documentation provides a clear understanding of the FaaS tool's features, capabilities, and best practices, guiding developers in utilizing the tool to its full potential. It helps with the development, deployment, and

management of the architecture, ensuring smooth integration between different components. Detailed documentation enhances the feasibility of the architecture by reducing implementation errors, facilitating knowledge transfer, and enabling developers to make informed decisions [19].

8. **Event Sources**: The compatibility of the FaaS tool with various event sources is essential for effective data ingestion and processing in edge device monitoring scenarios. Support for event sources such as HTTP, messaging protocols (e.g., MQTT, Kafka), and cloud storage services (e.g., AWS S3, Azure Blob Storage) enables the architecture to receive and process data from diverse sources. This compatibility is crucial in evaluating the architecture's feasibility in handling real-time data streams, scaling with increasing event volumes, and integrating with different data providers. By supporting a wide range of event sources, the FaaS tool contributes to the architecture's versatility and effectiveness in edge device monitoring [12].

9. **Function Orchestration**: Function orchestration capabilities provided by the FaaS tool are essential for managing workflows involving multiple functions within the architecture. This capability is particularly useful in coordinating different components and subsystems, assessing their effective integration, and evaluating the architecture's feasibility. Function orchestration allows developers to define the sequencing, dependencies, and interactions between functions, ensuring the desired workflow behavior. It enables the evaluation of how well different components work together and whether the architecture can effectively fulfill its intended purpose. Function orchestration is crucial in assessing the feasibility of the architecture's workflow management and integration capabilities [20].

10. **Observability**: The observability features provided by the FaaS tool, such as logging, monitoring, and metrics collection, are crucial for assessing the performance, scalability, and real-time characteristics of the architecture. These features enable developers to gain insights into the behavior of the architecture components, resource utilization, and system-level performance. By monitoring and collecting relevant metrics, developers can evaluate the architecture's scalability under varying workloads, ensure real-time requirements are met, and detect and diagnose performance issues. Observability tools contribute to the feasibility assessment by providing valuable information for optimization, identifying bottlenecks, and validating the architecture's performance [21].

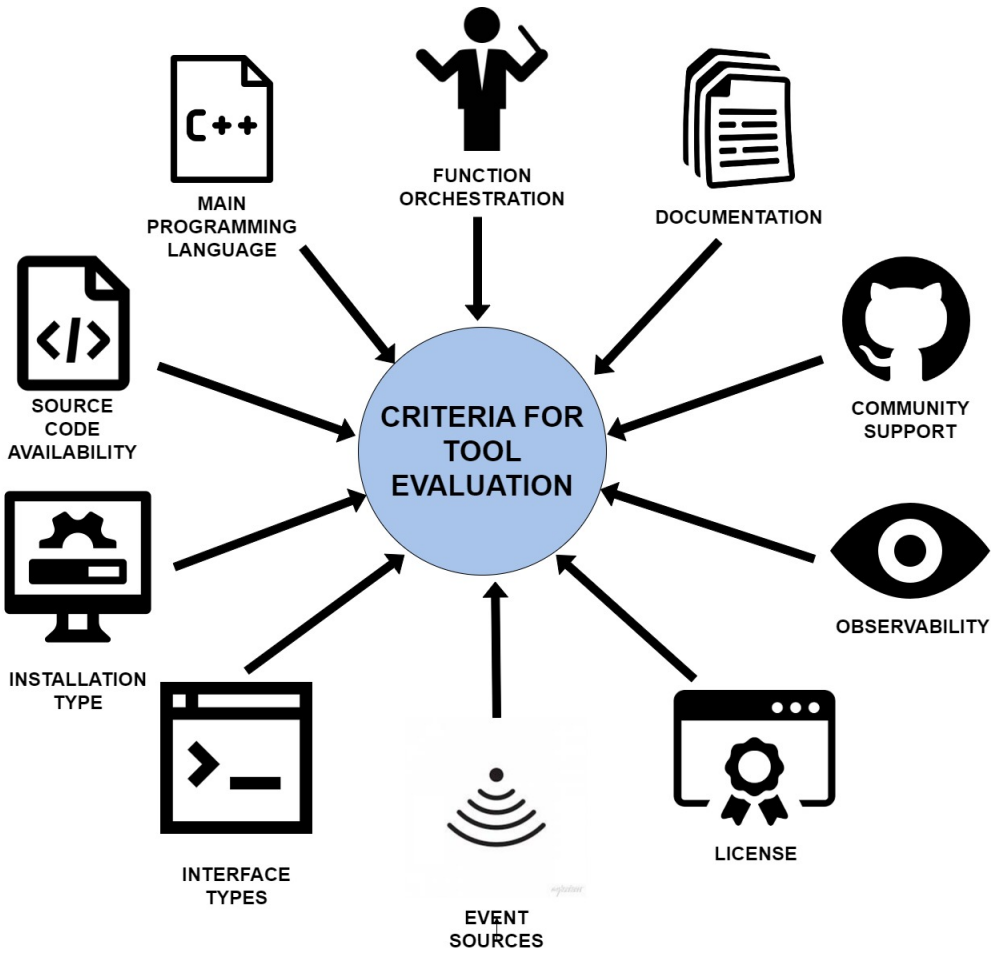

Figure 3.1: Criteria for Tool Evaluation

Considering the outlined criteria is instrumental in selecting a FaaS tool that supports the construction, integration, monitoring, and evaluation of a feasible architecture. Each criterion contributes to specific aspects of the architecture's feasibility, from customization and collaboration to compatibility, monitoring, and optimization. By carefully evaluating and selecting a tool that fulfills these criteria, researchers and developers can demonstrate the feasibility of the architecture while laying the foundation for further studies on its optimality.

3.2 Tools

In this section, each analyzed FaaS tool will be shortly presented.

3.2.1 OpenFaaS

OpenFaaS is an open-source serverless computing platform that allows developers to build and deploy functions quickly. It provides a framework for deploying containerized functions and managing them as microservices. OpenFaaS supports multiple programming languages and integrates with popular container orchestration platforms like Kubernetes. It offers a scalable and event-driven architecture, making it suitable for building serverless applications. OpenFaaS is designed for flexibility and can be deployed on various cloud providers or on-premises environments [22].

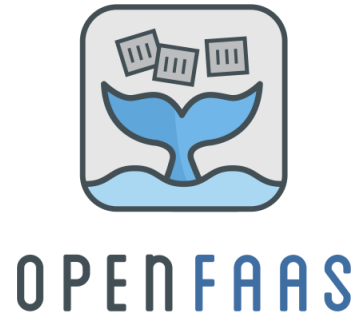

Figure 3.2: OpenFaas Logo

3.2.2 OpenWhisk

OpenWhisk is an open-source, event-driven serverless computing platform. It allows developers to create and run functions in response to events, such as HTTP requests or messages from message queues. OpenWhisk supports multiple programming languages and provides a flexible and scalable infrastructure for running serverless workloads. It offers fine-grained scaling, meaning that resources are allocated to functions only when needed, reducing costs and improving efficiency. OpenWhisk is part of the Apache Software Foundation and can be deployed on various cloud platforms or on-premises environments [23].

Figure 3.3: OpenWhisk Logo

3.2.3 Knative

Knative is an open-source platform built on top of Kubernetes that provides a set of building blocks for deploying and managing serverless workloads. It abstracts away the complexity of deploying and scaling applications by offering automatic scaling, eventing, and request-driven compute models. Knative allows developers to focus on writing code without worrying about infrastructure management. It supports various programming languages and integrates with container runtimes like Docker and container orchestration platforms like Kubernetes. Knative is designed to be portable and can be deployed on different cloud providers [24].

Figure 3.4: Knative Logo

3.2.4 AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). It enables developers to run code without provisioning or managing servers. With Lambda, developers can create functions in multiple programming languages and configure them to be triggered by various events, such as API calls, file uploads, or database updates. AWS Lambda automatically scales the functions based on incoming requests, ensuring high availability and cost efficiency. It integrates seamlessly with other AWS services, allowing developers to build powerful serverless architectures [25].

Figure 3.5: AWS Lambda Logo

3.2.5 Google Cloud Functions

Google Cloud Functions is a serverless computing service offered by Google Cloud Platform (GCP). It allows developers to write functions in languages like Node.js, Python, or Go and deploy them to Google's infrastructure. Google Cloud Functions can be triggered by various events, including HTTP requests, Pub/Sub messages, or changes in cloud storage. It automatically scales functions to handle incoming requests and provides integration with other GCP services, enabling developers to build serverless applications using a wide range of GCP offerings [26].

Figure 3.6: Google Cloud Functions Logo

3.2.6 Microsoft Azure Functions

Microsoft Azure Functions is a serverless computing service provided by Microsoft Azure. It allows developers to build and deploy functions in different programming languages such as C, JavaScript, or Python. Azure Functions can be triggered by various events, including HTTP requests, timers, or changes in data storage. The service automatically scales functions to handle incoming requests and integrates seamlessly with other Azure services, enabling developers to create robust serverless applications using the Azure ecosystem [27].

Figure 3.7: Microsoft Azure Functions Logo

3.3 Comparison Matrix

Table 3.1: Comparison Matrix – Open Source Tools [12]

Feature	OpenWhisk	OpenFaaS	Knative
License	Apache 2.0	MIT	Apache 2.0
Installation Type	Installable on Docker, Kubernetes, Linux, macOS, Windows, Mesos	Installable on Docker, Kubernetes, Linux, macOS, Windows	Installable on Kubernetes
Source Code Availability	Open Source, Repository: Apache Incubator (GitHub)	Open Source, Repository: GitHub	Open Source, Repository: GitHub
Main Programming Language	Ballerina, Binary, Docker Image, Go, Java, .NET, Node.js, PHP, Python, Ruby, Rust, Shell, Swift	C#, Docker Image, Go, Java, Node.js, Python, PHP, Ruby	Docker
Interface Type	CLI, API, GUI	CLI, API, GUI	CLI, API
Community Support	16,800 GitHub Stars, 932 GitHub Forks, 274 GitHub Issues, 2,790 Stack Overflow Questions	4,900 GitHub Stars, 487 GitHub Forks, 215 GitHub Issues, 1,156 Stack Overflow Questions	3,600 GitHub Stars, 764 GitHub Forks, 173 GitHub Issues, 632 Stack Overflow Questions
Documentation	**Platform Documentation**: usage, development, deployment, architecture. **Functions Documentation**: development, deployment	**Platform Documentation**: usage, development, deployment, architecture. **Functions Documentation**: deployment	**Platform Documentation**: usage, development, deployment. Functions Documentation: deployment

Continued on next page

Table 3.1 – Continued from previous page

Feature	**OpenWhisk**	**OpenFaaS**	**Knative**
Event Sources	Scheduler, Apache Kafka, IBM Message Hub, GitHub, Slack, Weather Company Data, IBM Push Notifications, Websocket API, hooks, polling, connection	Scheduler, AWS SQS, AWS SNS, MQTT, Apache Kafka, Azure Event Grid, IFTTT, VMware vCenter, NATS, connector plugins	Scheduler, AWS SQS, AWS SNS, Google Cloud PubSub, RabbitMQ, Apache Kafka, AWS Kinesis, Apache Camel, Kubernetes API server events, GitHub, GitLab, BitBucket, Google Cloud Scheduler, AWS CodeCommit, AWS Cognito, FTP/SFTP, Heartbeat events, Websocket, plugin-based
Event Sources (Endpoint)	**Synchronous Endpoint**: HTTP. **Asynchronous Endpoint**: HTTP. **Customization**: yes. **TLS support**: yes	**Synchronous Endpoint**: HTTP. **Asynchronous Endpoint**: HTTP. **Customization**: yes. **TLS support**: yes	**Synchronous Endpoint**: HTTP. **Asynchronous Endpoint**: -. **Customization**: yes. **TLS support**: yes

Continued on next page

Table 3.1 – Continued from previous page

Feature	OpenWhisk	OpenFaaS	Knative
Function Orchestration	**Function Orchestrator**: Apache OpenWhisk Composer. **Function Workflows Definition**: Orchestrating function. **Orchestrating Function Languages**: JavaScript, Python. **Control Flow Constructs Documentation**: Yes. **Function Workflow Execution Time Quota**: Yes. **Function Workflow Task I/O Size Quota**: No	**Function Orchestrator**: -. **Function Workflows Definition**: -. **Orchestrating Function Languages**: -. **Control Flow Constructs Documentation**: -. **Function Workflow Execution Time Quota**: -. **Function Workflow Task I/O Size Quota**: -	**Function Orchestrator**: Knative Eventing. **Function Workflows Definition**: custom DSL. **Orchestrating Function Languages**: -. **Control Flow Constructs Documentation**: yes. **Function Workflow Execution Time Quota**: no. **Function Workflow Task I/O Size Quota**: no
Observability	Kamon, Prometheus, Datadog	OpenFaaS Gateway + Prometheus	-

Table 3.2: Comparison Matrix – Cloud Provider Tools [12]

Feature	AWS Lambda	Google Cloud Functions	Azure Functions
License	AWS Service Terms (proprietary)	Google Cloud Platform Terms (proprietary)	SLA for Azure Functions (proprietary)

Continued on next page

Table 3.2 – Continued from previous page

Feature	**AWS Lambda**	**Google Cloud Functions**	**Azure Functions**
Installation Type	as-a-service	as-a-service	installable and as-a-service. Installable on Kubernetes, Linux, MacOS, and Windows
Source Code Availability	Closed Source	Closed Source	Open Source, Repository: GitHub
Main Programming Language	Go, Java, .NET, Node.js, Python, Ruby, Shell	Go, Java, Node.js, Python	Docker, C#, JavaScript, F#, Java, Powershell, Python, TypeScript
Interface Type	CLI, API, GUI	CLI, API, GUI	CLI, API, GUI
Community Support	16,800 Stack Overflow Questions	12,386 Stack Overflow Questions	1,300 GitHub Stars, 279 GitHub Forks, 948 GitHub Issues, 7,200 Stack Overflow Questions
Documentation	**Platform Documentation**: usage. **Functions Documentation**: development, deployment	**Platform Documentation**: usage. **Functions Documentation**: development, deployment	**Platform Documentation**: usage, deployment. **Functions Documentation**: development, deployment

Continued on next page

Table 3.2 – Continued from previous page

Feature	**AWS Lambda**	**Google Cloud Functions**	**Azure Functions**
Event Sources	Scheduler, AWS SQS, AWS SNS, Amazon Kinesis, Amazon Alexa, AWS CloudTrail, AWS CloudWatch, AWS CodeCommit, AWS CodePipeline, Amazon Cognito, AWS Config, AWS EC2, Elastic Load Balancing, AWS IoT, Amazon Kinesis Data Firehose, Amazon Lex, Amazon MQ, Amazon RDS, Amazon SES, AWS X-Ray, event source mappings	Scheduler, Google Cloud Pub/Sub, Google Analytics for Firebase, Firebase Authentication, Firebase Remote Config, Cloud Logging, Gmail, webhooks, messaging-based	Scheduler, Azure Queue Storage, Azure Service Bus, RabbitMQ, Azure Event Hubs, Apache Kafka, Azure Event Grid, Azure IoT Hub, Azure Mobile Apps, Azure Notification Hubs, Azure SignalR Service, SendGrid, Twilio, custom I/O bindings
Event Sources (Endpoint)	**Synchronous Endpoint**: HTTP. **Asynchronous Endpoint**: HTTP. **Customization**: yes. **TLS support**: yes	**Synchronous Endpoint**: HTTP, RPC. **Asynchronous Endpoint**: -. **Customization**: yes. **TLS support**: yes	**Synchronous Endpoint**: HTTP. **Asynchronous Endpoint**: -. **Customization**: yes. **TLS support**: yes

Continued on next page

Table 3.2 – Continued from previous page

Feature	AWS Lambda	Google Cloud Functions	Azure Functions
Function Orchestration	**Function Orchestrator**: AWS Step Functions. **Function Workflows Definition**: custom DSL. **Orchestrating Function Languages**: -. **Control Flow Constructs Documentation**: Yes. **Function Workflow Execution Time Quota**: Yes. **Function Workflow Task I/O Size Quota**: Yes	**Function Orchestrator**: -. **Function Workflows Definition**: -. **Orchestrating Function Languages**: -. **Control Flow Constructs Documentation**: -. **Function Workflow Execution Time Quota**: -. **Function Workflow Task I/O Size Quota**: -	**Function Orchestrator**: Azure Durable Functions. **Function Workflows Definition**: orchestrating function. **Orchestrating Function Languages**: C#, JavaScript, Python, PowerShell. **Control Flow Constructs Documentation**: yes. **Function Workflow Execution Time Quota**: no. **Function Workflow Task I/O Size Quota**: no
Observability	Amazon CloudWatch	Google Cloud Operations	Azure Application Insights

3.4 Selection of OpenWhisk

After conducting a comprehensive analysis of various serverless computing tools based on the defined evaluation criteria, OpenWhisk has been chosen as the preferred framework for the proposed solution.

3.4.1 Exclusion Of Cloud Provider Tools

This section explores the decision to exclude cloud provider tools from the architecture and the reasons behind this choice. The focus is on the potential obstacles that these tools can pose to the demonstration of feasibility, real-time capabilities, and scalability of the proposed solution. Additionally, an alternative approach is outlined to overcome these challenges.

- **Proprietary Licenses**: The proprietary license of the explored cloud provider tools may limit the ability to customize and modify the tools to fit the architecture requirements. This can hinder the objective of demonstrating the feasibility of the specific architecture. Customization and adaptation are often crucial in implementing an architecture that aligns perfectly with any needs [28].

- **Limited Installation Types**: Another factor that can impede the demonstration of feasibility is the limited installation options offered by cloud provider tools such ad AWS Lambda and Google Cloud Functions, which operate just as-a-service. Since these tools barely support any kind of deployment platforms, it becomes challenging to validate the architecture's feasibility across different environments. The architecture may need to operate on specific infrastructures, and the lack of installation flexibility within cloud provider tools restricts the ability to assess feasibility comprehensively. To address this, alternative solutions with extensive installation options are sought, ensuring compatibility with a wide range of platforms.

- **Limited Source Code Accessibility**: The closed-source nature of cloud provider tools such as AWS Lambda and Google Cloud Functions prevents accessing and modifying their underlying source code. Understanding the tool being used is critical in implementing the architecture effectively. Limitations imposed by closed-source tools hinder the ability to adapt and fine-tune the tools to meet specific requirements, potentially compromising the architecture's feasibility. By opting for alternative solutions with open-source availability, access to the tool's source code is gained, making customization and adaptation feasible [29].

The exclusion of cloud provider tools from the architecture is a deliberate decision driven by the need to address specific challenges that can hinder the demonstration of feasibility. Adopting alternative solutions with open-source licensing, installation flexibility and source code accessibility allows for closer alignment of the tools with the architectural objectives. This approach enhances the ability to validate the feasibility of the architecture.

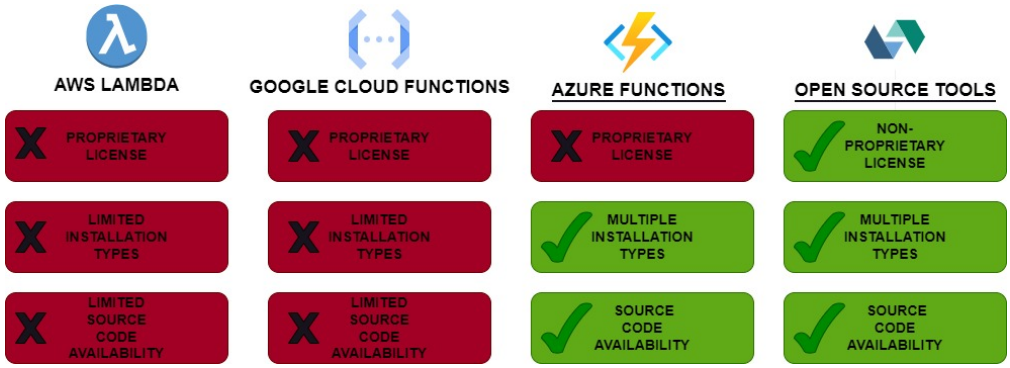

Figure 3.8: Exclusion Of Cloud Provider Tools

3.4.2 Exclusion of OpenFaaS and Knative

This section explores the decision to exclude OpenFaaS and Knative and the reasons behind this choice. The focus is on the specific advantages of OpenWhisk over OpenFaaS and Knative, highlighting how these advantages directly contribute to achieving the demonstration of feasibility of the proposed solution. Some features that appear in the matrix will not be compared, since they are almost equivalent between OpenFaaS, Knative and OpenWhisk.

OpenWhisk vs. OpenFaaS

- **Language Support**: OpenWhisk supports a broader range of programming languages compared to OpenFaaS, including Ballerina, Binary, Go, Java, .NET, Node.js, PHP, Python, Ruby, Rust, Shell, and Swift. This language diversity provides developers with more flexibility in choosing the most suitable language for their components, enhancing productivity and integration possibilities. By supporting multiple languages, OpenWhisk enables us to evaluate the feasibility of the architecture by accommodating a wider range of existing codebases and developer preferences. This is important also considering the possibility of further studies on the proposed architecture.
- **Function Orchestration**: OpenWhisk provides built-in function orchestration capabilities through Apache OpenWhisk Composer. In contrast, OpenFaaS does not seem to offer a native orchestration solution. The availability of function orchestration simplifies the coordination and sequencing of functions, enabling developers to create more complex workflows. This capability is crucial for evaluating the feasibility of the architecture's workflow management capabilities, as it allows the developer to define and analyze intricate function compositions. This is important also considering the possibility of further studies on the proposed architecture.

- **Event Sources**: OpenWhisk supports a wider range of event sources compared to OpenFaaS. The broader support for event sources allows for more diverse data ingestion and processing scenarios. By incorporating different event-driven workflows, OpenWhisk enables us to evaluate the architecture's capabilities in handling various real-world use cases, contributing to the demonstration of feasibility. This is important also considering the possibility of further studies on the proposed architecture.

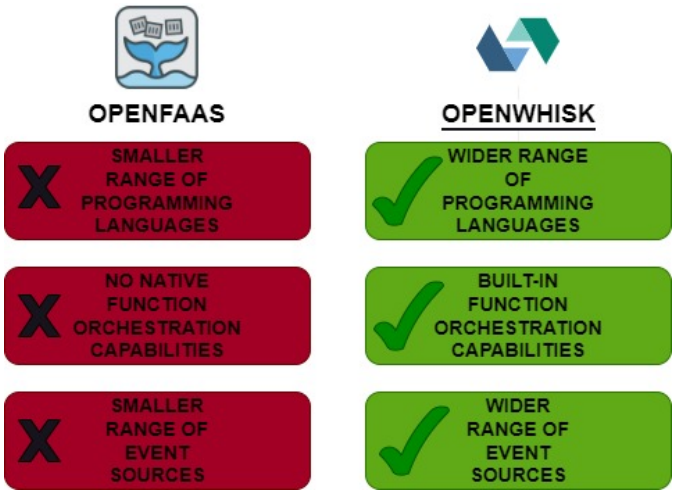

Figure 3.9: Exclusion Of OpenFaaS

OpenWhisk vs. Knative

- **Installation Flexibility**: OpenWhisk supports multiple installation environments, including Docker, Kubernetes, Linux, macOS, Windows, and Mesos. In contrast, Knative is primarily designed for Kubernetes-based deployments. OpenWhisk's broader installation options make it easier to deploy and manage the FaaS tool in various target hosts, reducing deployment complexities and increasing the feasibility of the architecture in different environments. This flexibility aligns with the objective of demonstrating the feasibility of the architecture by ensuring that OpenWhisk can be seamlessly integrated into different infrastructure setups. This is important also considering the possibility of further studies on the proposed architecture.

- **Community Support**: OpenWhisk has a more extensive and active community compared to Knative, as evident from its higher number of GitHub stars, forks, issues, and Stack Overflow questions. The vibrant community around OpenWhisk provides developers with a wealth of resources, expertise,

and support. This strong community support is essential for the objective of evaluating feasibility as it allows us to tap into a vast knowledge base, seek assistance in overcoming challenges, and collaborate with other users, ultimately contributing to the successful implementation and assessment of the architecture.

- **Observability**: OpenWhisk offers observability features such as Kamon, Prometheus, and Datadog for logging, monitoring, and metrics collection. While Knative doesn't seem to provide built-in observability capabilities, OpenWhisk's built-in support for these tools simplifies the setup and configuration process. The availability of robust observability features enhances the feasibility assessment by providing valuable insights into the architecture's performance, scalability, and resource utilization. By easily monitoring and analyzing the behavior of the system, OpenWhisk empowers us to evaluate its feasibility in terms of meeting scalability requirements. This is important also considering the possibility of further studies on the proposed architecture.

Figure 3.10: Exclusion Of Knative

By highlighting these advantages, it becomes evident that OpenWhisk surpasses OpenFaaS and Knative in terms of language support, function orchestration capabilities, event source compatibility, installation flexibility, community support, and observability features. These strengths make OpenWhisk the preferred choice for evaluating the feasibility and practicality of the architecture, while also defining the architecture itself.

3.4.3 OpenWhisk: The Chosen Tool for Feasibility Assessment

After carefully evaluating various open-source options and excluding cloud provider-specific tools, OpenWhisk emerged as the most suitable choice for assessing the feasibility of the architecture. In this section, it will be shown how each evaluated feature of OpenWhisk aligns with the objectives, in particular with the objective of feasibility assessment.

- **License**: OpenWhisk utilizes the Apache 2.0 license. This open-source license promotes customization, collaboration, and innovation, enabling developers to modify and enhance the FaaS tool to align with the specific requirements of the architecture. The open nature of the license facilitates experimentation, optimization, and integration efforts, contributing to the feasibility and practicality of the architecture.

- **Installation Type:** OpenWhisk can be installed on various target hosts, including Docker, Kubernetes, Linux, macOS, Windows, and Mesos. This compatibility with multiple installation environments allows developers to seamlessly deploy and manage the architecture in their preferred environment. A smooth installation process and compatibility with different platforms reduce deployment complexities, streamline management efforts, and contribute to the overall feasibility of the architecture.

- **Source Code Availability**: OpenWhisk provides an open-source repository on GitHub, allowing developers to access and modify the source code. This accessibility facilitates customization and integration with other components within the architecture. Developers can experiment with different configurations, optimize interactions between components, and ensure a well-integrated system. The availability of source code fosters collaboration, enhancing the feasibility and effectiveness of the architecture.

- **Main Programming Language**: OpenWhisk supports a wide range of programming languages, including Ballerina, Binary, Docker Image, Go, Java, .NET, Node.js, PHP, Python, Ruby, Rust, Shell, and Swift. This flexibility accommodates the preferences and expertise of developers, enabling the selection of the most appropriate language for each component. Compatibility with multiple programming languages fosters a diverse ecosystem of developers, enhancing the feasibility and versatility of the architecture.

- **Interface Types**: OpenWhisk offers a variety of interfaces, including a command-line interface (CLI), application programming interface (API), and

graphical user interface (GUI). These interfaces simplify the usage, development, deployment, and management of architecture components. CLI interfaces enable automation and scripting, API interfaces facilitate programmatic control and integration with other tools, and GUI interfaces offer intuitive visual representations and management capabilities. Having multiple interface types increases the usability and accessibility of the tool, enabling developers to effectively work with the architecture and evaluate its feasibility.

- **Community Support**: OpenWhisk boasts a robust community with 16,800 GitHub Stars, 932 GitHub Forks, 274 GitHub Issues, and 2,790 Stack Overflow Questions. This active community provides valuable resources, knowledge sharing, and troubleshooting assistance. Developers can leverage the collective expertise of the community to address challenges, seek guidance on best practices, and overcome implementation obstacles. The strong community support accelerates development and ensures successful implementation of the architecture, contributing to its feasibility.

- **Documentation**: OpenWhisk offers comprehensive and well-maintained documentation, covering various aspects such as platform usage, development, deployment, architecture, and functions deployment. This documentation provides a clear understanding of the tool's features, capabilities, and best practices, guiding developers in utilizing it to its full potential. Detailed documentation enhances the feasibility of the architecture by reducing implementation errors, facilitating knowledge transfer.

- **Event Sources**: OpenWhisk's support for various event sources, such as Scheduler, Apache Kafka, GitHub, and more, plays a crucial role in demonstrating the feasibility of the architecture's scalability. By enabling developers to trigger functions based on specific events or external triggers, OpenWhisk allows the architecture to scale dynamically in response to workload demands. This ability to handle events in real time and scale resources accordingly showcases the feasibility of building scalable and responsive applications using OpenWhisk.

- **Function Orchestration**: The function orchestration capabilities offered by OpenWhisk, including Apache OpenWhisk Composer, are essential for proving the feasibility of building complex workflows and demonstrating real-time properties. By allowing developers to define and manage interconnected functions as a cohesive workflow, OpenWhisk enables the orchestration of real-time processes. This capability showcases the feasibility of implementing real-time data processing, complex event-driven systems, and reactive applications within the architecture.

- **Observability**: OpenWhisk's observability features, such as integration with monitoring tools like Kamon, Prometheus, and Datadog, contribute to proving the feasibility of the architecture's real-time properties. By providing comprehensive monitoring, logging, and tracing capabilities, OpenWhisk allows developers to gain real-time insights into the system's performance and behavior. This observability enables proactive monitoring, timely troubleshooting, and optimization of the architecture's responsiveness, demonstrating its feasibility for real-time applications.

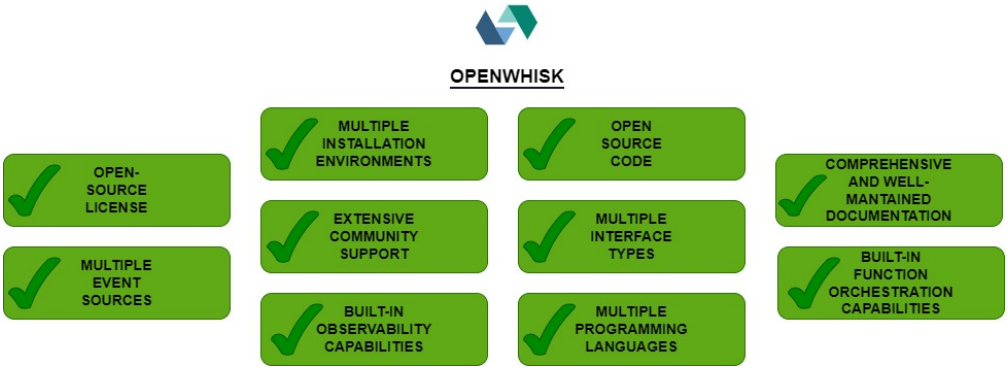

Figure 3.11: Reasons for Choosing OpenWhisk

In conclusion, OpenWhisk has been identified as the suitable tool for assessing the feasibility of the architecture based on its various features and capabilities. The criteria evaluated, such as license, installation type, source code availability, programming language support, interface types, community support, and documentation, contribute to the overall feasibility of the architecture by promoting customization, collaboration, ease of deployment, and developer productivity.

It's important to note that the assessed criteria also enable the possibility of further studies and improvements to the architecture. OpenWhisk's open-source nature and comprehensive documentation allow developers to experiment, optimize, and integrate the architecture with other components. The strong community support provides valuable resources and troubleshooting assistance, facilitating ongoing development and refinement of the architecture.

Overall, by choosing OpenWhisk and considering the evaluated criteria, it is possible to be more confident in the feasibility of the architecture and its potential for scalability and real-time capabilities. The assessed features provide a solid foundation for future studies, enhancements, and successful implementation of the architecture.

Chapter 4

Deep Dive Into OpenWhisk

This chapter provides a comprehensive exploration of OpenWhisk, an event-driven compute platform also known as Serverless Computing or Function-as-a-Service (FaaS). The aim of this chapter is to examine the various aspects of OpenWhisk, including its architecture, components, command-line interface (CLI), actions, triggers, and rules.

The chapter begins by presenting an overview of the OpenWhisk architecture, highlighting the key components that work together to enable the execution of actions and processing of events. The roles and interactions of components are explained.

Following the architecture discussion, the chapter dives into the OpenWhisk Command Line Interface (CLI). The installation and configuration of the CLI are covered, along with demonstrations of its usage for creating, updating, and deleting actions, triggers, and rules. The CLI's capability to invoke actions and monitor their execution is also explored.

The next section focuses on actions in OpenWhisk. The purpose of actions is discussed, along with their creation, management, and support for multiple programming languages. Packaging and deploying actions as self-contained entities, as well as versioning and lifecycle management, are also examined.

Finally, the chapter delves into triggers and rules in OpenWhisk, which play a crucial role in event-driven architectures and task automation. Triggers are explained, along with their ability to initiate the execution of actions based on events or conditions. The creation, management, and triggering of actions using triggers are explored. Additionally, rules and their significance in associating actions with triggers, thereby enabling the creation of event-driven workflows, are discussed.

By the end of this chapter, readers will have gained a comprehensive understanding of OpenWhisk's architecture, CLI, actions, triggers, and rules.

4.1 Introduction to OpenWhisk Components

OpenWhisk is an event-driven compute platform, also known as Serverless computing or Function as a Service (FaaS). It provides a scalable and flexible environment for running code in response to events or direct invocations [23]. The architecture of OpenWhisk is composed of several key components that work together to enable the execution of actions and processing of events. This section provides an overview of the OpenWhisk architecture and highlights the key components involved in the system.

4.1.1 Overview of OpenWhisk Architecture

The OpenWhisk architecture is designed to handle the execution of actions in response to events or direct invocations. It follows a distributed and scalable model that leverages various components to achieve efficient and reliable computation. At a high level, the architecture consists of the following components [30]:

- **Nginx**: Nginx serves as the entry point for user requests in the OpenWhisk system. It handles tasks such as SSL termination, load balancing, and routing of HTTP requests to the appropriate components within the architecture.
- **Controller**: The Controller component is the central orchestrator of the OpenWhisk system. It provides the RESTful API interface for users to interact with OpenWhisk. The Controller receives user requests, performs authentication and authorization checks, and coordinates the execution of actions by interacting with other components.
- **CouchDB**: CouchDB is a distributed NoSQL database used for storing metadata and state information in OpenWhisk. It serves as the primary data store for entities such as actions, triggers, rules, and user information. The Controller interacts with CouchDB to store and retrieve data, including action code, default parameters, resource restrictions, and authentication details.
- **Load Balancer**: The Load Balancer component plays a crucial role in distributing incoming requests across multiple Invokers. It maintains a global view of available Invokers and their health status. The Load Balancer selects an appropriate Invoker to handle each request, ensuring efficient resource utilization and load balancing across the system.

- **Kafka**: Kafka is a distributed event streaming platform used for reliable messaging in OpenWhisk. It acts as a message broker between the Controller and Invokers, buffering and persisting messages to ensure fault tolerance and scalability. Kafka enables asynchronous communication and coordination between components, allowing the Controller to publish action invocation requests and Invokers to consume and execute them.
- **Invoker**: The Invoker component is responsible for executing actions in OpenWhisk. It operates as a pool of execution environments, typically based on Docker containers. When an action invocation request is received, the Invoker selects an available container, injects the action code, and sets up the execution environment. The action code is then executed within the container, and the results are captured and stored for further processing.
- **Activations Database**: The Activations Database, stored in CouchDB, serves as a persistent storage layer for storing the results and metadata of action invocations. Each action invocation generates an activation record that contains information such as the activation ID, response status, result data, logs, and timestamps. The Activations Database allows users to retrieve past invocations and access their results.

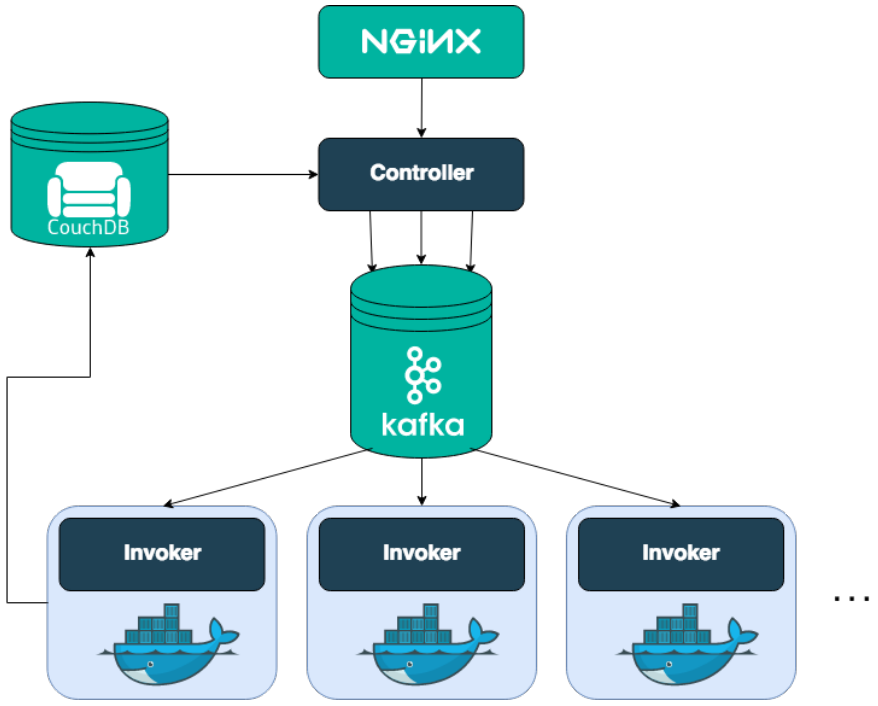

Figure 4.1: Main OpenWhisk Components

These components work together to create a scalable and event-driven compute platform in OpenWhisk. Nginx serves as the entry point, forwarding requests to the Controller, which handles request processing, authentication, and authorization. CouchDB stores metadata and state information, while the Load Balancer ensures efficient distribution of requests among Invokers. Kafka enables reliable messaging between the Controller and Invokers, and the Activations Database stores the results and metadata of action invocations for later retrieval.

4.2 OpenWhisk CLI

The OpenWhisk Command Line Interface (CLI) is a powerful tool that allows developers to interact with OpenWhisk and manage serverless applications from their local development environment. It provides a command-line interface with a rich set of commands for performing various operations on actions, triggers, and rules. This section provides an overview of actions in OpenWhisk CLI, including its overview, installation and interaction with actions, triggers and rules. All the informations about this paragraph have been retrieved from the same source, which can be retrieved from the bibliography with the following number: [31]

Figure 4.2: OpenWhisk CLI

4.2.1 Overview of OpenWhisk CLI

The OpenWhisk CLI is built on top of the OpenWhisk API and provides a convenient way to interact with OpenWhisk components using command-line commands. It allows you to create, update, and delete actions, triggers, and rules, as well as invoke actions and monitor their execution.

4.2.2 Working with the CLI: Installation and Configuration

To start using the OpenWhisk CLI, individuals are required to install it on their local machines. The installation process is dependent on the operating system being used. Once installed, the CLI can be configured by setting up authentication credentials, including the API key and endpoint URL, to establish a connection with the OpenWhisk instance. This configuration ensures secure and authorized access to the OpenWhisk resources.

4.3 Actions in OpenWhisk

Actions are the fundamental building blocks in OpenWhisk that encapsulate units of computation. They represent individual pieces of code that can be executed in response to events or direct invocations. This section provides an overview of actions in OpenWhisk, including their purpose, creation, management, supported programming languages, packaging, deployment, and versioning. All the informations about this section have been retrieved from the same source, which can be retrieved from the bibliography with the following number: [32]

4.3.1 Understanding Actions in OpenWhisk

Actions in OpenWhisk are the fundamental units of work that can be executed in response to events or direct invocations. An action in OpenWhisk can be a small snippet of code or a custom binary code embedded in a Docker container. Actions are designed to be stateless, self-contained, and independently executable. They can be written in various programming languages, allowing developers to choose the language they are most comfortable with.

Actions in OpenWhisk are executed whenever a trigger fires, making them event-driven. They can also be directly invoked using the OpenWhisk API, CLI, or SDKs. Actions can be combined and chained together to create complex workflows without the need to write additional code. The output of one action can be passed as input to the next action in the sequence.

4.3.2 Creating and Managing Actions

Creating and managing actions in OpenWhisk is a straightforward process. Actions can be created using the OpenWhisk CLI or API, allowing developers to define the code and specify any required parameters or dependencies. Actions can be associated with triggers to enable event-driven execution.

Once created, actions can be managed using various commands and APIs provided by OpenWhisk. This includes updating the code or parameters of an action, enabling or disabling actions, and retrieving information about actions such as their status and invocation history.

Considering the OpenWhisk CLI to interact with actions, they can be created using the following command:

```
$ wsk action create <action-name> <action-file>
```

Where `<action-name>` is the name of the action and `<action-file>` is the file containing the code. Actions can be updated using the following command:

```
$ wsk action update <action-name> <action-file>
```

Where `<action-name>` is the name of the action and `<action-file>` is the file containing the code. To list all the actions in a specific OpenWhisk namespace, the following command can be used:

```
$ wsk action list
```

This command provides an overview of the available actions, including their names and associated parameters. Actions can be deleted using the following command:

```
$ wsk action delete <action-name>
```

Where `<action-name>` is the name of the action.

4.3.3 Supported Programming Languages

OpenWhisk supports multiple programming languages, providing developers with flexibility in choosing the language that best suits their needs and expertise. Some of the programming languages supported by OpenWhisk include:

- Node.js (JavaScript)
- Python

- Java
- Swift
- PHP
- Go

Each programming language has its own runtime environment and dependencies, allowing developers to write actions in the language they are most comfortable with. The OpenWhisk runtime environment takes care of executing the actions and managing their lifecycle.

4.3.4 Packaging and Deploying Actions

In OpenWhisk, actions can be packaged and deployed as self-contained entities, making it easy to share and reuse them across different applications and environments. Packaging actions involves bundling the code and any required dependencies into a single artifact.

OpenWhisk provides support for packaging actions using Docker containers. Actions can be defined as custom binaries embedded within a Docker container, allowing developers to specify the necessary runtime environment and dependencies. Docker provides isolation and portability, ensuring consistent execution of actions across different environments.

4.3.5 Versioning and Managing Action Lifecycle

Managing the lifecycle of actions is an important aspect of developing applications with OpenWhisk. OpenWhisk allows developers to version their actions to maintain different iterations and track changes over time. Versioning helps in maintaining backward compatibility and enables easy rollback to previous versions if needed.

To manage the lifecycle of actions, OpenWhisk provides commands and APIs for creating, updating, and deleting versions of an action. Additionally, developers can invoke specific versions of an action to ensure consistent behavior across different invocations.

4.3.6 Interaction Between Components To Invoke an Action

Invoking an action in OpenWhisk involves a series of interactions between different components of the system. Let's explore the flow of processing that occurs behind the scenes when invoking an action [30]:

1. **Entering the system: Nginx**
   The user initiates the action activation process by sending an HTTP request to OpenWhisk's user-facing API. This request is intercepted by Nginx, an HTTP and reverse proxy server, which forwards it to the next component in the system.

2. **Controller**
   The request is then passed on to the Controller, which serves as the interface for user interactions in OpenWhisk. The Controller receives the request and determines the action activation operation based on the HTTP method used.

3. **Authentication and Authorization: CouchDB**
   Before proceeding with the action activation, the Controller verifies the user's identity and checks if the user has the necessary privileges. This authentication and authorization process involves validating the user's credentials against a CouchDB database, ensuring that the user has the required permissions to invoke the action.

4. **Getting the action: CouchDB**
   As the Controller is now sure the user is allowed in and has the privileges to invoke his action, it actually loads the action from the whisks database in CouchDB. The record of the action contains mainly the code to execute and default parameters that are passed to the action, merged with the parameters included in the actual invoke request. It also contains the resource restrictions imposed on it in execution, such as the memory it is allowed to consume.

5. **Load Balancer**
   After the action activation is successfully validated, the Load Balancer component comes into play. The Load Balancer maintains a global view of the available resources in the system, specifically the invokers responsible for executing actions. It selects an available invoker to handle the execution of the newly created action.

6. **Invoker and Docker**
   The selected invoker creates a Docker container, which provides a self-encapsulated environment for executing the action. The action's code is injected into the Docker container, and any required dependencies are resolved. The invoker then triggers the execution of the action within the container, passing the necessary parameters.

7. **Execution and Result Storage: CouchDB**
   The action code executes within the Docker container, producing a result. The invoker captures the output, including any logs generated during the execution,

and stores it in a CouchDB database. The result, along with metadata such as the activation ID and timestamps, is stored as an activation record.

8. **Retrieving the Result**
   To access the result of the action execution, the user can make another HTTP request or use the wsk CLI. By providing the activation ID associated with the action, the user can retrieve the corresponding activation record from the CouchDB database, obtaining the execution result and any additional information.

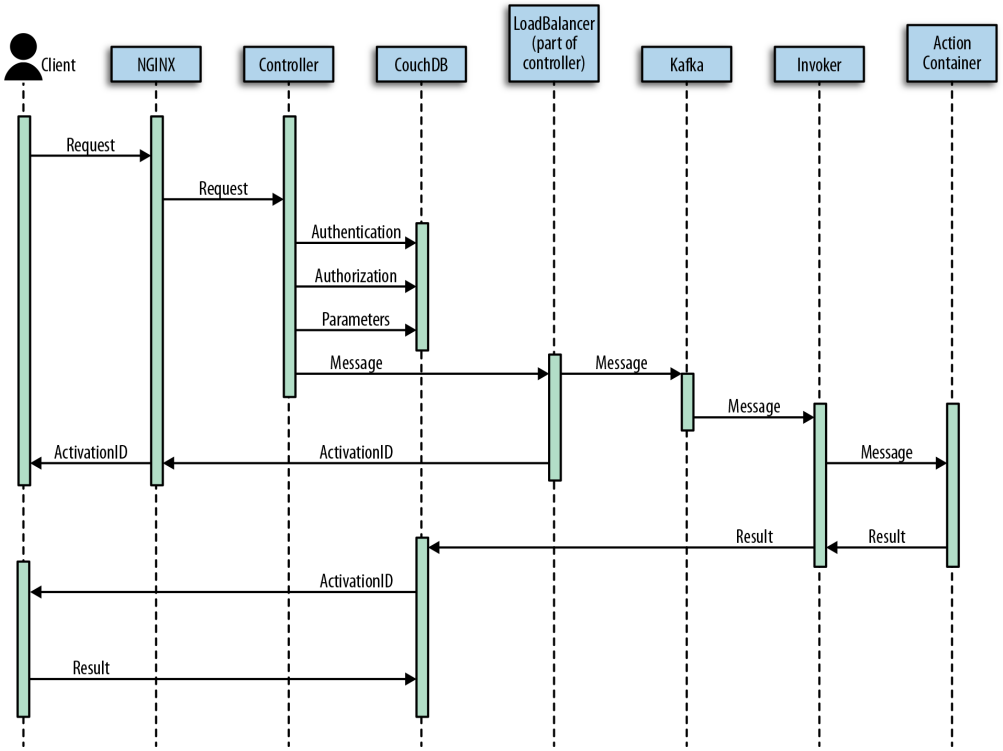

Figure 4.3: OpenWhisk Action Execution Process

This interaction between different components within OpenWhisk enables the seamless creation and execution of actions, providing users with a scalable and efficient serverless computing experience.

4.4 Triggers and Rules in OpenWhisk

Triggers in OpenWhisk play a crucial role in enabling event-driven architectures and reactive systems. They give the possibility to initiate the execution of actions

based on specific events or conditions. By associating actions to triggers with rules, it is possible to build event-driven workflows and automate the execution of tasks in response to events. This section provides an overview of triggers and rules in OpenWhisk, including their purpose, creation, management, the action triggering, supported types, and combination with rules. All the informations about this paragraph have been retrieved from the same source, which can be retrieved from the bibliography with the following number: [33]

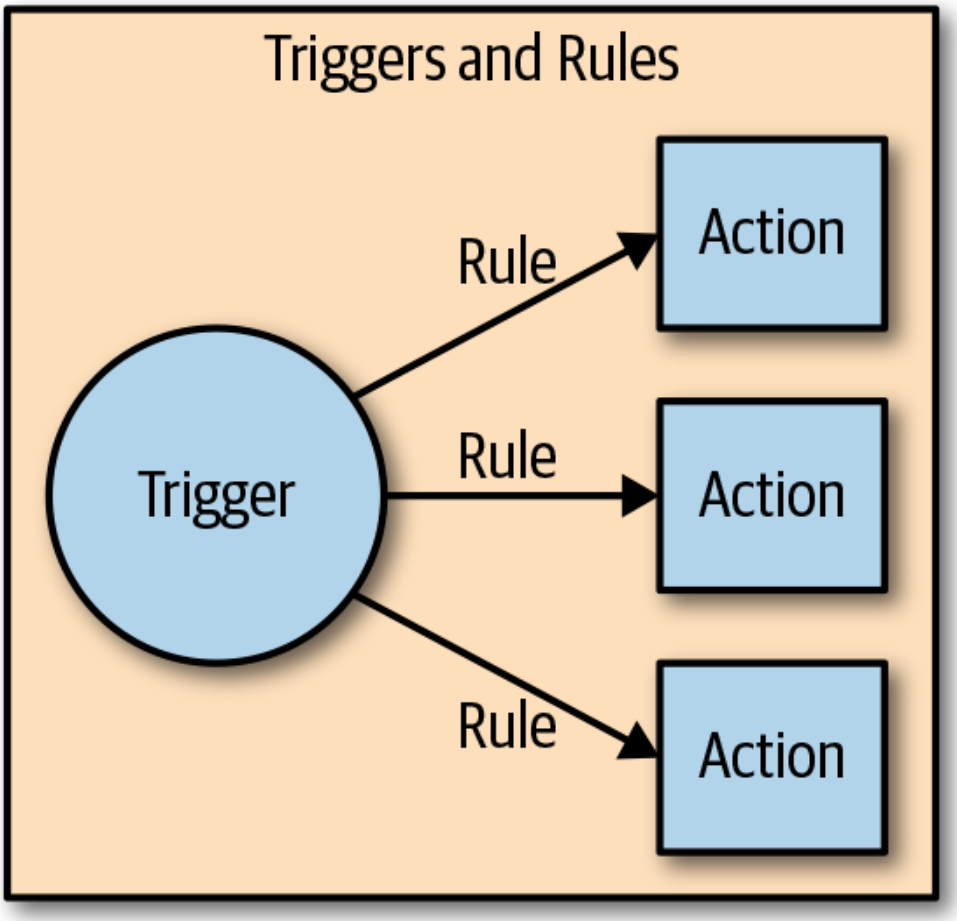

Figure 4.4: OpenWhisk Triggers and Rules

4.4.1 Creating and Managing Triggers and Rules

Creating and managing triggers and rules in OpenWhisk is a straightforward process. Triggers and rules can be created using the OpenWhisk CLI or API. Considering the OpenWhisk CLI, it is possible to do several things to create and manage triggers

and rules. Triggers can be created using the following command:

```
$ wsk trigger create <trigger-name>
```

where `<trigger-name>` is the name of the trigger. Triggers can be updated using the following command:

```
$ wsk trigger update <trigger-name>
```

where `<trigger-name>` is the name of the trigger. To list all the triggers in a specific OpenWhisk namespace, the following command can be used:

```
$ wsk trigger list
```

This command provides an overview of the available triggers, including their names and associated parameters. The following command can be used to delete a trigger:

```
$ wsk trigger delete <trigger-name>
```

where `<trigger-name>` is the name of the trigger. To create a rule, the following command can be used:

```
$ wsk rule create <rule-name> <trigger-name> <action-name>
```

where `<rule-name>` is the name of the rule, `<action-name>` is the name of the action and `<trigger-name>` is the name of the trigger. This command gives the opportunity to specify the name of the rule and the trigger and action associated with it. The rule defines the relationship between a trigger and an action, specifying that the action should be executed when the trigger fires. To update an existing rule, the following command can be used:

```
$ wsk rule update <rule-name> <trigger-name> <action-name>
```

where `<rule-name>` is the name of the rule, `<action-name>` is the name of the action and `<trigger-name>` is the name of the trigger. This command enables the modifications of the properties or configurations of a rule, such as changing the associated trigger or action. To list all the rules in a specific OpenWhisk namespace, the following command can be used:

```
$ wsk rule list
```

This command provides an overview of the available rules, including their names, associated triggers, and actions. To remove a rule, the following command can be used:

```
$ wsk rule delete <rule-name>
```

where `<rule-name>` is the name of the rule, `<action-name>` is the name of the action and `<trigger-name>` is the name of the trigger. This command will delete the rule and its associated trigger-action relationship.

4.4.2 Triggering Actions with Triggers

Once a trigger is created, one or more actions can be associated with it. When the trigger is fired or activated, the associated actions are executed. In this way, specific actions that should be triggered in response to certain events can be defined.

4.4.3 Supported Trigger Types

OpenWhisk supports various types of triggers to accommodate different event sources. Some of the supported trigger types include feed-based triggers, periodic triggers, and web triggers.

4.4.4 Combining Trigger with Rules for Event-Driven Workflows

In OpenWhisk, triggers can be combined with rules to create event-driven workflows. With rules it is possible to define conditions that determine when actions associated with a trigger should be executed. By combining triggers and rules, it is possible to build complex event-driven systems and automate workflows based on specific events or conditions.

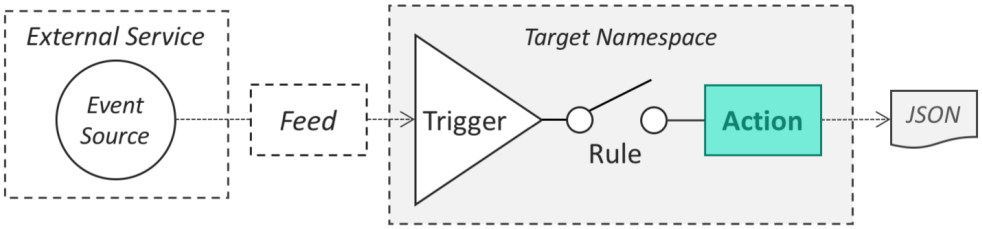

Figure 4.5: Trigger-Rule-Action Mechanism in OpenWhisk

4.5 OpenWhisk Packages

This section presents a comprehensive examination of OpenWhisk packages, which form an integral part of the OpenWhisk serverless platform. OpenWhisk packages enable the bundling of actions and feeds, facilitating the organization, sharing, and reuse of functionalities within the OpenWhisk ecosystem. Throughout this chapter, the concept of OpenWhisk packages will be explored, their components will be examined, and their potential for enhancing application development and collaboration will be elucidated. Moreover, package organization, browsing, creation, sharing, and their significance in constructing serverless applications will

be discussed. The information provided in this chapter is derived from a specific resource, which can be referenced in the bibliography under the number [34].

4.5.1 Introduction to OpenWhisk Packages

In OpenWhisk, packages play a crucial role in bundling together a collection of related actions and feeds. They provide a convenient way to organize and share sets of functionalities with other users. A package can consist of various actions and feeds, each serving a specific purpose within the package.

4.5.2 Action In OpenWhisk Packages

Actions in OpenWhisk packages are individual pieces of code that can be executed within the OpenWhisk environment. These actions encapsulate specific functionalities and can be designed to perform various tasks. For example, the Cloudant package includes actions to read and write records to a Cloudant database. By including these actions in a package, users can easily access and utilize them as part of their applications.

4.5.3 Feeds in OpenWhisk Packages

Feeds are an essential component of OpenWhisk packages and are used to configure external event sources to trigger actions within the OpenWhisk system. A feed within a package allows users to integrate external event-driven data into their workflows. For instance, the Alarm package includes a feed that can fire a trigger at a specified frequency, enabling users to schedule and automate actions based on time intervals.

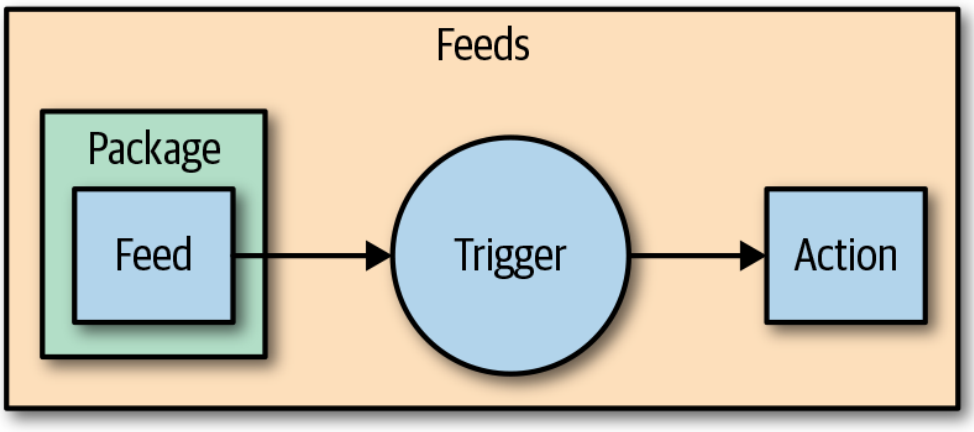

Figure 4.6: Feeds in OpenWhisk Packages

4.5.4 Packages Organization and Namespace

In OpenWhisk, every entity, including packages, belongs to a namespace. The fully qualified name of an entity follows the format `/namespaceName[/packageName]/entityName`. This hierarchical structure allows for efficient organization and management of packages, making it easier to locate and reference specific functionalities within the system.

4.5.5 Browsing and Utilizing Packages

OpenWhisk provides various tools and commands to browse and interact with packages. Users can obtain a list of packages within a specific namespace, list the entities within a package, and obtain detailed descriptions of individual entities within a package. For example, by executing the command:

```
$ wsk package list /whisk.system
```

users can retrieve a list of packages in the `/whisk.system` namespace, including the MQTT provider and Alarm packages.

4.5.6 MQTT and Alarm Packages

The MQTT package integrates MQTT functionality into OpenWhisk applications, enabling seamless communication between devices and applications in IoT scenarios. Use cases for the MQTT Provider package include IoT data ingestion and real-time event processing [35].

The Alarm package in OpenWhisk provides a convenient way to schedule and trigger actions at specific times or intervals. The Alarm package is useful for implementing scheduled tasks, event-driven automation, and periodic data processing within OpenWhisk applications [36].

4.6 Load Balancer

The Load Balancer in OpenWhisk is responsible for distributing incoming workloads across multiple invokers to ensure efficient utilization of resources and optimal performance. One of the load balancing strategies employed by OpenWhisk is the `ShardingContainerPoolBalancer`, which utilizes a hashing algorithm for workload scheduling. All the following informations can be retrieved directly from the source, which can be found in the bibliography at this number: [37].

4.6.1 Introduction to Load Balancer

The Load Balancer is a critical component in OpenWhisk's architecture, designed to evenly distribute incoming workloads across available invokers. By balancing the load, it ensures that resources are utilized efficiently and that functions are executed in a timely and responsive manner.

4.6.2 ShardingContainerPoolBalancer Overview

The `ShardingContainerPoolBalancer` is a load balancing strategy employed by OpenWhisk. It uses a hashing algorithm to distribute workloads among invokers based on a calculated hash value. This strategy aims to minimize collisions and evenly distribute the workload across available resources.

4.6.3 Algorithm Explanation

The `ShardingContainerPoolBalancer` algorithm determines the home-invoker for a given namespace and action pair by calculating a hash value. The hash value is used to select the initial invoker, and if that invoker is healthy and has available capacity, the request is scheduled to it. If the initial invoker doesn't meet the requirements, the algorithm increments the index by a step-size and checks the next invoker in the progression. This process continues until all invokers have been checked, at which point the load balancer employs an "overload" strategy to handle situations where no invoker meets the criteria.

4.6.4 Capacity Checking and User-Memory Configuration

To ensure efficient load balancing, the Load Balancer performs capacity checking on the invokers. It determines the maximum capacity per invoker based on various factors such as available resources and configured limits. Additionally, the Load Balancer takes into account the user-memory configuration to allocate appropriate resources for each invocation, considering the needs and constraints of different users.

4.6.5 Invoker Health Checking

The Load Balancer actively monitors the health status of the invokers. It performs periodic health checks, including system error checks and ping responses, to ensure that the invokers are functioning properly and can handle incoming workloads. If an invoker is found to be unhealthy, it is temporarily removed from the pool until it becomes available again.

Chapter 5

Use Case Definition

In this section, a specific use case scenario will be defined for monitoring multiple data centers using sensor data. The use case scenario will be described, and the requirements for its implementation will be outlined. This analysis aims to provide insights into the monitoring process and offer guidance for developing a solution that meets the identified needs and objectives.

5.1 Use Case Scenario

The use case scenario involves monitoring multiple data centers using sensor data. In this scenario, each data center is equipped with sensors that capture temperature, gas percentage, and humidity readings. The goal is to continuously monitor the health and performance of the data centers by collecting and analyzing sensor data in real-time.

The data centers are geographically distributed and can be located in different regions or countries. Each data center houses critical infrastructure and servers, and it is essential to ensure that they operate within optimal conditions to avoid downtime, equipment failure, or safety hazards.

To achieve effective monitoring, an architecture is designed to collect, process, and visualize the sensor data from each data center. The architecture utilizes IoT sensors, a messaging protocol, a serverless computing platform, and a data visualization tool.

The data is then processed and analyzed in real-time using a serverless computing platform, like OpenWhisk, which triggers actions to perform operations such as data validation, anomaly detection, and aggregation.

The processed data is stored in a database, such as InfluxDB, enabling historical analysis and trend identification. Finally, a data visualization tool like Grafana is used to create customizable dashboards that display real-time and historical sensor data, allowing operators and administrators to monitor the data center conditions effectively.

By continuously monitoring the sensor data from multiple data centers, potential issues or anomalies can be detected early on. This enables proactive maintenance and timely response to critical situations.

The use case scenario of monitoring multiple data centers using sensor data is particularly relevant in industries that rely heavily on data center operations, such as cloud computing providers, telecommunications companies, and large-scale enterprises. Ensuring the smooth operation of data centers is crucial to maintain service availability, meet SLAs (Service Level Agreements), and prevent costly downtime.

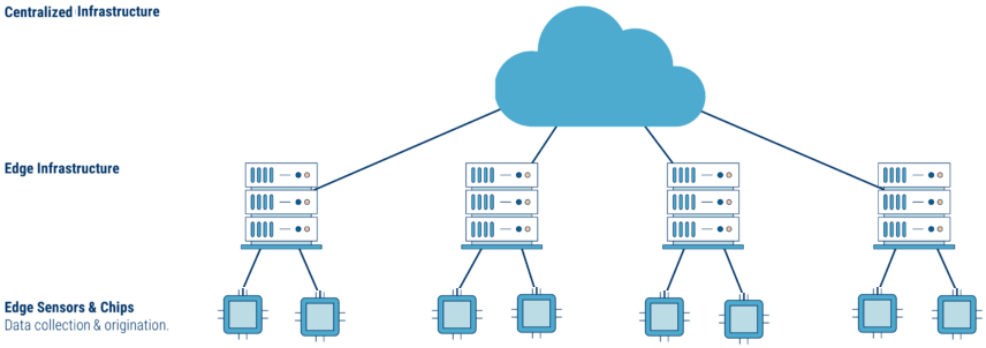

Figure 5.1: Use Case General Architecture

5.2 Use Case Scenario Adaptability

The described use case of monitoring multiple data centers using sensor data can be easily adapted to various other scenarios beyond data center monitoring. The architecture and components involved can be leveraged to monitor and analyze sensor data in different contexts, such as environmental monitoring [38], air quality assessment [38], plant growth optimization [39], and more. By replacing or adding appropriate sensors and configuring the data processing logic, organizations can extend the use case to address specific monitoring requirements in different domains.

In the context of environmental monitoring [38], the architecture can be applied

to capture data related to air quality parameters, including pollutant levels, particulate matter, and carbon dioxide concentration. By deploying suitable sensors and leveraging the existing architecture and processing pipeline, organizations can collect, analyze, and visualize environmental sensor data in real-time. This empowers them to monitor air quality in urban areas, industrial sites, or indoor environments, enabling timely interventions and decision-making to improve air quality and ensure the well-being of individuals.

Similarly, in agricultural settings [39], the architecture can be adapted to monitor plant health and optimize growth conditions. By deploying sensors to measure parameters such as soil moisture, temperature, light intensity, and nutrient levels, organizations can collect valuable data. The collected sensor data can be processed and analyzed to provide insights into the health and growth status of plants. This information enables farmers to make informed decisions regarding irrigation, fertilization, and pest control, ultimately leading to improved crop yield and resource utilization.

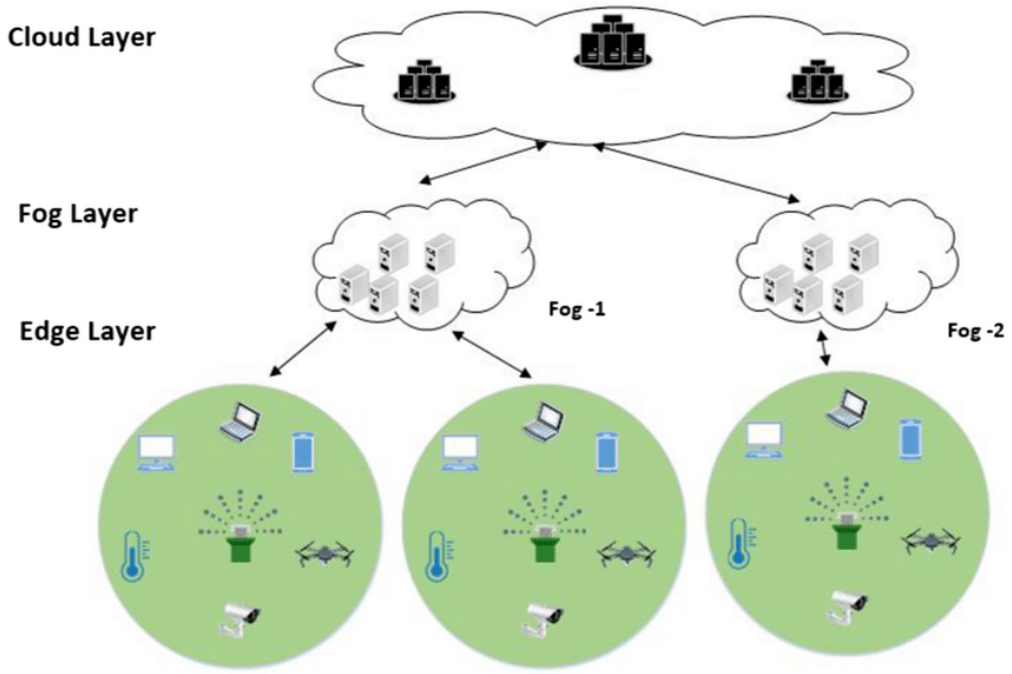

Figure 5.2: Use Case Adapted to Agriculture

The versatility of the architecture and components outlined in this use case allows organizations to tailor the solution to meet the specific monitoring needs of various domains and applications. The ability to adapt and extend the use case to different situations highlights its flexibility, making it a valuable framework for diverse

monitoring requirements.

Chapter 6

Architecture and Implementation Details

In this chapter, an exploration will be conducted into the architecture and implementation details of the use case, aiming to provide insights into the design and components involved. The overall architecture, including all the different components, will be examined, emphasizing their roles in enabling the monitoring and analysis of multiple data centers using sensor data.

The implementation details will be delved into, focusing on two main aspects: the main action mechanism and the interaction between components. The main action mechanism will outline the primary steps, processes, and logic employed in data processing, validation, and aggregation. This will provide an understanding of how the collected sensor data is transformed into meaningful insights.

The interaction between components will be explored, highlighting how they collaborate and exchange data within the architecture. This analysis will shed light on the seamless communication and cooperation among the different components, facilitating the monitoring and analysis process.

By delving into the architecture and implementation details, a comprehensive understanding of the underlying framework and mechanisms that drive the successful monitoring of multiple data centers using sensor data will be provided.

6.1 Use Case Architecture

The use case architecture is designed to monitor multiple data centers using sensor data and consists of various components that enable data processing, storage, and

visualization. The architecture is designed to achieve specific objectives in order to demonstrate its feasibility, including scalability, real-time data processing, centralized management, and seamless integration between components. To effectively address these objectives, a deliberate decision was made to adopt a distributed approach with both centralized and edge instances.

6.1.1 Architecture Overview

The architecture consists of two main components: the **centralized management instance** and the **edge instances**. The centralized management instance serves as the core component for storing aggregate data received from the edge instances. On the other hand, the edge instances are strategically deployed close to the data sources (sensors in the data centers) to facilitate real-time data processing and aggregation.

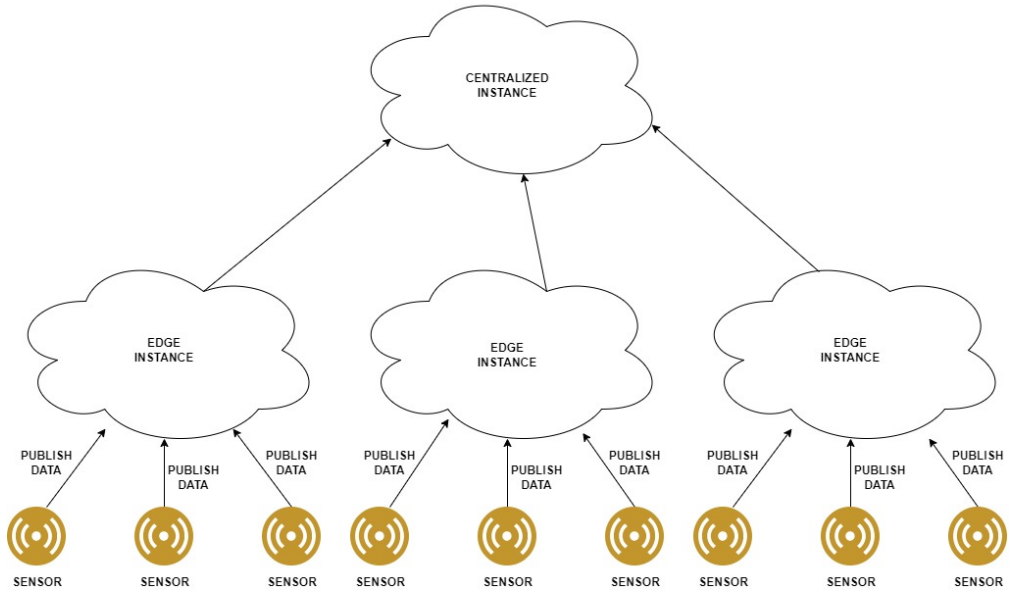

Figure 6.1: Architecture Overview

The decision to split the architecture into centralized and edge instances stems from the need to achieve scalability, real-time data processing, and centralized management. By distributing the computation load across multiple edge instances, the architecture should become inherently scalable. As the number of data centers and sensors increases, the architecture should be able to handle the growing workload by leveraging the additional edge instances. Each edge instance can handle a portion of the overall computation, allowing for parallel processing and accommodating the increased demand without overwhelming a single centralized

instance. Additionally, processing data at the edge brings computation closer to the data source, reducing the latency between data collection and processing. With edge instances handling data processing in close proximity to the sensors or data centers, the architecture should be able to achieve faster response times [40].

6.1.2 Sensor Data Publication

The architecture should enable sensors to publish the data they collect in a scalable manner. Sensors need a reliable mechanism to transmit their readings to the edge systems for further processing and analysis.

To address this challenge, the architecture incorporates an MQTT broker as the chosen instrument. The MQTT broker is deployed exclusively on the edge instances, acting as a central communication hub for the sensors.

The usage of MQTT Broker brings several advantages that align with the architecture's objectives and requirements:

- **Scalability**: The MQTT broker enables scalability by providing a unified platform for sensor data publication. As more sensors are added to the system, they can connect to the MQTT broker deployed on the edge instances. The broker efficiently handles the communication between sensors, supporting concurrent connections and horizontal scaling [41].
- **Real-time Data Processing**: By utilizing the MQTT protocol, the architecture facilitates real-time data processing. The MQTT broker enables sensors to publish messages with their data in near real-time. The lightweight publish-subscribe model of MQTT ensures minimal latency in transmitting the sensor readings to the broker. This real-time data flow allows the centralized management system to process and analyze the data as it arrives, enabling prompt decision-making and timely responses to changing conditions [42].

There are several reasons why MQTT is a good choice for implementing the MQTT broker in this architecture [43]:

- **Efficiency**: MQTT is designed to be lightweight and efficient, making it well-suited for resource-constrained environments. The protocol minimizes the overhead associated with message transmission, allowing sensors to publish their data with minimal impact on system resources.
- **Reliability**: MQTT provides reliable message delivery mechanisms, ensuring that sensor data is transmitted reliably to the MQTT broker. It supports different levels of Quality of Service (QoS), allowing the architecture to choose

the appropriate level based on the reliability requirements. This reliability ensures that no data is lost during transmission, enhancing the overall robustness of the system.

- **Scalability**: MQTT is designed to handle large-scale deployments and supports a large number of clients. It can efficiently handle the increasing number of sensors in the architecture without introducing significant complexity or performance degradation. This scalability enables the architecture to grow and adapt as more sensors are added.
- **Flexibility**: MQTT provides flexibility in terms of message routing and topic-based subscriptions. It allows sensors to publish messages to specific topics, and subscribers can choose to receive messages based on their interests. This flexibility enables selective data consumption and reduces unnecessary network traffic, enhancing the overall efficiency of the system.

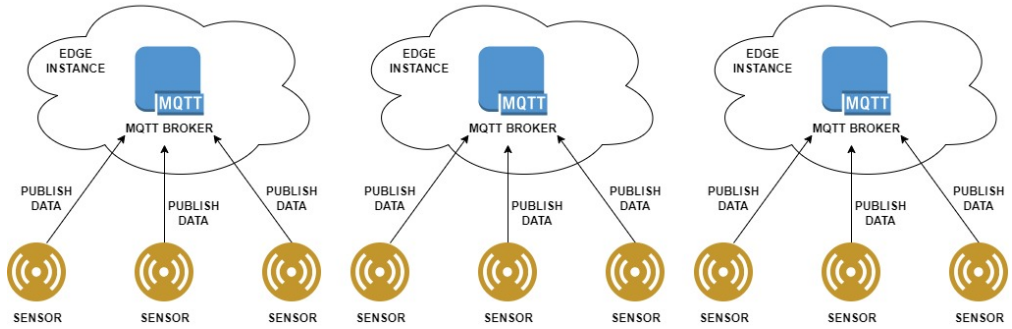

Figure 6.2: Sensor Data Publication

6.1.3 Data Storage for Sensor Readings

As sensors generate data in the data centers, there arises a need for a storage solution to capture and retain the sensor readings. Without an appropriate data storage mechanism, the architecture would struggle to process and analyze the sensor data effectively.

To address the data storage requirement, the architecture adopts a decentralized approach by incorporating local data storage within each edge instance. This means that each edge instance has its own storage mechanism to capture and retain the sensor data locally.

This decentralized approach to data storage brings several advantages that align with the architecture's objectives and requirements:

- **Scalability**: By storing data locally in each edge instance, the architecture adopts a distributed approach. This distributed storage minimizes the need for continuous communication with the centralized management instance for every data retrieval, this approach should reduce network congestion thus improving scalability. Each edge instance can handle its own data storage, processing, and retrieval independently, this may enable the architecture to seamlessly scale as the number of data centers and sensors grows [44].
- **Real-time Data Processing**: Local data storage allows for immediate availability of the captured sensor data for processing and analysis within the edge instances. This should facilitate real-time data processing, as the edge instances can efficiently access and process the data without relying heavily on external resources or frequent communication with the centralized management instance [45].

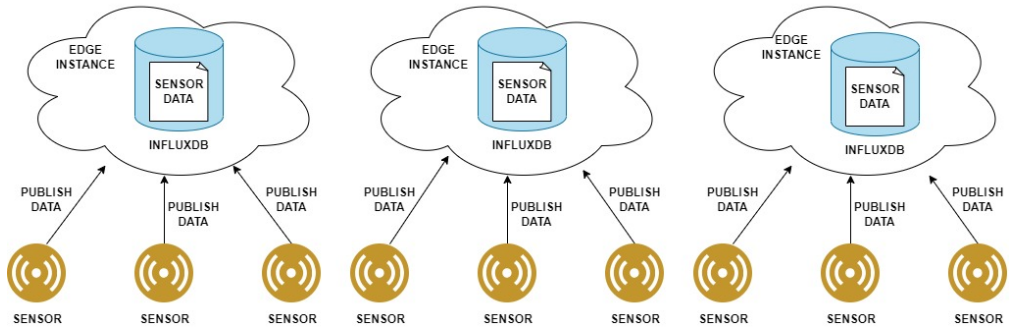

Figure 6.3: Data Storage for Sensor Readings

InfluxDB is chosen as a suitable technology for local data storage within each edge instance due to its strengths as a time-series database. Time-series databases excel at efficiently handling data points with timestamps, making InfluxDB an excellent choice for storing and retrieving sensor readings. Its design allows for optimized storage and retrieval mechanisms specifically tailored for time-based data queries. InfluxDB's architecture ensures fast and efficient data ingestion, indexing, and query execution, supporting real-time data processing requirements at the edge [46].

6.1.4 Aggregates Storage

As sensor data is collected and processed in each edge instance, there is a requirement to compute aggregates periodically for each data center. These aggregates capture meaningful information and insights from the sensor readings. The computed aggregates need to be stored and managed effectively to facilitate further

analysis.

To address the storage requirement for processed and aggregated data, the architecture employs a centralized component, such as InfluxDB, to store the aggregated data from multiple edge instances. This centralized storage approach ensures that all the computed aggregates are stored in a unified location.

The utilization of centralized aggregates storage contributes to the achievement of several key objectives of the architecture:

- **Scalability**: By using a centralized storage solution, the architecture can handle the aggregated data from multiple edge instances. As the number of data centers and sensors grows, the centralized storage should ensure scalability without overwhelming the individual edge instances.
- **Centralized Management Ease**: With a centralized storage component, the management and handling of aggregated data should become more streamlined. It provides a unified and standardized approach to storing, retrieving, and managing the aggregated data from different edge instances. By reducing the complexity of data handling, it should be possible to have an easier and stronger centralized management.

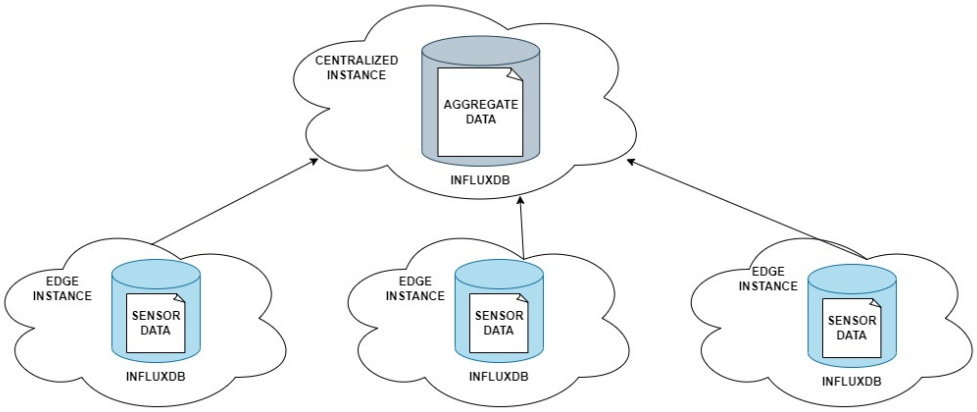

Figure 6.4: Aggregates Storage

InfluxDB is chosen as a suitable technology for the centralized aggregates storage component due to its capabilities as a time-series database. Time-series databases, like InfluxDB, are optimized for storing and querying time-stamped data efficiently. InfluxDB's design allows for the storage and retrieval of large volumes of time-series data with high performance. Its indexing and compression techniques ensure fast

and efficient access to the aggregated data. In addition, InfluxDB provides flexible query capabilities that enable data analysis and integration with other components of the architecture [46].

6.1.5 OpenWhisk in Each Instance: Centralized CouchDB and Local Caches

To address the challenges of storing sensor data in local databases and computing aggregates for subsequent storage in the centralized database, OpenWhisk actions are utilized. These actions, implemented in OpenWhisk, enable the execution of data processing tasks and the computation of aggregates. By deploying OpenWhisk in each instance, the architecture should simplify the execution of actions in a distributed and scalable manner.

A centralized CouchDB instance is employed to store the actions and trigger code for all edge instances. This approach ensures a centralized repository of actions and triggers, providing a unified and streamlined approach to manage the behavior of the OpenWhisk system across all instances. By leveraging a centralized and unique CouchDB, which is the default database of OpenWhisk, the architecture benefits from its scalability, replication capabilities, and efficient handling of document-oriented data. It is important to note that by default each OpenWhisk instance deploys its own CouchDB, which would have involved the replication of each action an trigger code: the choice of sharing a single CouchDB for the entire system was taken in order to avoid this behaviour, also in order to go towards the objective of centralized managment.

In addition to the centralized CouchDB for storing actions and triggers, each edge instance maintains a local cache. This cache is used to store frequently accessed action code, eliminating the need for continuous communication with the centralized CouchDB. By utilizing a local cache, the architecture should enhance system performance and scalability by reducing latency and network overhead. The local cache allows the edge instances to access the frequently used action code quickly, enabling faster response times for data processing and aggregation tasks.

The adoption of OpenWhisk in each instance, combined with a centralized CouchDB for actions and triggers, and local cache, contributes to achieving the objectives of the architecture:

- **Scalability**: By deploying OpenWhisk in each instance and utilizing CouchDB as the centralized storage for actions and triggers, the architecture should enforce its scalability. The distributed nature of OpenWhisk allows for parallel execution of actions across multiple edge instances, promising efficient

utilization of resources and handling of increasing workloads, which are key elements to achieve scalability [47]. Centralized CouchDB ensures fewer copies of actions and triggers across all instances, supporting scalability.

- **Real-time Data Processing**: OpenWhisk's event-driven architecture and serverless functions promise to facilitate real-time data processing. By triggering periodic actions in the edge instances, the architecture should provide timely computation of aggregates from sensor readings. On the other hand, the presence of caches in the edge instances should enable a faster retrieval of actions and triggers, without having to contact the centralized instance, promising to save the time to reach the centralized CouchDB.
- **Centralized Management Efficiency**: The choice of using a centralized CouchDB for storing actions and triggers ensures centralized management efficiency. With a single repository for actions and triggers, the architecture cleary simplifies administration, versioning, and deployment of the OpenWhisk system. Having a unique point of storage for actions and triggers of the entire system ensures that the system administrator should perform just one operation when having to update, upgrade or create something, avoiding to have to touch each single edge instance.

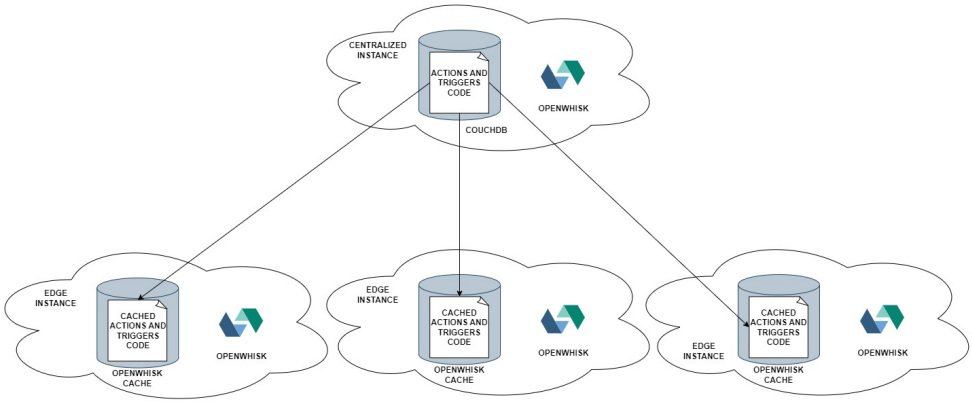

Figure 6.5: OpenWhisk in Each Instance: Centralized CouchDB and Local Caches

6.1.6 Enable Action Execution

To enable the execution of actions when sensor data is published, triggers associated with the actions need to be fired. In the case of edge instances, the MQTT Provider is utilized to trigger actions in real-time as sensor data is published. This should help ensuring immediate response and real-time data processing capabilities. Additionally, for the computation of aggregates, triggers are periodically fired using

the OpenWhisk Alarm Provider. This mechanism allows for the periodic execution of actions to compute aggregates based on sensor readings.

In the centralized instance, the MQTT Provider and OpenWhisk Alarm Provider are used to create the triggers that will be shared across all instances. This centralized approach simplifies the trigger creation process, eliminating the need to define triggers individually for each edge instance.

The operation of firing a trigger, invoking an action and creating actions and triggers can be done by making the right HTTP request to the OpenWhisk API Host, which can be found both on the edge and centralized instances. This is a default component included in the OpenWhisk deploymet.

The choice to use of OpenWhisk Alarm Provider and MQTT Provider contributes to achieving the objectives of the architecture:

- **Real-time Data Processing**: The MQTT Provider plays a crucial role in trying to achieve real-time data processing. By firing triggers associated with actions when sensor data is published, the architecture should provide fast response and processing of data at the edge.

- **Centralized Management Efficiency**: In the centralized instance, the MQTT Provider and OpenWhisk Alarm Provider are used to create triggers that are shared across all instances. This centralized trigger creation approach clearly enhances management efficiency by eliminating the need to define triggers separately for each edge instance. This approach simplifies trigger management, versioning, and deployment, leading to centralized control and streamlined management of the architecture.

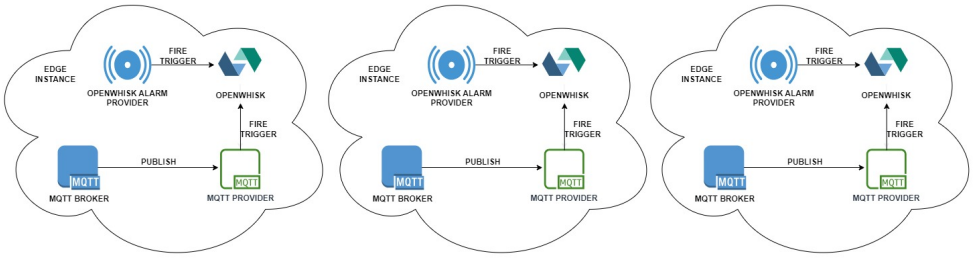

Figure 6.6: Enable Action Execution

6.1.7 Aggregates Visualization

One of the challenges is to enable administrators to monitor the computed aggregates from each data center in a centralized manner. This requires a solution that can provide a comprehensive view of the aggregated data from different locations, facilitating monitoring and analysis.

In the centralized instance, a visualization tool like Grafana is integrated to address the problem of monitoring aggregates. Grafana serves as a centralized platform for data visualization and allows to create customized dashboards to monitor the aggregated data from various data centers.

The usage of Grafana contributes to achieving the objectives of the architecture:

- **Scalability**: One of the reasons why Grafana should contribute to scalability in the architecture is that it eliminates the need to have a separate Grafana instance for each edge instance. By having a centralized Grafana installation, the architecture should handle a growing number of data centers and sensors without the need for additional resources to set up and maintain individual Grafana instances. This centralized approach should ensure scalability by reducing the overhead and complexity associated with managing multiple Grafana installations.
- **Centralized Management Ease**: Grafana's ability to create a single dashboard that can be customized and shared across multiple data centers enables centralized management of the visualization and monitoring process. Administrators can create a master dashboard template that includes all the necessary metrics and visualizations, and then easily use it for each data center by changing the data center identifier. This centralized management approach streamlines the configuration and maintenance of the dashboards, allowing administrators to manage and monitor the aggregated data from all data centers from a single point of control.

Grafana is widely recognized as a powerful and versatile visualization tool with several features that make it a good choice for integrating with the architecture [48]. Some reasons include:

- **User-Friendly Interface**: Grafana offers an intuitive and user-friendly interface, making it easy for administrators and stakeholders to create and customize dashboards according to their specific requirements. Its drag-and-drop functionality and extensive library of visualization options simplify the process of creating informative and visually appealing dashboards.
- **Real-time Monitoring**: Grafana excels in real-time data monitoring and

visualization. It provides real-time updates and supports dynamic querying of data, allowing users to view and analyze the computed aggregates as they are being processed.

- **Extensibility and Integration**: Grafana offers extensive support for various data sources and integrations. It can connect to a wide range of databases and data storage systems, including InfluxDB, which is used for storing the centralized aggregates. This flexibility enables seamless integration with the architecture's centralized component, enhancing the visualization and monitoring capabilities.

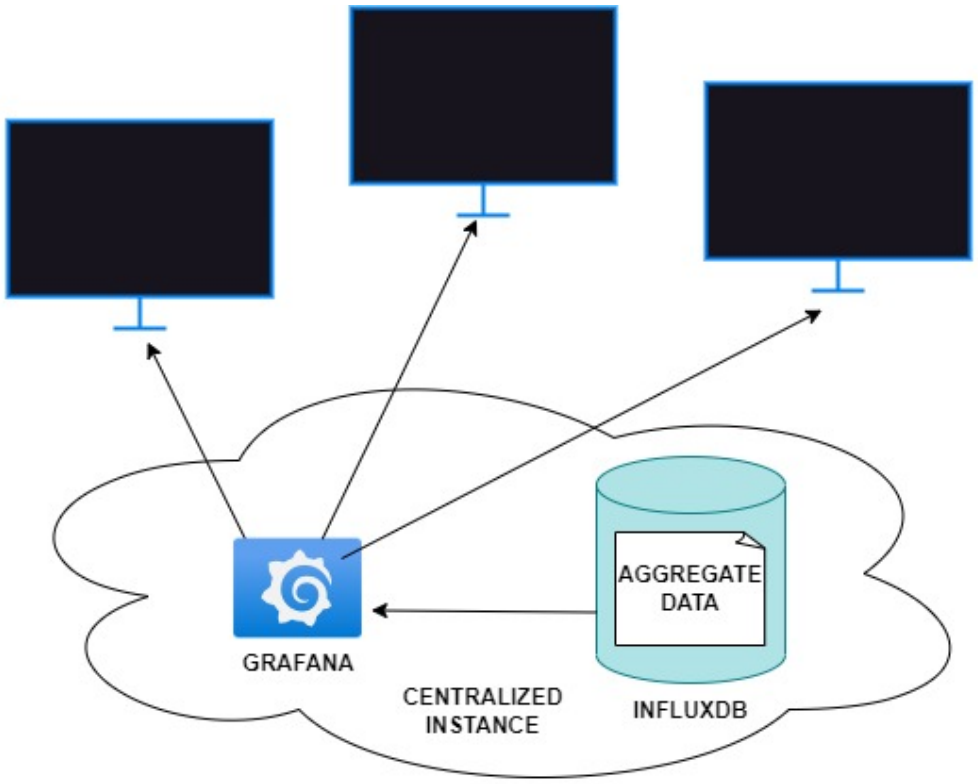

Figure 6.7: Aggregates Visualization

6.1.8 Architecture Summary

To sum up, the architecture comprises two main components: **edge instances** and the **central management instance**.

The edge instances can be deployed close to the data sources, which in this

case are the sensors in the data centers. They are responsible for processing and aggregating data in real-time and sending it to the central management instance. The main components of the edge instances include:

- OpenWhisk APIHost
- MQTT Broker
- MQTT Provider
- Local InfluxDB
- OpenWhisk Alarm Provider
- OpenWhisk Cache

The central management instance is responsible for storing and processing the data received from the edge instances. Its main components include:

- OpenWhisk APIHost
- CouchDB
- MQTT Provider
- Centralized InfluxDB
- OpenWhisk Alarm Provider
- OpenWhisk Cache
- Grafana

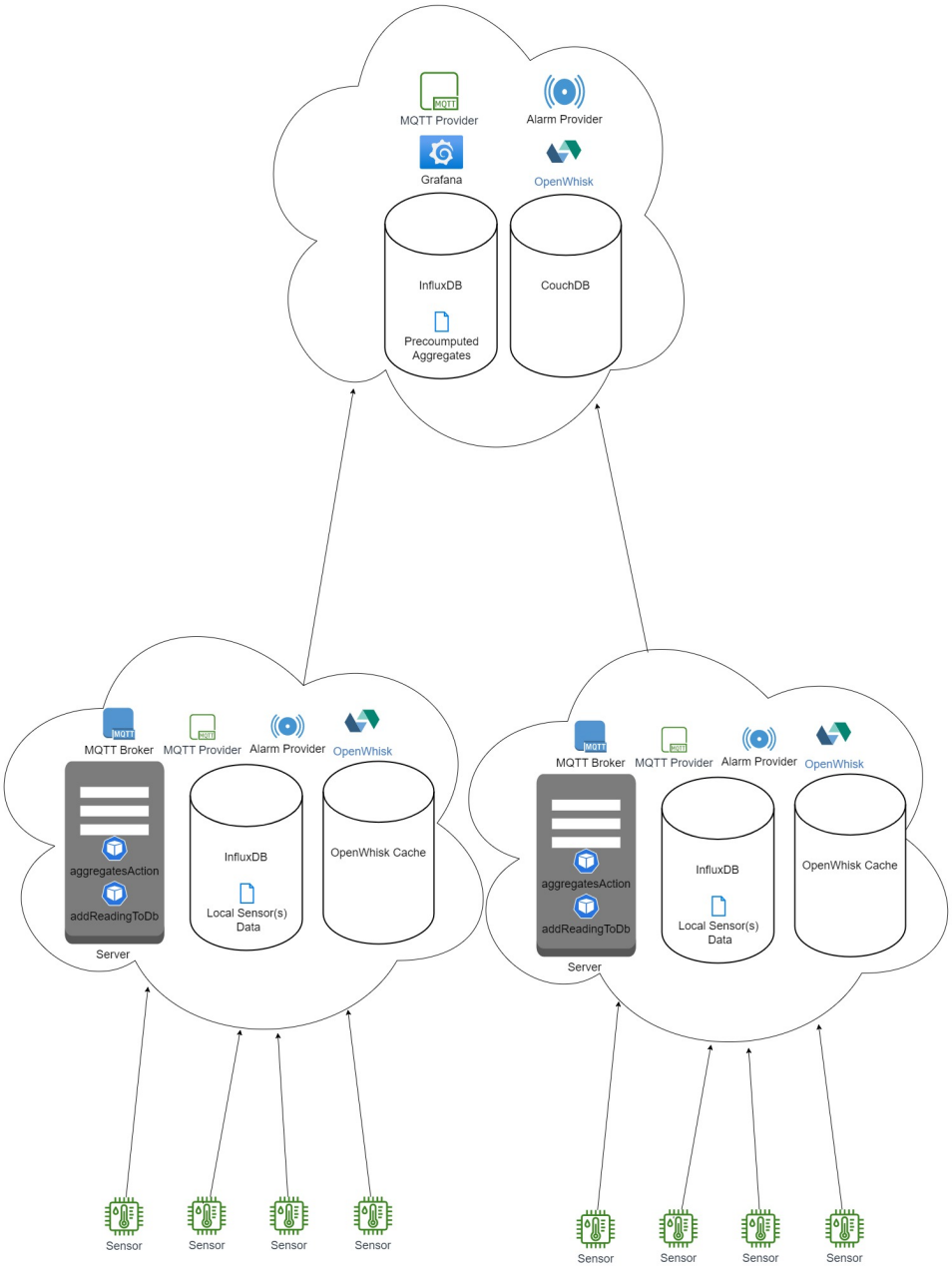

Figure 6.8: Use Case Architecture

6.1.9 Components of the Architecture

This section presents in detail the main components that are part of the architecture, providing a technical overview for each of them and presenting how they are used

in the chosen use case. The reasons why each component has been included in the architecture have already been specified in the previous section.

MQTT Broker

The MQTT Broker, specifically using the Mosquitto implementation, serves as the intermediary between the sensors and the edge instances. It acts as a lightweight, open-source message broker that receives data from the sensors and publishes them to the relevant subscribers. The broker facilitates the publish-subscribe pattern, where sensors publish their data to specific topics, and the edge instances, acting as subscribers, can subscribe to these topics to receive the data [49]. The MQTT Broker ensures efficient and reliable message delivery between the sensors and the edge instances.

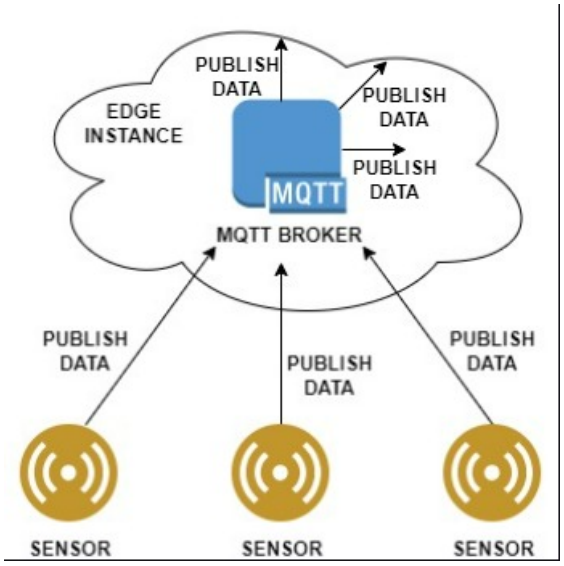

Figure 6.9: MQTT Broker

MQTT Provider

The MQTT Provider component, present in both the edge and central instances, plays a critical role in managing triggers and handling MQTT messages. It creates, updates, and deletes specific triggers and listens to incoming MQTT messages, ensuring the appropriate triggers are fired to initiate further actions [35].

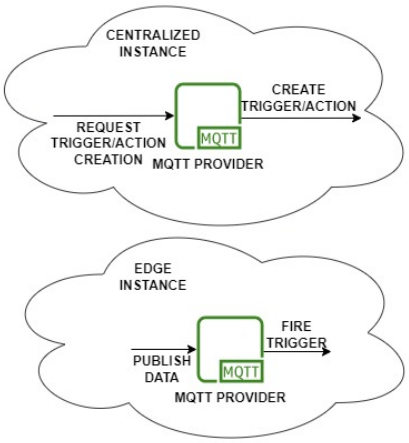

Figure 6.10: MQTT Provider

OpenWhisk API Host

The OpenWhisk API Host component is deployed in both the edge and central instances. It acts as the interface for managing API requests, such as adding triggers, actions, and invoking them. In this use case, it is specifically used when the MQTT broker fires a new trigger, triggering the corresponding actions.

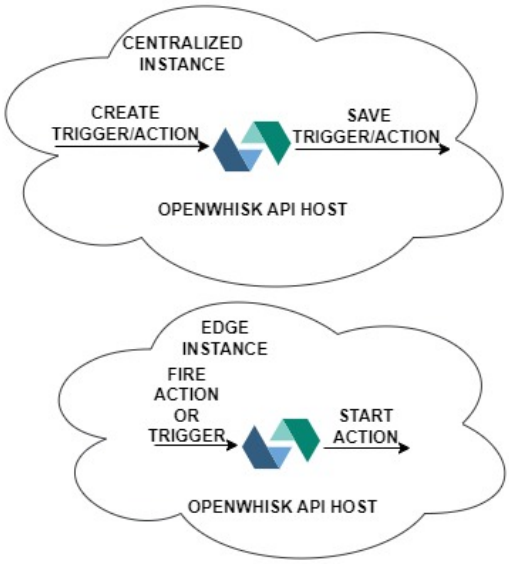

Figure 6.11: OpenWhisk API Host

OpenWhisk Alarm Provider

The OpenWhisk Alarm Provider component is utilized in both the edge and central instances. It is responsible for creating, updating, and deleting specific triggers based on predefined schedules or intervals. It ensures that OpenWhisk triggers are fired periodically, enabling timely data processing and analysis [36].

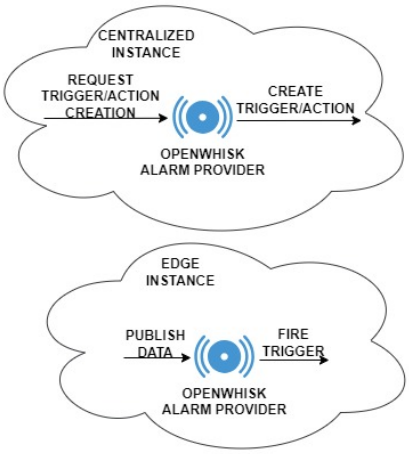

Figure 6.12: OpenWhisk Alarm Provider

OpenWhisk Cache

The OpenWhisk Cache component, found in the edge instances, is responsible for caching data. It helps improve the overall performance by storing frequently accessed data in memory, reducing the need for repeated data retrieval from the underlying storage systems. When invoking an action, for example, OpenWhisk first looks for the action's code and metadata in the cache. If nothing can be found, it searches for data in the centralized database.

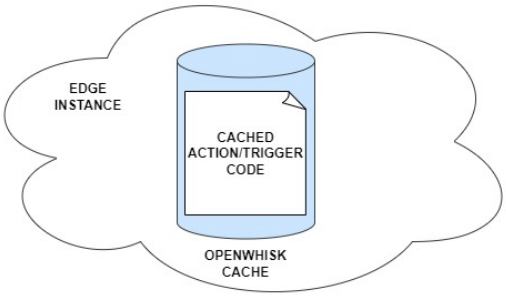

Figure 6.13: OpenWhisk Cache

InfluxDB

InfluxDB is a time series database designed to handle high volumes of time-stamped data efficiently. It provides a purpose-built storage solution for time-series data, making it ideal for storing and analyzing sensor data in real-time. InfluxDB offers powerful querying capabilities, retention policies, and downsampling techniques, enabling organizations to effectively manage and analyze large amounts of time-series data [50]. It serves as the storage system for the collected sensor data on the edge instances, while it stores aggregate data in the central instance. In this use case, it stores the data received from the sensors and processed by the OpenWhisk actions, allowing historical analysis and trend identification.

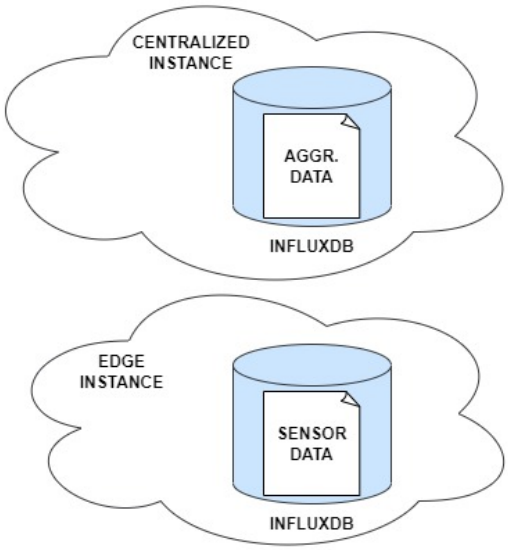

Figure 6.14: InfluxDB

CouchDB

CouchDB is a NoSQL document-oriented database that provides a flexible and distributed storage solution. It offers a schema-less data model, allowing for easy and dynamic data storage and retrieval. CouchDB supports replication, making it suitable for distributed and fault-tolerant architectures. It provides a RESTful API for interacting with the database, allowing developers to perform CRUD operations on documents and utilize powerful querying capabilities [51]. It is used in the central instance to store OpenWhisk data such as actions, triggers, and other relevant information.

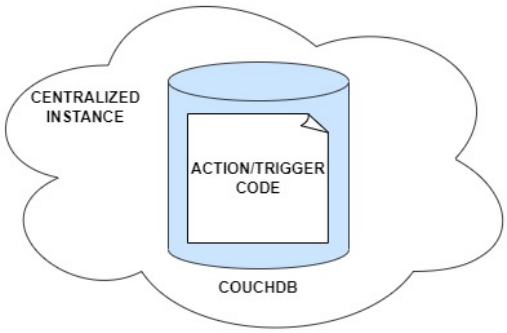

Figure 6.15: CouchDB

Grafana

Grafana is an open-source analytics and visualization platform that allows users to create and share customizable dashboards. It supports a wide range of data sources, including InfluxDB, to retrieve and visualize data. Grafana provides a rich set of visualization options, including graphs, charts, gauges, and alerting mechanisms, enabling users to monitor and analyze data effectively. It offers a user-friendly interface for building interactive ìdashboards and exploring data in real-time [48]. It is located in the centralized instance. It plays a crucial role in visualizing the aggregates derived from the collected sensor data.

Figure 6.16: Grafana

6.2 Implementation Details

In the implementation details of the use case, two key aspects are focused on: the main **action mechanism** and the **interaction between components**.

6.2.1 Main Action Mechanism

During the implementation phase, several choices were made to determine the most suitable actions for the main action mechanism. These choices were driven by the objectives of the thesis, ultimately leading to the implementation of the `addReadingToDb` and the actions for computing aggregates.

The `addReadingToDb` action was implemented as the primary action responsible for capturing and storing sensor data from the MQTT broker. This choice was motivated by the need to achieve real-time data storage and processing. By implementing this action, the system should be able to capture the sensor data as soon as it is received. As already explained in the previous section about the use case architecture 6.1, the use of OpenWhisk serverless functions for the implementation should ease the scalable execution of this action, accommodating varying workloads and ensuring high availability.

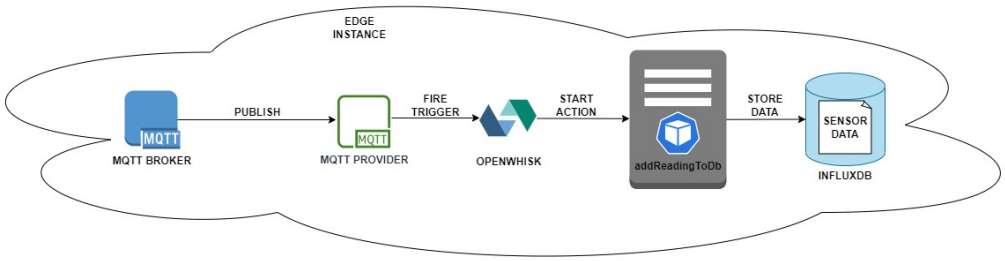

Figure 6.17: addReadingToDb Action Mechanism

The implementation of the actions for aggregates was based on the requirement to derive meaningful insights and analytics from the stored sensor data. These actions are triggered periodically using the OpenWhisk Alarm Provider, allowing for regular computation of aggregates based on the accumulated sensor readings. By implementing these actions, valuable information regarding the data center's performance, trends, and anomalies can be extracted. The implementation of the actions to compute aggregates facilitates efficient data analysis and supports decision-making processes. As already explained in the previous section about the use case architecture 6.1, by leveraging OpenWhisk serverless functions, the implementation of these actions can be executed in a distributed and parallel manner across multiple edge instances, and this aspect should enable scalability and efficient resource utilization.

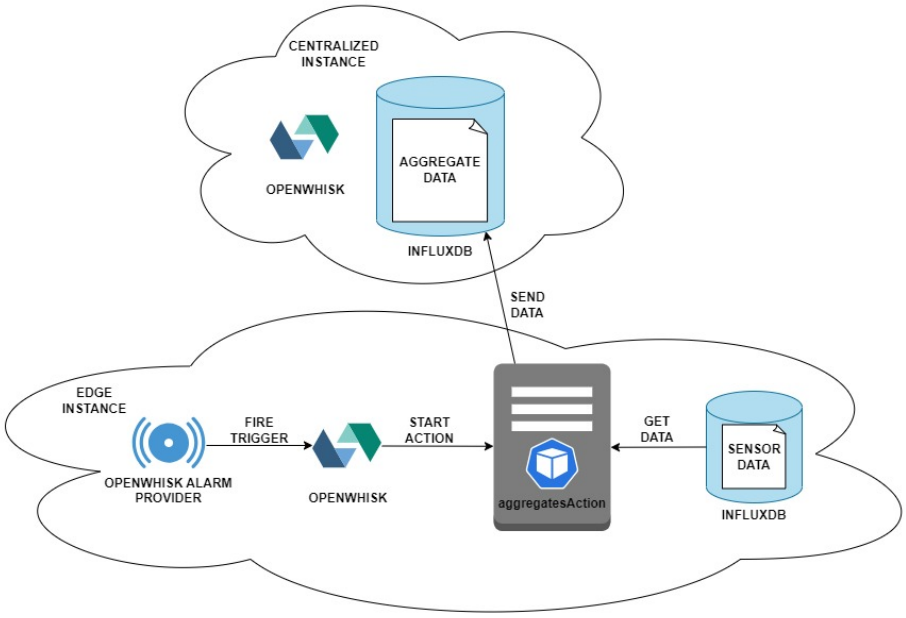

Figure 6.18: Aggregate Action Mechanism

These choices align with the objectives of the architecture and the desired outcomes:

- **Real-time Data Processing**: The implementation of the `addReadingToDb` action has been done to enable real-time storage of sensor data, trying to ensure immediate availability for monitoring. By implementing this action to capture the data as soon as it is received, the system should promptly store them in the local database. As already observerd, all these aspects have already been presented in the section about the use case architecture 6.1.

- **Scalability**: The use of OpenWhisk serverless functions for both the actions for aggregates and for the `addReadingToDb` action should be able to allow scalable execution. As the workload increases, additional instances of the actions can be automatically provisioned to handle the demand. This choice has been done to try ensuring that the system can efficiently accommodate varying data volumes without compromising performance or availability. As already observerd, all these aspects have already been presented in the section about the use case architecture 6.1.

- **Centralized Management**: The implementation phase also considered the aspect of centralized management. By utilizing OpenWhisk actions and triggers stored in a single CouchDB instance, the system achieves centralized management of code. This choice has been done to enable efficient handling

of code across multiple instances. Updates or modifications to the code can be easily applied universally, eliminating the need for individual code changes on each instance. This centralized management approach simplifies implementation, improves maintenance, and provides scalability and flexibility to the system. As already observerd, all these aspects have already been presented in the section about the use case architecture 6.1.

In summary, the choices made during the implementation phase led to the implementation of the `addReadingToDb` and of the actions to compute aggregates as the main action mechanism. These choices were motivated by the requirements for real-time data storage, scalability, and centralized managment. The implementation of the `addReadingToDb` action has been done to try ensuring prompt storage of sensor data, while the implementation of the actions for aggregates enables computation of meaningful aggregates for analysis and decision-making. Leveraging OpenWhisk serverless functions should further enhance the scalability and efficiency of the chosen actions, aligning with the objectives of the architecture.

6.2.2 Interaction Between Components

Interaction for addReadingToDb action

When sensor data is published to the local MQTT broker within each data center, a MQTT provider listens for these data publications. Once data is published, the MQTT provider triggers an interaction with the local OpenWhisk instance.

The trigger, configured using the MQTT package, is responsible for firing an OpenWhisk action called `addReadingToDb`. This action is implemented as a JavaScript function and is deployed as a kubernetes pod. It is responsible for writing the published sensor data to the local InfluxDB database.

Upon receiving the trigger, the local OpenWhisk instance checks if the trigger and its parameters are present in the local cache. The local cache, automatically created during OpenWhisk deployment, stores frequently used triggers and their parameters, improving performance by reducing the need for accessing the centralized CouchDB.

If the trigger and its parameters are found in the local cache, the local OpenWhisk instance retrieves them directly, avoiding the need for accessing the centralized CouchDB. However, if the trigger or its parameters are not present in the local cache, the local OpenWhisk instance queries the centralized CouchDB to retrieve the required information. This ensures the availability of triggers and their associated action code and parameters, even if they haven't been accessed recently or have been evicted from the local cache. The same steps are followed to retrieve the

action code and metadata.

Once the local OpenWhisk instance has obtained the trigger, the action and its parameters, it determines the appropriate invoker, which is an available cluster node, to create a pod that runs the `addReadingToDb` action. The pod is responsible for executing the JavaScript code of the action, which includes writing the sensor data to the local InfluxDB database. After the action completes its execution, the pod is terminated.

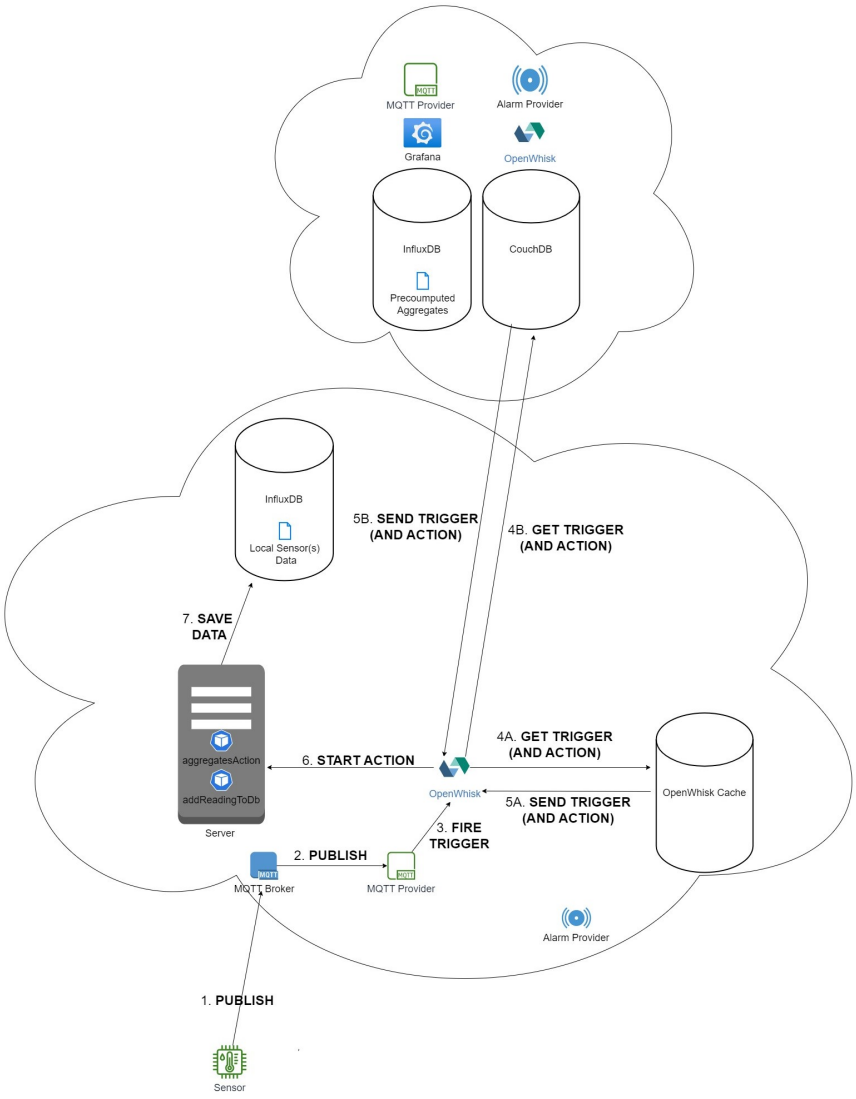

Figure 6.19: Interaction Between Components for addReadingToDb action

Interaction for the Actions Related to Aggregates

Computing aggregates from collected data in local data centers involves scheduled actions triggered by OpenWhisk alarms. The local OpenWhisk instance optimizes performance by employing a caching strategy for essential components like triggers, action code, and parameters. When the scheduled time arrives, the alarm trigger fires, and the local instance swiftly checks its cache for the required components. If found, it efficiently retrieves them, avoiding additional queries.

If the trigger or its parameters are not present in the local cache, the local OpenWhisk instance proceeds to query the centralized CouchDB, a reliable repository for storing trigger and action-related information. By querying the CouchDB, the necessary components, including the trigger, associated action code, and parameters, are obtained, ensuring all the required data is available for the computation process.

Once the trigger and action components are obtained, the local OpenWhisk instance determines the suitable invoker based on factors such as resource availability, load balancing, and optimization. The chosen invoker then creates a dedicated pod specifically designed for executing the actions related to aggregate computation.

Within the created pod, the JavaScript code associated with the actions is executed. The pod establishes connections with the local InfluxDB instances, which serve as the data sources for retrieving relevant information required for aggregate computation. Leveraging the connectivity to the local InfluxDB instances, the pod efficiently retrieves the necessary data and performs the required computations to generate the aggregates.

Once the computation process is complete, the actions store the computed aggregates in the centralized InfluxDB database. This centralized storage, located in the centralized cluster, ensures consistent and centralized storage of the aggregates. It facilitates further analysis, exploration, and data-driven decision-making based on the processed data.

In summary, the periodic triggering of aggregate computation actions, along with the caching mechanism and utilization of the OpenWhisk alarm package, enables efficient and timely computation of aggregates from collected data. By leveraging the connectivity to the centralized CouchDB and local InfluxDB instances, the system efficiently retrieves and processes the required information. This comprehensive workflow ensures reliable computation and centralized storage of aggregates, empowering organizations to extract valuable insights and make data-informed decisions that drive their success.

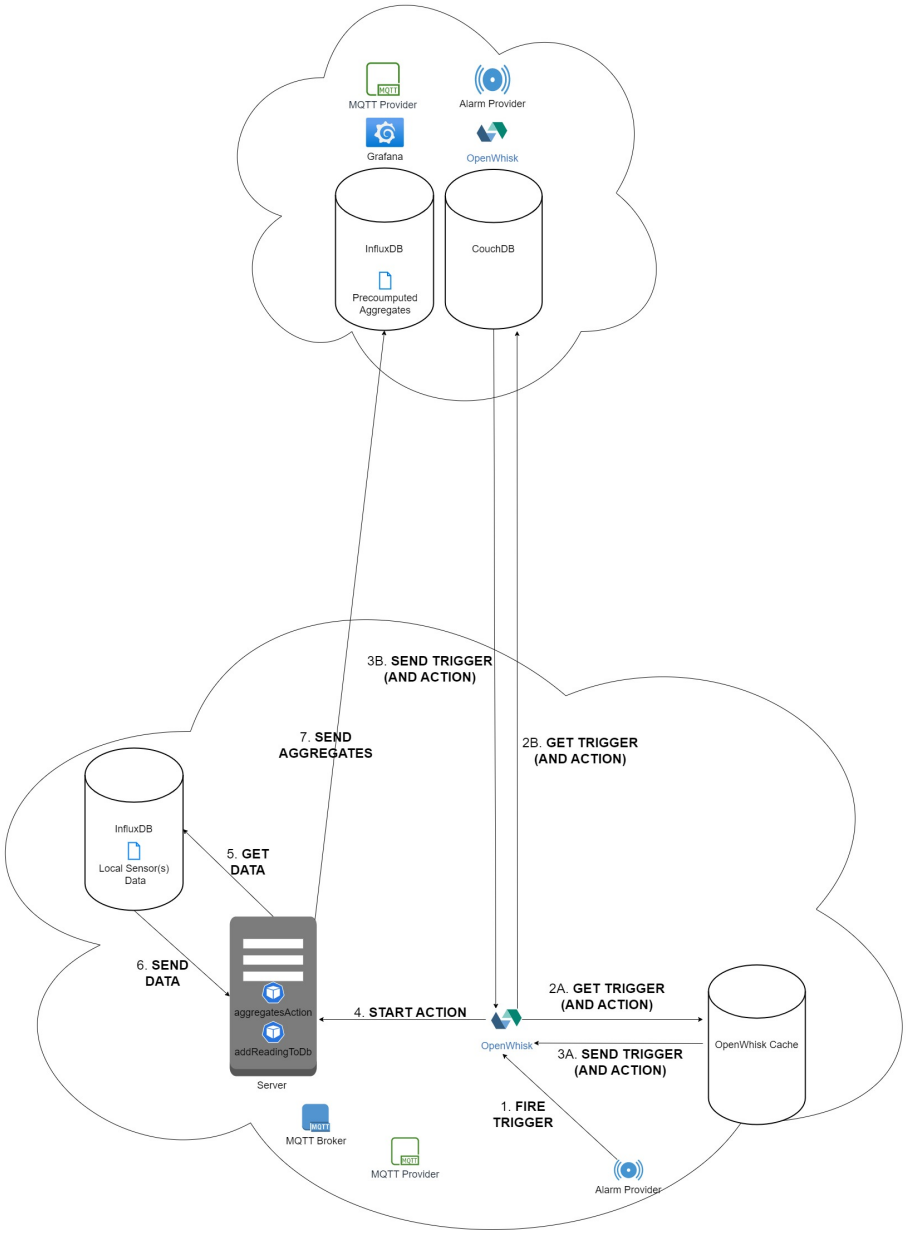

Figure 6.20: Interaction Between Components for the Actions Related to Aggregates

Chapter 7

Experimental Deployment

This chapter focuses on the practical implementation of the system, shedding light on the technologies, tools, and processes involved in deploying the edge and centralized instances. By covering topics such as deployment environment, sensor integration, deployment process, and deployed charts, it is possible to gain a comprehensive understanding of the deployment process and its outcomes.

7.1 Technologies and Tools

This section presents the different technologies and tools used for the experimental deployment phase of the chosen architecture.

7.1.1 Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a robust framework for running distributed systems and managing containerized workloads across a cluster of machines. Kubernetes offers features such as automatic scaling, load balancing, self-healing, and rolling updates, making it an ideal choice for deploying and managing microservices architectures [52].

In this thesis, Kubernetes has been utilized as the underlying infrastructure for deploying and managing the central and edge instances of the system. Its powerful capabilities in container orchestration and management have enabled efficient scaling, fault tolerance, and resource allocation for the application.

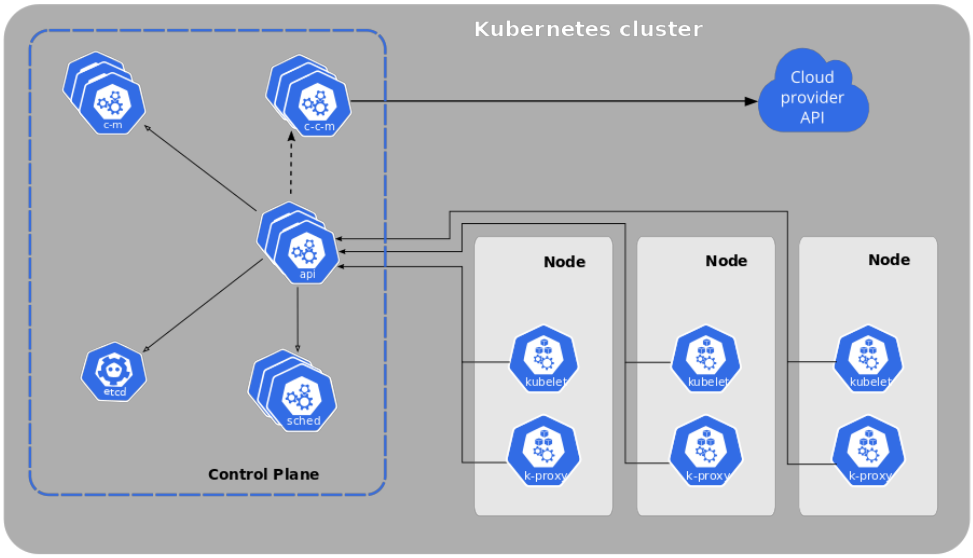

Figure 7.1: Components of Kubernetes

7.1.2 Docker

Docker is an open-source platform that automates the deployment and management of applications using containerization. It allows applications to be packaged into containers, which encapsulate all the dependencies and configurations needed for the application to run consistently across different environments. Docker provides a lightweight, portable, and isolated runtime environment for applications, ensuring reproducibility and simplifying deployment processes [53].

Docker has played a crucial role in this thesis by containerizing the application components, making them independent of the underlying infrastructure. This containerization approach has facilitated easy deployment, version control, and scalability of the system.

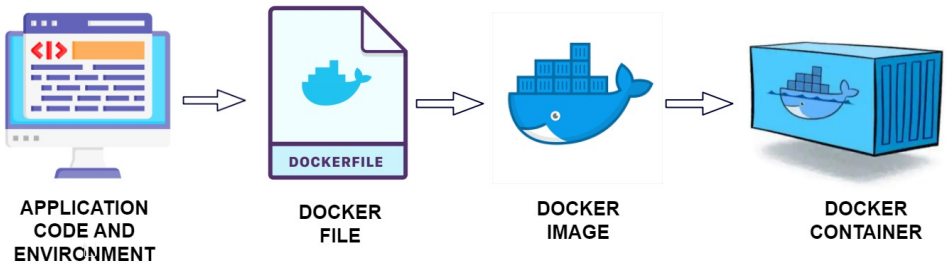

Figure 7.2: Containerization Process

7.1.3 Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications by providing a templating and release management system. It allows the definition of application configurations as charts, which are collections of pre-configured Kubernetes manifest files. Helm charts enable the easy installation, upgrade, and rollback of applications, making it a valuable tool for managing complex deployments [54].

In this thesis, Helm has been used to define and deploy the central and edge instances of the system. The charts provided a standardized and reproducible way to configure and deploy the application, simplifying the deployment process and ensuring consistency across different environments.

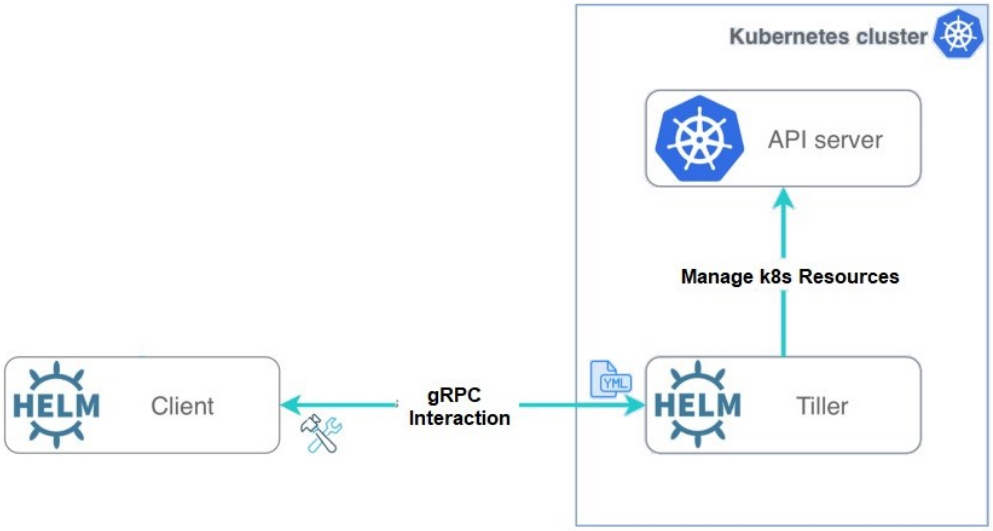

Figure 7.3: Helm Architecture

7.1.4 ChartMuseum

ChartMuseum is an open-source Helm chart repository that allows the hosting and distribution of Helm charts. It provides an easy way to store, share, and version control charts, enabling teams to collaborate and deploy applications consistently. ChartMuseum offers features such as chart indexing, authentication, and secure chart serving over HTTPS [55].

ChartMuseum has been utilized in this thesis as the repository for storing and distributing the Helm charts used in the deployment of the central and edge instances. It provided a centralized and reliable location for managing the charts, allowing seamless integration with the Helm deployment process.

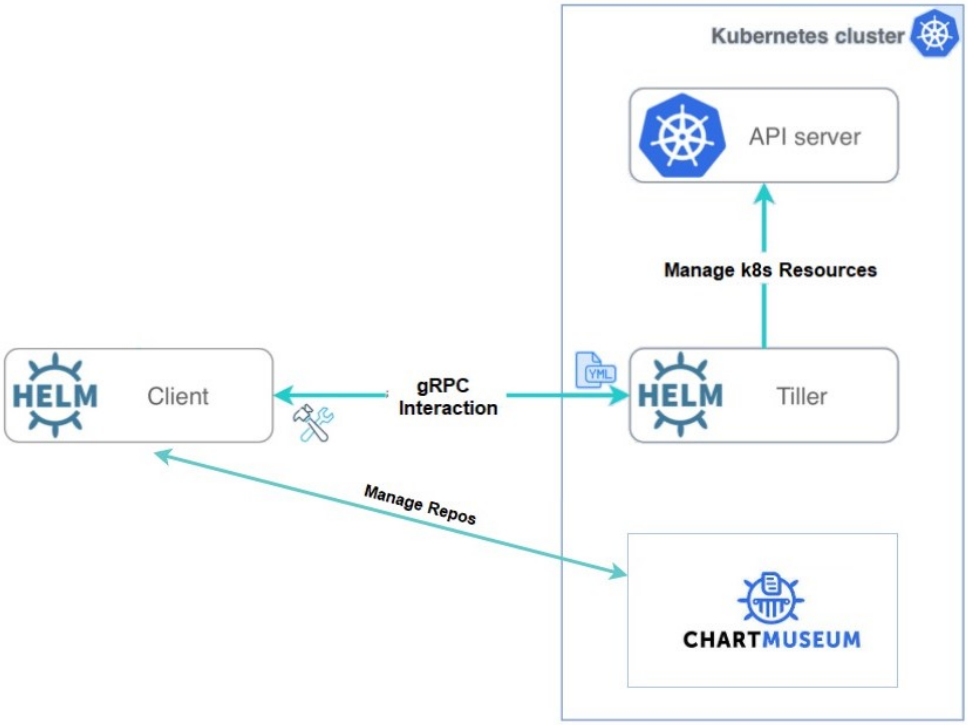

Figure 7.4: Chartmuseum and Its Interaction With Helm

7.1.5 Traefik

Traefik is a modern, dynamic, and cloud-native reverse proxy and load balancer designed for microservices architectures. It acts as an entry point for incoming traffic to the application and dynamically routes requests to the appropriate services based on configurable rules. Traefik supports automatic service discovery, SSL/TLS termination, circuit breakers, and various load balancing algorithms [56].

In this thesis, Traefik has been utilized as the Ingress Controller for the Kubernetes cluster, routing external traffic to the central and edge instances of the system. Its dynamic configuration capabilities and seamless integration with Kubernetes have facilitated efficient traffic management and load balancing.

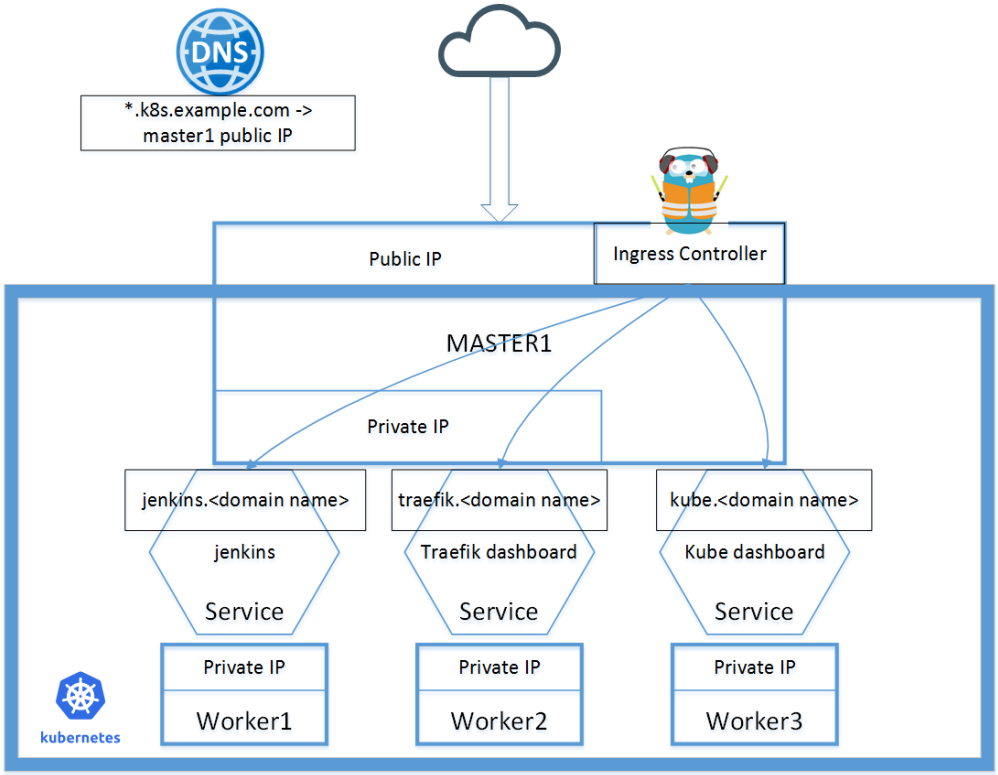

Figure 7.5: Details About How Traefik Ingress Controller Works

7.1.6 Arduino IDE

The Arduino IDE is the software component that provides a user-friendly programming environment for writing and uploading code to the Arduino boards. It is based on the C/C++ programming language and simplifies the development process by abstracting low-level details. The IDE offers a rich library ecosystem, containing pre-built functions and examples, making it easier to interface with sensors, perform data acquisition, and control devices [57].

Arduino IDE was utilized in this thesis to enable sensor integration and data transmission. Arduino boards, such as the ESP8266 NodeMCU, were employed to collect telemetry data from sensors and establish communication with remote

servers or MQTT brokers.

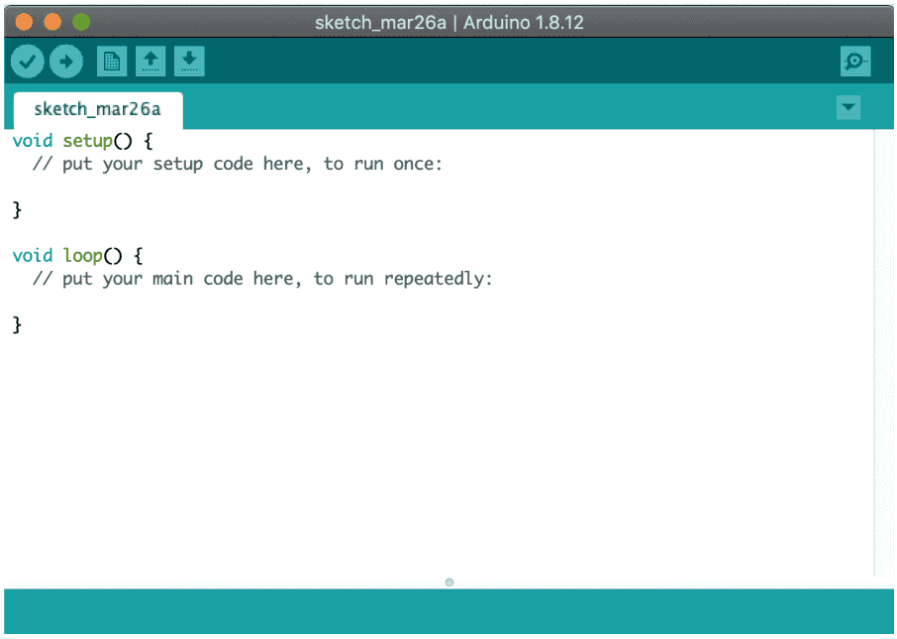

Figure 7.6: Arduino IDE

7.2 Deployment Environment and Sensors Integration

7.2.1 Deployment Environment

The deployment environment for this thesis consists of a single-node cluster with limited resources. The cluster is based on k3s, a lightweight Kubernetes distribution [58], and is specifically designed for resource-constrained environments. The single-node cluster is equipped with 6 CPU cores and 16 GB of RAM.

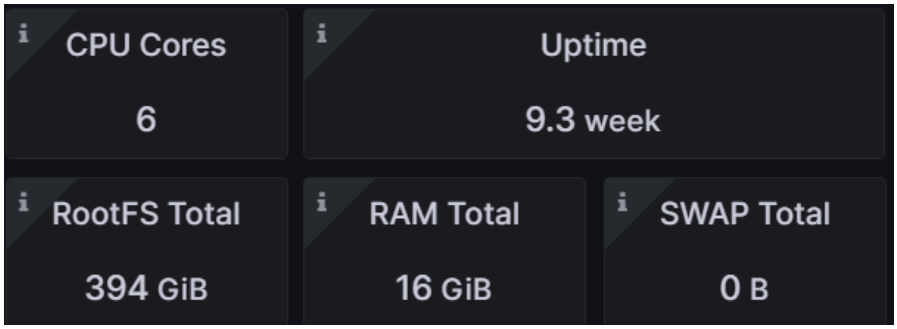

Figure 7.7: Single-Node Cluster Details

7.2.2 ESP8266 Sensor Integration

In addition to the deployment environment, the thesis incorporates the integration of a ESP8266 sensor. The ESP8266 is a widely used microcontroller with built-in Wi-Fi capabilities, making it suitable for IoT applications. These sensor can collect data from the physical environment and transmit it wirelessly to the edge instances [59].

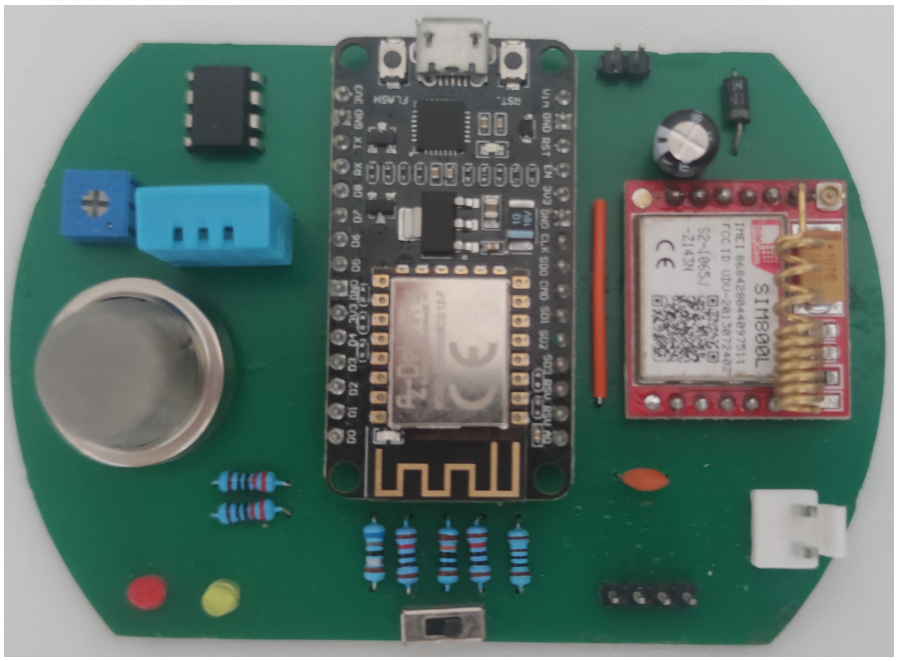

Figure 7.8: Use Case Sensor

7.3 Deployment Process

7.3.1 Edge Instance Deployment

Deployment Script

The OpenWhisk edge deployment involves the configuration and deployment of various components and services to create a robust environment. A central component of this deployment is the `deploy.sh` script, which automates the entire deployment process, ensuring a streamlined and efficient setup. By executing the `deploy.sh` script, users can deploy the entire edge solution seamlessly.

The `deploy.sh` script plays a vital role in the deployment process by performing several key functions. Firstly, it handles the parameter configuration, allowing users to customize the deployment according to their specific requirements. This ensures that the edge solution is tailored to meet the unique needs of the environment in which it will operate.

Furthermore, the script handles the setup of the repository, specifically the chartmuseum repository for helm charts. This repository provides access to the required helm charts for deploying the helm chart for the edge solution. By adding the chartmuseum repository, the `deploy.sh` script ensures that the necessary resources are available for the deployment process.

Finally, the `deploy.sh` script deploys the helm chart that comprises the edge solution. The helm chart is a collection of multiple subcharts, each serving a specific purpose in enabling edge computing capabilities. These subcharts include OpenWhisk, InfluxDB, Mosquitto, and an MQTT provider. Each subchart contributes to the overall functionality of the edge solution, such as serverless computing, data storage, and MQTT communication.

To ensure a successful deployment, proper configuration using the `values.yaml` file is essential, which is used during the deployment of the helm chart. This file contains crucial parameters that define the behavior and settings of the deployed services. By carefully setting these parameters in the `values.yaml` file, users can tailor the edge solution to their specific needs, such as specifying authentication credentials, network configurations, and integration details. The script takes also care of configuring in the right way those parameters.

In summary, the OpenWhisk edge deployment involves the configuration and deployment of various components and services using the `deploy.sh` script. This

script automates the process and performs key functions such as parameter configuration, repository setup, and helm chart deployment.

In order to access and utilize the `deploy.sh` script, it can be found in the `/openwhisk/deploy/edge` folder of the GitHub repository for this thesis[1]. The `values.yaml` file is also located in the same folder, providing the parameters modified by the script for the helm deployment. Additionally, the `parameters.yaml` file, containing specific configuration parameters which are used to set the `values.yaml` file, can be found in the `/openwhisk/deploy/parameters` folder[2].

Sensors Setup

The sensor setup for the edge instance involves the utilization of an Arduino project. This project employs an ESP8266 NodeMCU board and various sensors, including a DHT11 temperature and humidity sensor and an analog pin for gas concentration detection. The project enables the collection of telemetry data from these sensors and the transmission of the data to an MQTT broker.

To successfully set up the `send-telemetry.ino` project, specific parameters need to be configured. These parameters include the Wi-Fi network credentials (SSID and password) and the MQTT broker details (address, topic, username, password, and port). Administrators responsible for the data center must ensure the correct configuration of these parameters to establish network connectivity and MQTT communication.

This is the code in the file `send-telemetry.ino`:

```
// Pin configuration for ESP8266 NodeMCU connected to the output (OUT
    ) pin of the DHT11 sensor
#define tipoDHT DHT11
/* Defines the type of DHT sensor: in this case, DHT11 is selected,
   but there are other sensors such as DHT21 and DHT22
*/
DHT dht(D1, tipoDHT); // Instantiate the dht object of the DHT class
float h;              // Humidity
float t;              // Temperature
float g;              // Gas
```

[1] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/edge`

[2] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/parameters`

```
int o = 1;
int l = 0;

//*** Wi-Fi Configuration: Network credentials ***
const char* ssid = "ssid";                  // Insert the SSID here
const char* password = "password";  // Insert the Wi-Fi password here
//WiFiClient  client;

unsigned long time_last = 0;  // Last measurement time
unsigned long time_current;   // Current time
const long interval = 10000;  // Time interval between measurements (
    e.g., 15 minutes)

// MQTT Broker: TO BE CONFIGURED WITH YOUR BROKER VARIABLES
const char* mqtt_broker = "38.242.158.232";
const char* topic = "test";
const char* mqtt_username = "test";
const char* mqtt_password = "test";
const int mqtt_port = 31345;

WiFiClient espClient;
PubSubClient client(espClient);

// Device fingerprint and string with the ID of the datacenter
uint32_t ucid = ESP.getChipId();
uint32_t dataCenterID;

void setup() {
    delay(1000);

    pinMode(D3, OUTPUT);
    pinMode(D4, OUTPUT);
    pinMode(A0, INPUT);
    dht.begin();

    // Open a serial connection for debugging purposes
    Serial.begin(115200);

    // Get device fingerprint
    Serial.print("UCID: ");
    Serial.println(ucid, HEX);

    // Initialize the EEPROM library
    EEPROM.begin(sizeof(dataCenterID));
    EEPROM.get(0, dataCenterID);
    Serial.print("DC ID from EPROM: ");
    Serial.println(dataCenterID, DEC);

    WiFi.begin(ssid, password); // Connect to the Wi-Fi router
```

```
58      Serial.println("Connecting to Wi-Fi...");
59      while (WiFi.status() != WL_CONNECTED) {
60          delay(500);
61          Serial.print(".");
62      }
63      Serial.println("");
64      Serial.print("ESP8266 NodeMCU connected to Wi-Fi network: ");
65      Serial.print(ssid);
66      Serial.print(" with IP address: ");
67      Serial.println(WiFi.localIP());  // ESP8266's IP assigned by DHCP
68      Serial.println();
69      digitalWrite(D3, 1);  // Turn on LED 1 to indicate that Wi-Fi is
        active
70      delay(500);
71      rileva_invia();  // Call the function to read and send humidity
        and temperature data
72  }
73
74  void loop() {
75      time_current = millis();
76      if (time_current - time_last >= interval) {
77          time_last = time_current;
78          rileva_invia();
79          digitalWrite(D4, 1);
80          delay(2 * 60 * 1000);
81          l += 2;
82          if (l == 120) {
83              l = 0;
84              o = o * 2;
85          }
86          digitalWrite(D4, 0);
87      }
88  }
89
90  void rileva_invia() {
91      for (int i = 0; i < 3; i++) {
92          h = dht.readHumidity();      // Read humidity
93          t = dht.readTemperature();  // Read temperature in Celsius
94          g = (analogRead(A0) * 100) / 1024;
95          delay(300);
96      }
97      if (isnan(h) || isnan(t)) {
98          Serial.println("Reading error...");
99          return;
100         h = 0.0;
101         t = 0.0;
102         g = 0.0;
103     }
```

```
104     String s = "Humidity= " + String(h) + "      Temperature=" +
    String(t);
105     Serial.println(s);
106     String s2 = String(h) + "#" + String(t) + "#";
107     client.setServer(mqtt_broker, mqtt_port);
108     client.setKeepAlive(6000);
109     while (!client.connected()) {
110         String client_id = "esp8266-client-";
111         client_id += String(WiFi.macAddress());
112         Serial.printf("The client %s connects to the public MQTT
    broker\n", client_id.c_str());
113         if (client.connect(client_id.c_str(), mqtt_username,
    mqtt_password)) {
114         } else {
115             Serial.print("Failed with state ");
116             Serial.print(client.state());
117             delay(2000);
118         }
119     }
120
121     // Convert sensor data to JSON format
122     StaticJsonDocument<200> doc;
123     doc["humidity"] = h;
124     doc["temperature"] = t;
125     doc["gas_perc"] = g;
126     doc["device_fingerprint"] = ucid;
127     doc["datacenter_id"] = dataCenterID;
128     String json_data;
129     serializeJson(doc, json_data);
130
131     // Publish sensor data to MQTT topic
132     client.publish(topic, json_data.c_str());
133
134     // Print sensor data to serial monitor
135     Serial.println(json_data);
136 }
```

Additionally, another project incorporates the storage of the data center ID in the EEPROM of the device. This ID is retrieved and sent along with the telemetry data to provide identification and context. By storing the data center ID in the EEPROM, the device can maintain this information even after power cycles or reboots, ensuring consistent identification throughout its operation.

This capability allows for effective data center identification within the edge computing infrastructure, facilitating the organization, analysis, and management of the collected data.

This is the code in the file `store-datacenterID-eeprom.ino`:

```
#include <ESP8266WiFi.h>
#include <ESP_EEPROM.h>


// data center unique ID
uint32_t dataCenterID = 1; // replace this with a unique identifier
    for the data center

void setup() {
  Serial.begin(115200);
  Serial.println();

  EEPROM.begin(sizeof(dataCenterID));

  EEPROM.put(0, dataCenterID);
  EEPROM.commit();

  EEPROM.get(0, dataCenterID);
  Serial.print("DC ID from EPROM: ");
  Serial.println(dataCenterID, DEC);

}

void loop() {
  // empty
}
```

In order to access and utilize the projects, they can be found in the folder `/openwhisk/deploy/edge/sensors` of the GitHub repository for this thesis[3].

7.3.2 Centralized Instance Deployment

Deployment Script

The OpenWhisk centralized deployment involves the configuration and deployment of various components and services to create a robust environment. A central component of this deployment is the `deploy.sh` script, which automates the entire deployment process, ensuring a streamlined and efficient setup. By executing the `deploy.sh` script, users can deploy the entire centralized solution seamlessly.

[3] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/edge/sensors`

The `deploy.sh` script plays a vital role in the deployment process by performing several key functions. Firstly, it handles the parameter configuration, allowing users to customize the deployment according to their specific requirements. This ensures that the centralized solution is tailored to meet the unique needs of the environment in which it will operate.

Furthermore, the script handles the setup of the repository, specifically the chartmuseum repository for helm charts. This repository provides access to the required helm charts for deploying the helm chart for the centralized solution. By adding the chartmuseum repository, the `deploy.sh` script ensures that the necessary resources are available for the deployment process.

Finally, the `deploy.sh` script deploys the helm chart that comprises the centralized solution. The helm chart is a collection of multiple subcharts. These subcharts include OpenWhisk, CouchDB, InfluxDB, Grafana Operator, Grafana, and MQTT provider. Each subchart contributes to the overall functionality of the centralized solution, such as serverless computing, data storage, dahboard capabilities, and MQTT communication.

To ensure a successful deployment, proper configuration using the `values.yaml` file is essential, which is used during the deployment of the helm chart. This file contains crucial parameters that define the behavior and settings of the deployed services. By carefully setting these parameters in the `values.yaml` file, users can tailor the edge solution to their specific needs, such as specifying authentication credentials, network configurations, and integration details. The script takes also care of configuring in the right way those parameters.

After deploying OpenWhisk, the script initializes the CouchDB database. This step ensures that the necessary tables required by the MQTT provider, deployed in edge instances, are available.

The script checks if the OpenWhisk actions, triggers, and rules have already been created. If they are not found, it creates them. These components are crucial for processing and handling incoming data from sensors. In the end, the script also creates an external user to access the Grafana Dashboard. This allows authorized users to monitor and visualize data through Grafana.

In summary, the OpenWhisk centralized deployment involves the configuration and deployment of various components and services using the `deploy.sh` script. This script automates the process and performs key functions such as parameter configuration, repository setup, helm chart deployment, database setup, creation

of actions and triggers and creation of Grafana users.

In order to access and utilize the `deploy.sh` script, it can be found in the `/openwhisk/deploy/central` folder of the GitHub repository for this thesis[4]. The `values.yaml` file is also located in the same folder, providing the parameters modified by the script for the helm deployment. Additionally, the `parameters.yaml` file, containing specific configuration parameters which are used to set the `values.yaml` file, can be found in the `/openwhisk/deploy/parameters` folder[5].

addReadingToDb Action

To store sensor data in the local InfluxDB instance, an `addReadingToDb` action is created using JavaScript code. This action receives messages published by the sensor through an MQTT broker and provider. The `deploy.sh` script facilitates the setup of the action and related components (trigger, feed, package) if necessary. By leveraging MQTT infrastructure and InfluxDB, the proposed solution ensures seamless data transfer and storage, enabling real-time data handling and subsequent analysis. The integration of the `addReadingToDb` action with MQTT and InfluxDB aligns with the goal of utilizing serverless computing and IoT technologies to capture and store sensor data effectively.

This is the code of the `addReadingToDb` action:

```
1  function main(args) {
2    var request = require('request');
3    // Retrieve the parameters to contact the local InfluxDB
4    const url= args.influx_url;
5    const token= args.influx_token;
6    const org= args.influx_org;
7    const bucket= args.influx_bucket;
8
9    try{
10     const data = JSON.parse(args.body); // Parse the string into a
       JSON object
11     const currentTime = Date.now() * 1000000;
12     const keys = Object.keys(data);
13
14     // Construct the InfluxDB line protocol string
```

[4] https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/central

[5] https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/parameters

```
15      let influxData = '';
16      for (let i = 0; i < keys.length; i++) {
17        if(keys[i]!="device_fingerprint" && keys[i]!="datacenter_id")
18          influxData += `${keys[i]},datacenter_id=${data["datacenter_id
        "]},device_fingerprint=${data["device_fingerprint"]} value=${data[
        keys[i]]} ${currentTime}\n`;
19      }
20
21      const options = {
22        url: url+'/api/v2/write?org='+org+'&bucket='+bucket+'&precision
        =ns',
23        method: 'POST',
24        headers: {
25          'Authorization': 'Token '+token,
26          'Content-Type': 'text/plain',
27          'Accept': 'application/json'
28        },
29        body: influxData // Use the constructed InfluxDB line protocol
        string as the request body
30      };
31
32      request(options, function(err, res, body) {
33        console.log(err);
34      });
35    }
36    catch(e){
37      console.log(e);
38    }
39  }
```

In order to access and utilize the code of the action, it can be found in the `/openwhisk/deploy/central/openwhisk/actions-and-triggers` folder of the GitHub repository for this thesis[6].

Actions for Aggregates

To process and store sensor data effectively, different aggregate actions are implemented for various types of data (humidity, gas percentage, temperature) and aggregates (minimum, maximum, and mean) over the last 30 minutes. These actions share a common purpose of connecting to the local and centralized InfluxDB, retrieving the desired data, computing the target aggregate, and sending the result to the centralized database. The `deploy.sh` script facilitates the setup of the action and related components (trigger, feed, package) if necessary. The workflow

[6] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/central/openwhisk/actions-and-triggers`

of the aggregate actions can be summarized as follows:

- **Connection Parameters**: Each aggregate action requires connection parameters for both the local and centralized InfluxDB instances. These parameters include the URL, token, organization, and bucket information.
- **Local InfluxDB Connection**: Establish a connection with the local InfluxDB using the provided parameters to retrieve data from the local database.
- **Data Retrieval**: Retrieve the relevant data from the last 30 minutes based on the specific data type (humidity, gas percentage, or temperature). Use appropriate queries or filters to fetch the required information.
- **Aggregate Computation**: Compute the target aggregate (minimum, maximum, or mean) for the retrieved data. Utilize built-in InfluxDB functions or custom calculations to determine the desired aggregate value.
- **Centralized InfluxDB Connection**: Connect to the centralized InfluxDB using the provided parameters to send the computed aggregate for storage and further analysis.
- **Aggregate Transmission**: Construct a write operation to send the computed aggregate to the centralized InfluxDB. Ensure the appropriate formatting and precision are maintained to accurately record the aggregate value.
- **Scheduled Execution**: Schedule the aggregate actions to run every 30 minutes using a scheduler or automation tool. This periodic execution ensures that the aggregates are computed and stored at regular intervals, keeping the data up-to-date.

By implementing separate aggregate actions for each data type and aggregate (min, max, and mean), the system can efficiently handle sensor data, facilitate real-time analysis, and store the computed aggregates for future reference. This is the code of one of the actions to compute aggregates, in particular of the `aggregatesGasAvgAction`:

```
function main(params) {
    const request = require('request');
    console.log(params);

    // Extract parameters
    const local_url= params.local_url;
    const local_token= params.local_token;
    const local_org= params.local_org;
    const local_bucket= params.local_bucket;
```

```
10
11      const central_url= params.central_url;
12      const central_token= params.central_token;
13      const central_org= params.central_org;
14      const central_bucket= params.central_bucket;
15
16
17      // Compute daily aggregates
18      const today = new Date();
19      const halfHourAgo = new Date(today.getTime() - (30 * 60 * 1000));
20
21
22
23     try{
24      const query = `
25      from(bucket: "measure")
26      |> range(start: ${halfHourAgo.toISOString()}, stop: ${today.
       toISOString()})
27      |> filter(fn: (r) => r["_measurement"] == "gas_perc")
28      |> filter(fn: (r) => r["_field"] == "value")
29      |> group(columns: ["datacenter_id", "_measurement"])
30      |> aggregateWindow(every: 1d, fn: mean, createEmpty: false)
31      |> yield(name: "mean")
32      `;
33
34    // Query InfluxDB instance
35    const queryURL = `${local_url}/api/v2/query?org=${local_org}&bucket
       =${local_bucket}&pretty=true`;
36    const requestOptions = {
37      url: queryURL,
38      method: 'POST',
39      headers: {
40        'Authorization': `Token ${local_token}`,
41        'Content-Type': 'application/csv'
42      },
43      json: {
44        query: query
45      }
46    };
47      request(requestOptions, function (error, response, body) {
48        if (!error && response.statusCode == 200) {
49          const output= body;
50          const rows = output.split("\n"); // split the string into an
       array of rows
51          var gas_perc_avg;
52          var location;
53          var j=0;
54
55          rows.forEach(row => {
```

```
56          const columns = row.split(","); // split the row into an
    array of columns
57          columns.shift();
58          if(j==0 && columns[0]=="mean"){
59            location=columns[5];
60            j++;
61          }
62          if(columns[0]=='mean'){
63            if(columns[4]=='gas_perc')
64              gas_perc_avg= columns[6];
65          }
66        });
67        var tday = today.getTime() * 1000000;
68        var year=today.getFullYear();
69        var month= today.getMonth()+1;
70        var day=today.getDate();
71        // Construct the InfluxDB line protocol string
72        var influxData=`gas_perc,datacenter_id=${location},year=${
    year},month=${month},day=${day} avg=${gas_perc_avg} ${tday}\n`;
73
74
75        const options = {
76          url: central_url+'/api/v2/write?org='+central_org+'&bucket=
    '+central_bucket+'&precision=ns',
77          method: 'POST',
78          headers: {
79            'Authorization': 'Token '+central_token,
80            'Content-Type': 'text/plain',
81            'Accept': 'application/json'
82          },
83          body: influxData // Use the constructed InfluxDB line
    protocol string as the request body
84        };
85
86        request(options, function(err, res, body) {
87          console.log(res)
88          console.log(err);
89          if (err) {
90            console.error(`Error sending daily aggregates to remote
    database: ${err}`);
91          } else {
92            console.log('Daily aggregates inserted into remote
    database');
93          }
94        });
95      }else {
96        console.error(`Error querying local InfluxDB instance: ${
    error}`);
97      }
```

```
98        });
99    }
100   catch(e){
101       console.log(e);
102   }
103 }
```

In order to access and utilize the code of those actions, they can be found in the folder `/openwhisk/deploy/central/openwhisk/actions-and-triggers/aggregates` of the GitHub repository for this thesis[7].

Grafana Dashboard for Monitoring Aggregate Data

The deployment of the centralized solution includes the deployment of a Grafana instance, which serves as a powerful tool for visualizing and monitoring aggregate data. A Grafana Dashboard is created using the provided Grafana chart, allowing users to monitor various metrics related to the system. The dashboard features a template variable that enables the selection of the desired data center, providing a customizable and flexible monitoring experience.

[7] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/central/openwhisk/actions-and-triggers/aggregates`

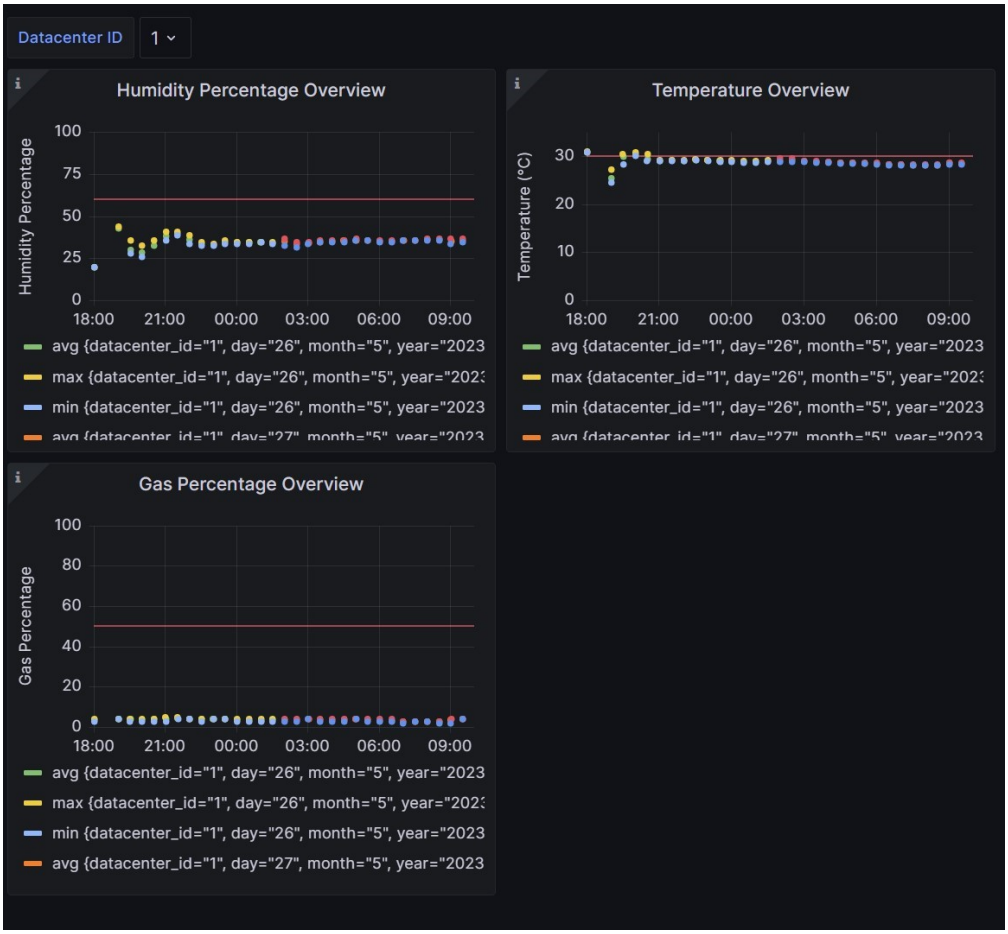

Figure 7.9: Grafana Dashboards for the Use Case

The Grafana Dashboard consists of three panels: Gas Percentage, Humidity Percentage, and Temperature. Each panel displays the average, maximum, and minimum values of their respective metrics. These values are computed every half an hour using an OpenWhisk action, ensuring that the dashboard reflects real-time data trends and patterns.

To access the Grafana Dashboard, users can visit the following address:

`https://grafana.<centralized-openwhisk-namespace>.<domain-name>`

The necessary credentials for accessing Grafana can be found in `parameters.yml` file under the `grafana.username` and `grafana.password` fields, which can be retrieved in the `/openwhisk/deploy/parameters` folder of the GitHub repository for

this thesis[8]. The admin username and password are required for full administrative access to Grafana.

In addition to the admin access, a default external user with read-only rights is also created during the deployment process with the `deploy.sh` script. The username and password for this external user are specified in the `parameters.yml` file under the `grafana.extUsername` and `grafana.extPassword` fields. This user profile is intended for use by the local data center administrators, allowing them to monitor the system's performance without making any modifications.

Adding new aggregate data to the Grafana Dashboard is a straightforward process, allowing for easy customization and expansion of the monitoring capabilities. The centralized admin has the flexibility to add new dashboards directly from the Grafana user interface or by creating a new `GrafanaDashboard` and updating the Helm chart.

The yaml file to create the current `GrafanaDashboard` can be found in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana/templates` of the GitHub repository for this thesis[9].

7.4 Deployed Charts

In the upcoming section, the deployment process of the OpenWhisk solution will be explored, with a focus on the Edge Chart and Centralized Chart. These charts play a crucial role in deploying the edge and centralized instances, respectively.

The section will delve into the significance of Helm charts, which serve as a packaging and deployment mechanism for the OpenWhisk solution. Both the Edge and Centralized solutions utilize a top Helm chart, which consists of various subcharts such as OpenWhisk, Mosquitto, MQTT provider, InfluxDB, CouchDB, Grafana, and Grafana Operator. Each subchart contributes to the overall functionality of the deployment.

Throughout the section, detailed discussions will be provided on the functionalities and interactions of these subcharts. This will give the necessary knowledge

[8] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/parameters`

[9] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana/templates`

to effectively configure and deploy the OpenWhisk solution using Helm charts.

By the end of the section, it will be possible to have a comprehensive understanding of the deployment process and be empowered to utilize Helm charts proficiently for deploying the OpenWhisk solution.

7.4.1 Centralized Chart

The Centralized Chart is specifically designed for deploying the centralized instance of the OpenWhisk solution. This chart plays a crucial role in setting up the necessary components to establish the centralized environment. It consists of various subcharts, each responsible for deploying a specific component of the solution. The subcharts included in the Centralized Chart are:

- CouchDB
- Grafana
- Grafana Operator
- InfluxDB
- OpenWhisk
- MQTT Provider

Each subchart contributes to the overall functionality of the centralized solution. To retrieve the Centralized Chart and its subcharts, the following commands can be used:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/centralized-chart
```

These commands will add the Chart Museum repository and download the Centralized Chart, which contains all the necessary subcharts for deploying the centralized instance of the OpenWhisk solution.

This chart can also be retrieved in the GitHub repository for this thesis[10], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/central-chart`.

[10] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/central-chart`

7.4.2 Edge Chart

The Edge Chart is specifically designed for deploying the edge instance of the OpenWhisk solution. This chart plays a crucial role in setting up the necessary components to establish the edge environment. It consists of various subcharts, each responsible for deploying a specific component of the solution. The subcharts included in the Edge Chart are:

- InfluxDB
- OpenWhisk
- MQTT Provider
- Mosquitto

Each subchart contributes to the overall functionality of the edge solution. To retrieve the Edge Chart and its subcharts, the following commands can be used:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/edge-chart
```

These commands will add the Chart Museum repository and download the Edge Chart, which contains all the necessary subcharts for deploying the edge instance of the OpenWhisk solution.

This chart can also be retrieved in the GitHub repository for this thesis[11], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/edge-chart`.

7.4.3 Subcharts

The Edge and Centralized Chart consist of several subcharts that are responsible for deploying different components of the edge and centralized solution. Each subchart plays a specific role in setting up the edge and centralized environment. In this section, all the subcharts included in the Edge and Centralized Chart will be presented in detail.

CouchDB

The CouchDB - ChartMuseum is an extended version of the CouchDB chart provided by Apache [60]. This chart incorporates additional functionalities and

[11] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/edge-chart`

modifications to enhance the capabilities of the CouchDB deployment.

An `IngressRoute` has been added to the CouchDB - ChartMuseum to enable connectivity between the edge instances and the CouchDB of the centralized instance. This `IngressRoute` allows external access to the CouchDB service from outside the Kubernetes cluster. The yaml file of the `IngressRoute` can be retrieved in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/couchdb/templates` of the GitHub repository for this thesis[12].

```
{{- if .Values.iroute.enabled }}
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: couchdb
  labels:
    app: couchdb
spec:
  routes:
  - kind: Rule
    match: Host(`{{ .Values.iroute.gateway }}`)
    services:
    - kind: Service
      name: {{ .Release.Name }}-couchdb
      namespace: {{ .Release.Namespace }}
      port: {{ .Values.service.externalPort}}
{{- end }}
```

Figure 7.10: IngressRoute for CouchDB

To retrieve the CouchDB - ChartMuseum, the following commands can be used:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/couchdb
```

Executing the above commands will add the ChartMuseum repository to the Helm configuration and retrieve the CouchDB chart from the repository. The downloaded chart can then be used to deploy the extended CouchDB instance within the Kubernetes environment.

[12] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/couchdb/templates`

This chart can also be retrieved in the GitHub repository for this thesis[13], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/couchdb`.

OpenWhisk

The OpenWhisk - ChartMuseum is an extension of the official OpenWhisk chart provided by Apache [61].

To address certain issues with the provided Helm package for the nodejs14 runtime in OpenWhisk, a new Docker image was created. The original Helm package had problems with the installation of the "requests" module using `npm`. In order to resolve this, a Docker image named `giuliabianchi1408/action-nodejs-v14` was created, and the runtimes file was modified to utilize this new image. The `Dockerfile` can also be retrieved in the GitHub repository for this thesis[14], as for the new configuration for runtimes[15].

```
1  FROM openwhisk/action-nodejs-v14:1.20.0
2
3  # Install request module
4  RUN npm install request
5
6  CMD ["node", "--expose-gc", "app.js"]
```

Figure 7.11: Dockerfile for the Updated Runtime

[13] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/couchdb`

[14] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/prototypes/fixing-nodejsaction/Dockerfile`

[15] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/openwhisk/runtimes.json`

Figure 7.12: Old Runtime vs New Runtime

In order to facilitate communication between various services deployed within the same namespace and enable external access when required, the network policies provided by OpenWhisk were extended. The old network policies can be retrieved in the GitHub repository for this thesis[16], as for the new network policies[17].

A bug was identified in the alarm provider component of OpenWhisk. Specifically, when using the default image provided by OpenWhisk, the URI of the API host is created using the internal API host name and port. This poses a security concern since the internal port (port 80) does not provide any security options, while the URI is configured for HTTPS. To address this issue, modifications were made to the alarm provider to utilize the external API host name and port, which are exposed on a secure port and support HTTPS. The old configuration for the alarm provider pod can be retrieved in the GitHub repository for this thesis[18], as for the new configuration[19].

[16] https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/original-charts/openwhisk-1.0.0/templates/network-policy.yaml

[17] https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/openwhisk/templates/network-policy.yaml

[18] https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/original-charts/openwhisk-1.0.0/templates/provider-alarm-pod.yaml

[19] https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/openwhisk/templates/provider-alarm-pod.yaml

Figure 7.13: Old Configuration of Alarm Provider Pod vs New Configuration

To retrieve the OpenWhisk - ChartMuseum, use the following command:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/openwhisk
```

By executing the above commands, the ChartMuseum repository will be added to the Helm configuration, and the OpenWhisk chart will be retrieved from the repository. The downloaded chart can then be used to deploy the extended OpenWhisk solution within the Kubernetes environment.

This chart can also be retrieved in the GitHub repository for this thesis[20], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/openwhisk`.

Mosquitto

The Mosquitto - ChartMuseum is an extension of another Mosquitto chart [62].

The modification made to the Mosquitto chart involves the addition of specific labels to enable seamless integration with OpenWhisk Network Policies. These labels ensure that the Mosquitto application functions correctly within the OpenWhisk environment, allowing smooth communication between different components.

To retrieve the Mosquitto - ChartMuseum, use the following command:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/mosquitto
```

[20] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/openwhisk`

By executing the above commands, the ChartMuseum repository will be added to the Helm configuration, and the Mosquitto chart will be retrieved from the repository. The downloaded chart can then be used to deploy the extended Mosquitto solution within the Kubernetes environment.

This chart can also be retrieved in the GitHub repository for this thesis[21], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/mosquitto`.

InfluxDB

The InfluxDB - ChartMuseum is an extension of the original InfluxDB chart [63].

An `IngressRoute` has been added to the InfluxDB - ChartMuseum to facilitate the connection between the edge instances and the centralized InfluxDB. The `IngressRoute` allows the edge instances to write aggregates and send data to the central InfluxDB for storage and analysis. The yaml file of the `IngressRoute` can be retrieved in the GitHub repository for this thesis[22].

To retrieve the InfluxDB - ChartMuseum, use the following command:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/influxdb
```

By executing the above commands, the ChartMuseum repository will be added to the Helm configuration, and the InfluxDB chart will be retrieved from the repository. The downloaded chart can then be used to deploy the extended InfluxDB solution within the Kubernetes environment.

[21] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/mosquitto`

[22] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/influxdb2/templates/ingress-influxdb-centralized.yaml`

```
{{- if .Values.iroute.enabled }}
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: influxdb
  labels:
    app: influxdb
spec:
  routes:
  - kind: Rule
    match: Host(`{{ .Values.iroute.externalUrl }}`)
    services:
    - kind: Service
      name: {{ .Release.Name }}-influxdb2
      namespace: {{ .Release.Namespace }}
      port: {{ .Values.service.port }}
{{- end }}
```

Figure 7.14: IngressRoute for InfluxDB

This chart can also be retrieved in the GitHub repository for this thesis[23], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/influxdb2`.

MQTT Provider

The MQTT Provider - ChartMuseum is a chart that deploys an MQTT provider for OpenWhisk. It is based on a solution provided at this bibliography reference [35].

This chart enhances the functionality of OpenWhisk by incorporating an MQTT provider, allowing seamless integration with MQTT-based applications and devices.

[23] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/influxdb2`

By deploying the MQTT Provider chart, it is possible to leverage the power of OpenWhisk to process and react to MQTT messages, enabling event-driven applications and workflows within your edge solution.

To retrieve the MQTT Provider - ChartMuseum, use the following command:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/mqtt-provider
```

By executing the above commands, the ChartMuseum repository will be added to the Helm configuration, and the MQTT Provider chart will be retrieved from the repository. The downloaded chart can then be used to deploy the MQTT Provider solution within the Kubernetes environment.

This chart can also be retrieved in the GitHub repository for this thesis[24], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/mqtt-provider`.

Grafana

The Grafana - ChartMuseum is a chart that deploys Grafana.

The chart includes a `GrafanaDataSource` that connects to an InfluxDB database. This data source allows Grafana to retrieve data from InfluxDB for visualization on the dashboards.

Another component of the chart is the `GrafanaDashboard`. The dashboard is designed to display specific data retrieved from the connected InfluxDB database. It provides visual representations and insights based on the collected data.

The yaml file of the `GrafanaDataSource` can be retrieved in the GitHub repository for this thesis[25], as for the `GrafanaDashboard`[26].

[24] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/mqtt-provider`

[25] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana/templates/05-grafanadatasource.yml`

[26] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana/templates/06-grafanadashboard.yml`

```
{{- if .Values.influx.enabled }}
apiVersion: integreatly.org/v1alpha1
kind: GrafanaDataSource
metadata:
  name: influx
  labels: {{ include "grafana.app.labels" . | nindent 4 }}
    app.kubernetes.io/component: influx
    name: grafana
    user-action-pod: "true"
spec:
  name: influx
  datasources:
  - access: "proxy"
    isDefault: true
    name: "influx"
    type: "influxdb"
    url: {{ .Values.influx.url }}
    jsonData:
      version: Flux
      organization: {{ .Values.influx.organization }}
      defaultBucket: {{ .Values.influx.bucket }}
      tlsSkipVerify: true
    secureJsonData:
      token: {{ .Values.influx.token }}
{{- end }}
```

Figure 7.15: GrafanaDataSource for InfluxDB

To retrieve the Grafana - ChartMuseum, use the following command:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/grafana
```

By executing the above commands, the ChartMuseum repository will be added to the Helm configuration, and the Grafana chart will be retrieved from the repository. The downloaded chart can then be used to deploy the Grafana solution within the

Kubernetes environment.

This chart can also be retrieved in the GitHub repository for this thesis[27], in the folder `/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana`.

Grafana Operator

The Grafana Operator - ChartMuseum is a chart that deploys Grafana Operator.

The Grafana Operator [64] extends the functionality of Grafana by providing custom resource definitions (CRDs) [65] and controllers that enable declarative management of Grafana instances. It simplifies the process of provisioning and configuring Grafana instances, including the setup of data sources, dashboards, and users.

To retrieve the Grafana Operator - ChartMuseum, use the following command:

```
$ helm repo add chartmuseum https://chart.liquidfaas.cloud
$ helm pull chartmuseum/grafana-operator
```

By executing the above commands, the ChartMuseum repository will be added to the Helm configuration, and the Grafana Operator chart will be retrieved from the repository. The downloaded chart can then be used to deploy the Grafana Operator solution within the Kubernetes environment.

This chart can also be retrieved in the GitHub repository for this thesis[28], in the path `/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana-operator`.

[27] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana`

[28] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/deploy/helm-charts/chartmuseum-charts/grafana-operator`

Chapter 8

Demonstration of Feasibility

This thesis presents a comprehensive solution for monitoring multiple data centers in a Fog Computing environment using OpenWhisk actions and triggers. The feasibility of the architecture is a crucial aspect that needs to be addressed, particularly in terms of scalability and real-time response.

Scalability and real-time properties are essential factors for determining the feasibility of any system architecture. In this case, they play a vital role in ensuring timely processing and computation of aggregates every 30 minutes. Additionally, having data available before the arrival of new data, preferably within a 2-minute interval, enhances the overall effectiveness of the monitoring system.

To evaluate the scalability and real-time properties of the architecture, the focus will primarily be on the edge instances and their ability to handle increasing sensor counts. By measuring response times and analyzing the results, the system's feasibility in maintaining real-time response rates as the workload grows can be assessed.

The chosen approach of utilizing OpenWhisk actions and triggers aligns well with the scalability and real-time requirements of the system. OpenWhisk actions automatically scale based on the incoming workload, ensuring that the system can handle the growing data from sensors while maintaining real-time response rates. This dynamic scalability eliminates the need for manual intervention or infrastructure adjustments, further enhancing the feasibility and real-time capabilities of the solution.

In the subsequent sections, the evaluation of scalability and real-time properties will be explored in detail. The metrics and tools employed, such as Prometheus and Grafana, will be discussed, as they enable the measurement and monitoring

of scalability and response times. These tools provide valuable insights into the system's performance, allowing for visualization and analysis of the collected data.

By systematically increasing the sensor count and measuring response times, it is possible to observe how the system handles the growing workload and evaluate its scalability and real-time properties. This evaluation will provide valuable evidence regarding the feasibility and scalability of the proposed solution for monitoring multiple data centers while ensuring real-time processing.

In conclusion, this chapter highlights the importance of evaluating the scalability and real-time properties of the experimental architecture. By focusing on the scalability and real-time response of the edge instances when adding sensors, the system's ability to handle increasing workloads and maintain the necessary real-time properties can be assessed. The subsequent sections will provide a comprehensive analysis of the chosen approach, metrics, tools, evaluation results, and additional considerations. Ultimately, this analysis will lead to a thorough conclusion regarding the feasibility, scalability, and real-time capabilities of the experimental architecture.

8.1 Chosen Properties

The chosen properties of scalability, real-time capabilities, components' integration and ease of management play pivotal roles in demonstrating the overall feasibility of the system. Each property addresses crucial aspects of the system's functionality and integration, providing valuable insights into its capabilities and effectiveness.

8.1.1 Scalability

Scalability is a pivotal property that showcases the system's ability to handle increasing workload demands and accommodate future growth [66]. It is crucial for demonstrating the feasibility of the system. In the experimental setup, limitations were faced regarding the deployment of additional edge instances due to physical constraints and resource limitations. The Kubernetes cluster on a single node with 16 GiB RAM and 6 CPU cores proved insufficient to deploy more than two edge instances and a centralized instance.

Considering these limitations, the focus was on demonstrating scalability by increasing the workload through the addition of more sensors on a single edge instance. Although it was not possible to add more edge instances to distribute the workload, this approach still allows the assessment of the system's capability to handle increasing demands and provides valuable insights into its scalability.

Acknowledging the limitation of not being able to increase the number of edge instances, the focus on scalability through increased sensor workload still allows the demonstration of the system's feasibility in handling expanding workloads. This approach provides valuable evidence of the system's ability to efficiently process data from a growing number of sensors and showcases its scalability within the given experimental constraints.

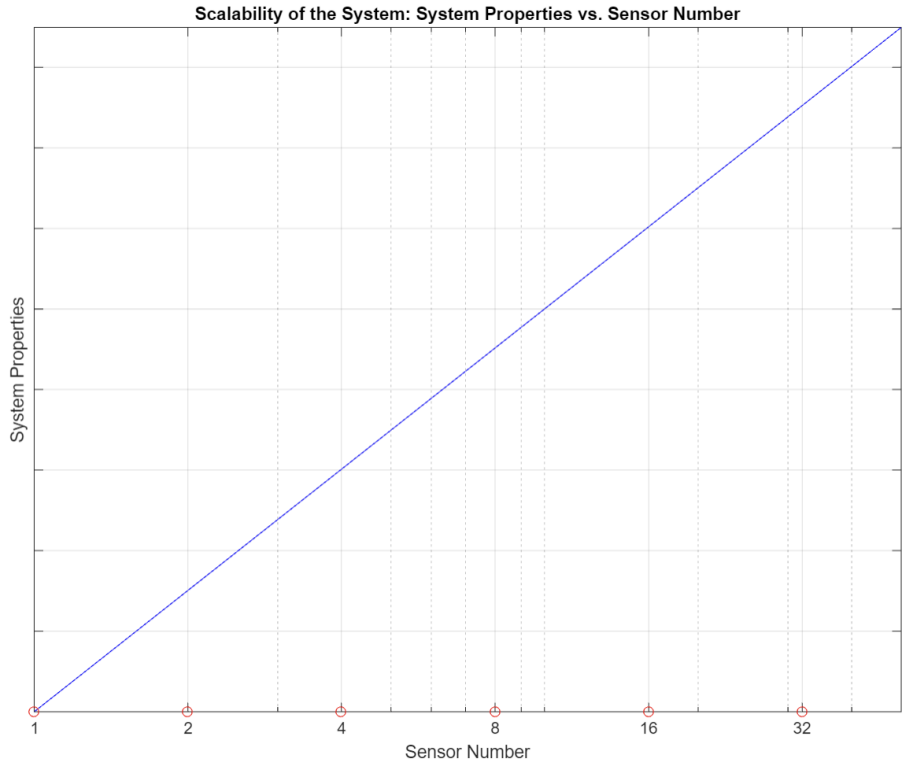

Figure 8.1: Desired System Behaviour for Scalability

8.1.2 Real Time Capabilites

When we say "real-time," we mean that the system can process and analyze data with minimal delay, ensuring that the information is available and actionable within the required time frames. Real-time capabilities are essential in scenarios where prompt decision-making and timely interventions are critical. By evaluating the system's performance in real-time scenarios, valuable evidence is obtained regarding its ability to handle and process data in a time-sensitive manner [67].

It is important to note that the choice of a 30-minute interval for computing aggregates is based on the specific requirements and constraints of the use case. However, it is possible to calculate aggregates within smaller time intervals if needed. This would increase the workload on the system, but once the scalability of the solution is demonstrated, it would be possible to handle such increased processing requirements.

Real-time capabilities are essential for demonstrating the feasibility of the system's overall functionality. By evaluating the system's real-time processing and response, it is possible to assess its ability to process and compute aggregates within the required 30-minute interval. Additionally, having data available within the 2-minute interval enhances the system's effectiveness in providing timely information. Demonstrating the feasibility of real-time capabilities showcases the system's reliability and its ability to meet the time-sensitive nature of the monitoring system. It provides evidence that the system can effectively handle and process data in real-time, contributing to its overall feasibility.

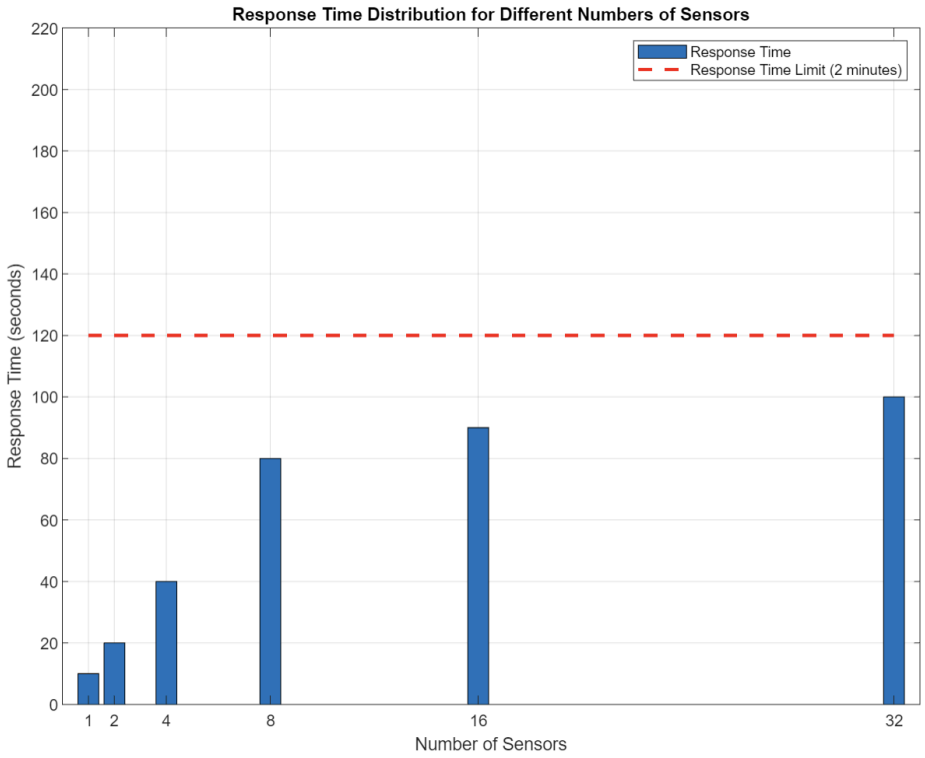

Figure 8.2: Desired System Behaviour for Real-Time Capabilities

8.1.3 Integration of System Components

The practical experimentation conducted for this thesis conclusively demonstrated the feasibility of integrating the system's components. This successful integration is vital to the overall feasibility of the system, as it ensures the smooth collaboration and interaction of the components, enabling the system to function effectively and achieve its objectives.

Furthermore, it is evident from the experimental results that the integration of components has been successfully achieved. Therefore, there is no need to reiterate the demonstration of practical feasibility.

Figure 8.3: Integration of System Components

8.1.4 Ease of System Managment

The feasibility of the system is further enhanced by the ease of system management provided by the centralized storage of OpenWhisk actions. This centralized approach, made possible by the decision to use a centralized CouchDB instance, simplifies the overall management of the system and offers several advantages.

During the architecture implementation phase, it was decided to use a centralized CouchDB instance to store the code for all the actions and triggers of the edge instances. This architectural decision was motivated by the need for centralized management. By utilizing the centralized CouchDB, the system administrator gains control over the configuration and deployment of actions, enabling efficient management of the system as a whole.

Through the experimental deployment phase, it was observed that instead of creating and managing individual actions and triggers for each edge instance, a single action and trigger could be created in the centralized instance and shared between edge instances. This streamlined approach reduces complexity and effort required for system management tasks, leading to more efficient management of the system.

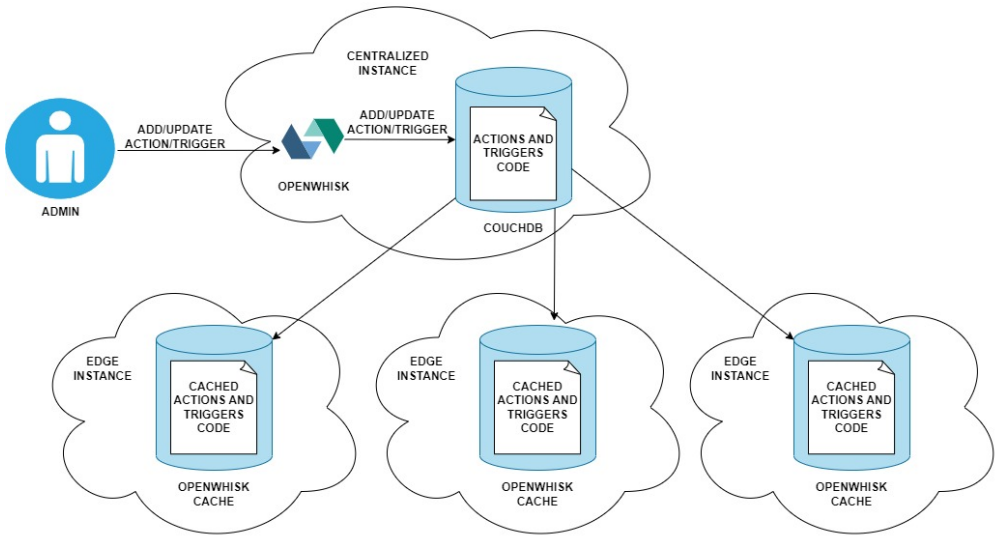

Figure 8.4: Centralized Managment with Single CouchDB

The decision to use a centralized CouchDB instance also facilitates swift and centralized modifications of actions. The system administrator can easily update or replace actions in the centralized database, since the action code is retrieved from

the centralized CouchDB. This ensures that all edge instances consistently use the most up-to-date and optimized versions of the actions, enhancing the system's performance and functionality.

In summary, the feasibility of the system is enhanced by its ease of management. The centralized storage of OpenWhisk actions enables swift and efficient modifications, simplifies system maintenance, and facilitates system upgrading. This centralized control alleviates the burden on the edge instances, allowing them to focus on their primary tasks. By providing a streamlined management process, the system ensures consistent usage of the most up-to-date actions, enhancing its overall feasibility. Since the nature and effectiveness of this property of the system is extremely clear, it won't be proved in the next sections of this chapter, as it happens for scalability and real-time capabilities.

8.2 Evaluation Methodology

To evaluate the scalability and real-time properties of the experimental architecture, a systematic methodology was employed. This methodology aimed to assess the system's ability to handle increasing workloads, maintain real-time response rates, and verify the feasibility of the proposed solution for monitoring multiple data centers. The following sections outline the key steps and procedures involved in the evaluation.

8.2.1 Measurement of Response Times

In order to evaluate the scalability and real-time properties of the system, the response times were measured for different message rates. However, due to limitations in directly increasing the number of sensors, an alternative approach was adopted. Instead, messages were published to the system at varying rates, simulating the effect of increasing sensor counts.

To ensure stable conditions and sufficient observation time, the message rate was doubled every two hours. This allowed for a gradual increase in workload and provided ample time to monitor the system's performance. It is worth noting that during this evaluation, the simulation was capped at a maximum of 64 sensors. This limitation was imposed due to the constraints of the free distribution of OpenWhisk, which restricts the firing of only 60 triggers per minute [68].

To obtain reliable and consistent response time measurements, the initial 30 minutes following each increase in the message rate were excluded from the analysis. This exclusion period allowed the system to adapt to the higher workload and reach

a steady state, ensuring that the response time measurements were representative of the system's performance under stable conditions.

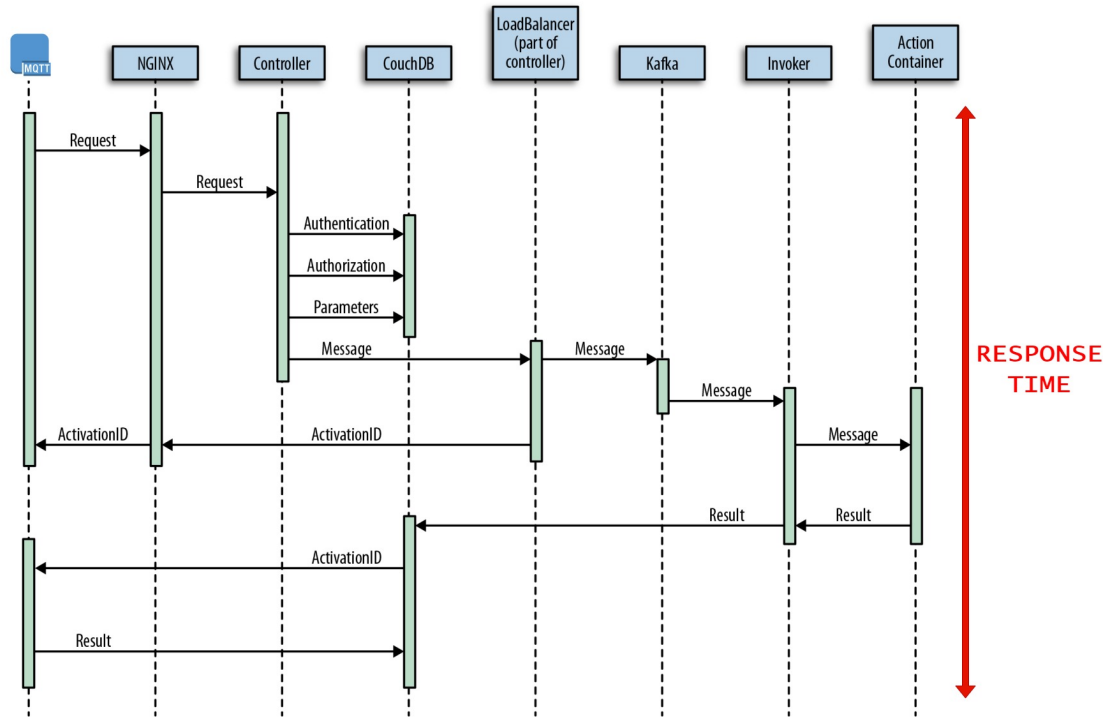

Figure 8.5: Response Time

8.2.2 Measurement of CPU and Memory Utilization

In addition to measuring response times, the average CPU and memory utilization of the centralized and edge systems were also monitored during the scalability evaluation. This provided insights into the resource consumption patterns as the message rate increased.

Considering the combined architecture approach with both the centralized and edge instances running on the same cluster, it is important to note that the response times, CPU utilization, and memory utilization might be higher compared to a distributed setup. This is due to the practical limitations imposed on the system.

Multiple measurements were taken to capture variations in CPU and memory utilization and ensure statistical accuracy.

8.2.3 Measurement of Traffic Exchanged

To further understand the system's behavior as the number of sensors increased and prove its scalability, the average traffic exchanged between the centralized and edge instances was measured. This metric provided insights into the communication load between the two components.

Similar to response times and resource utilization, the overall traffic exchanged was measured for different message rates. The message rate was doubled every two hours, and measurements were taken after the system reached a steady state following each increase.

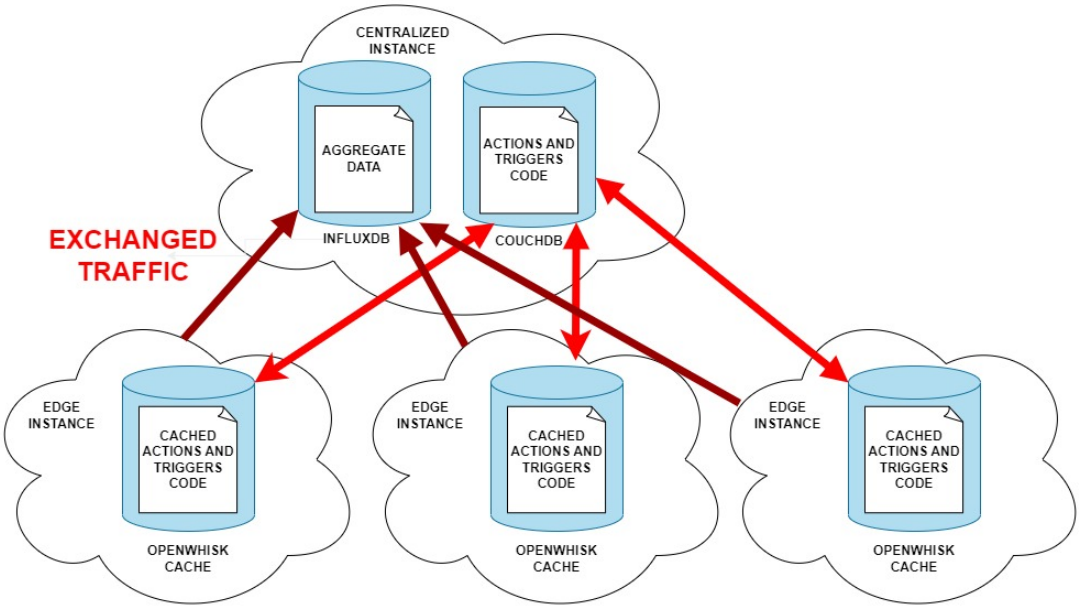

Figure 8.6: Traffic Exchanged

8.2.4 Evaluation of Fitted Models for Response Times, CPU, Memory Utilization and Traffic Exchanged

In the context of analyzing response times, CPU and memory utilization, as well as the overall traffic exchanged, mathematical models were fitted to the data. By fitting models to the observed data, the best mathematical representation that captures the relationship between the number of sensors and the corresponding metrics can be identified.

To evaluate the fit of the models, various statistical measures such as R-squared

(R^2), Root Mean Square Error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC) were employed. These measures quantified the goodness-of-fit and provided insights into the performance of the fitted models for response times, CPU and memory utilization, and traffic exchanged.

8.2.5 Analysis and Interpretation

After evaluating the fitted models for response times, CPU and memory utilization, and traffic exchanged, the results were analyzed to assess the scalability of the experimental architecture. The focus was on identifying patterns or trends in the fitted models and understanding how the performance, resource utilization, and communication load between the centralized and edge instances change as the number of sensors increases.

Considering the practical limitations and constraints of the combined architecture approach, the system demonstrated scalability within these limitations. The fitted models provided insights into how response times, CPU and memory utilization, and traffic exchanged increased as the number of sensors grew, indicating the system's capacity to handle a growing workload.

In addition to scalability, it was crucial to verify the real-time properties of the architecture. The maximum response time for each sensor was considered to assess if it exceeded a certain threshold, indicating a deviation from real-time requirements. Ensuring that data was available before the new data was sent 2 minutes later was also desirable for real-time processing.

By systematically evaluating the scalability and real-time properties of the architecture using response times, CPU and memory utilization, traffic exchanged, and fitted models, a comprehensive understanding of its performance and capacity to scale up was gained. These insights guided decision-making, optimization efforts, and further improvements to meet the desired performance and real-time requirements.

8.3 Model Fitting

In this section, the concept of model fitting is explored, which plays a crucial role in data analysis and modeling. The section discusses the significance of curve fitting and its application in various fields. Three generic curve models, namely linear, logarithmic, and exponential curves, are examined in detail. By understanding these curves, it is possible analyze and model the relationships between variables in this study.

8.3.1 Curve Fitting

In this section, the concept of curve fitting is explored, which involves finding a mathematical model that best approximates the relationship between variables based on the given data points. Curve fitting is a fundamental technique in data analysis and plays a crucial role in various fields, including science, engineering, finance, and social sciences.

Curve fitting aims to identify a curve or mathematical function that closely represents the observed data points. The chosen curve should capture the underlying patterns, trends, or relationships between the independent and dependent variables. By fitting a curve to the data, it is possible to make predictions, gain insights, and understand the behavior of the variables beyond the available data points [69].

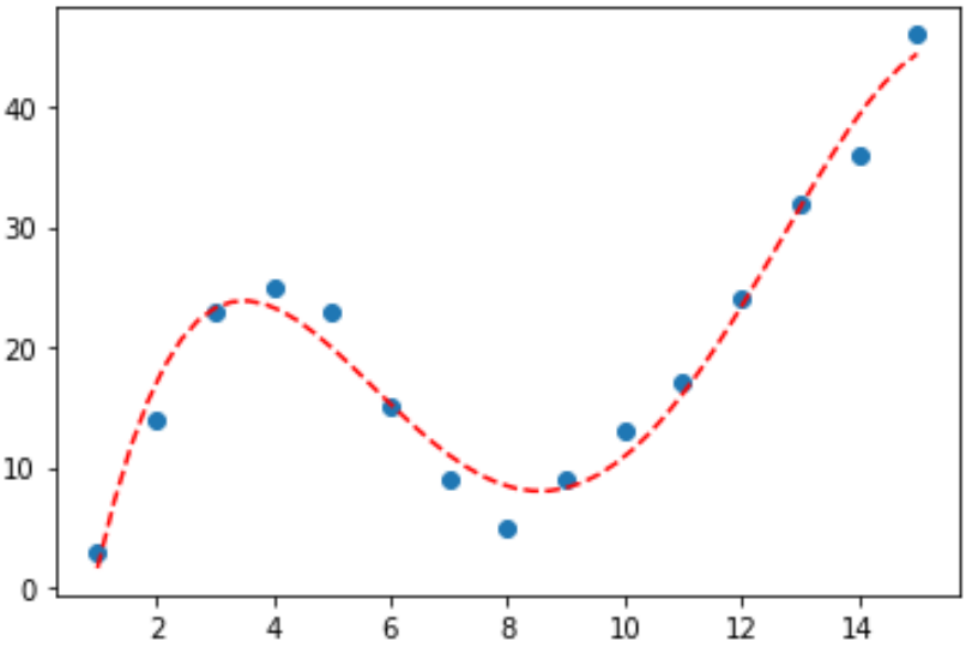

Figure 8.7: Example of Curve Fitting

Three generic curve models have been selected for fitting the data: linear, logarithmic, and exponential curves. These curves serve as the foundation for linear fitting, logarithmic fitting, and exponential fitting, respectively. These fittings are going to be called linear fitting, logarithmic fitting, and exponential fitting due to their use of the corresponding generic curves.

Linear Curve

The linear curve represents a straight line relationship between the independent variable(s) and the dependent variable. It is characterized by the equation:

$$y = mx + b \tag{8.1}$$

where:

- y is the dependent variable
- x is the independent variable
- m is the slope of the line
- b is the y-intercept of the line

Linear curves are commonly used for fitting when there is a linear relationship between the variables or when a simple and interpretable model is desired.

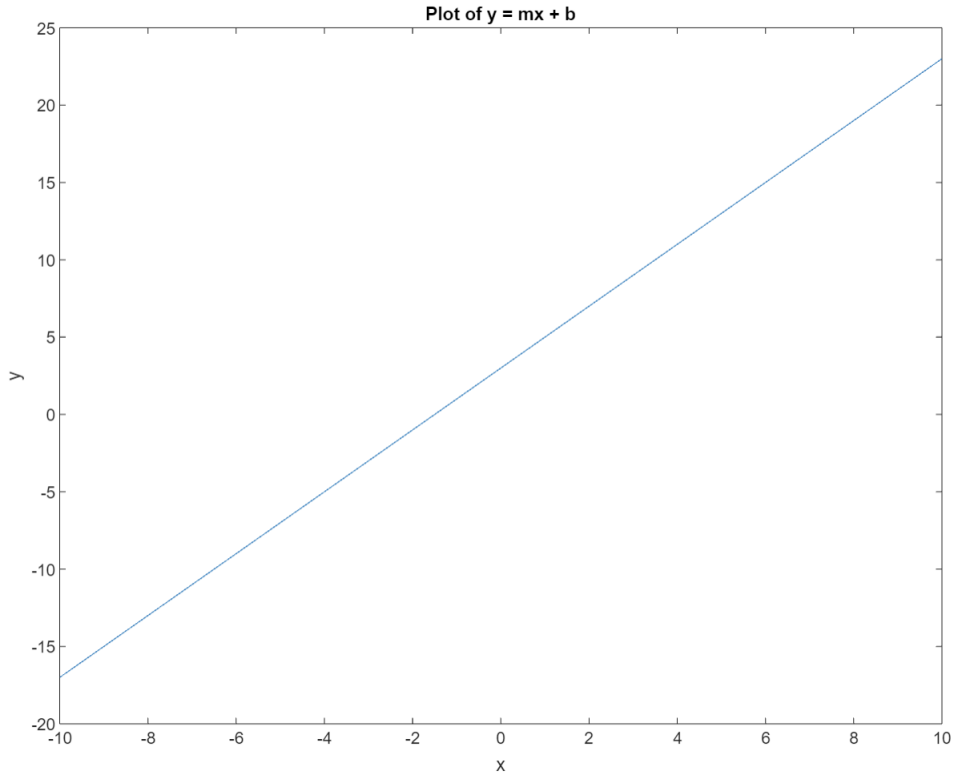

Figure 8.8: Example of a Plot for a Linear Curve

Logarithmic Curve

The logarithmic curve represents a logarithmic relationship between the independent variable(s) and the dependent variable. It is characterized by the equation:

$$y = a + b \cdot \log(x) \tag{8.2}$$

where:

- y is the dependent variable
- x is the independent variable
- a and b are coefficients to be determined

Logarithmic curves are useful for fitting when the relationship between the variables is expected to exhibit diminishing returns or when the data follows slow growth/decay.

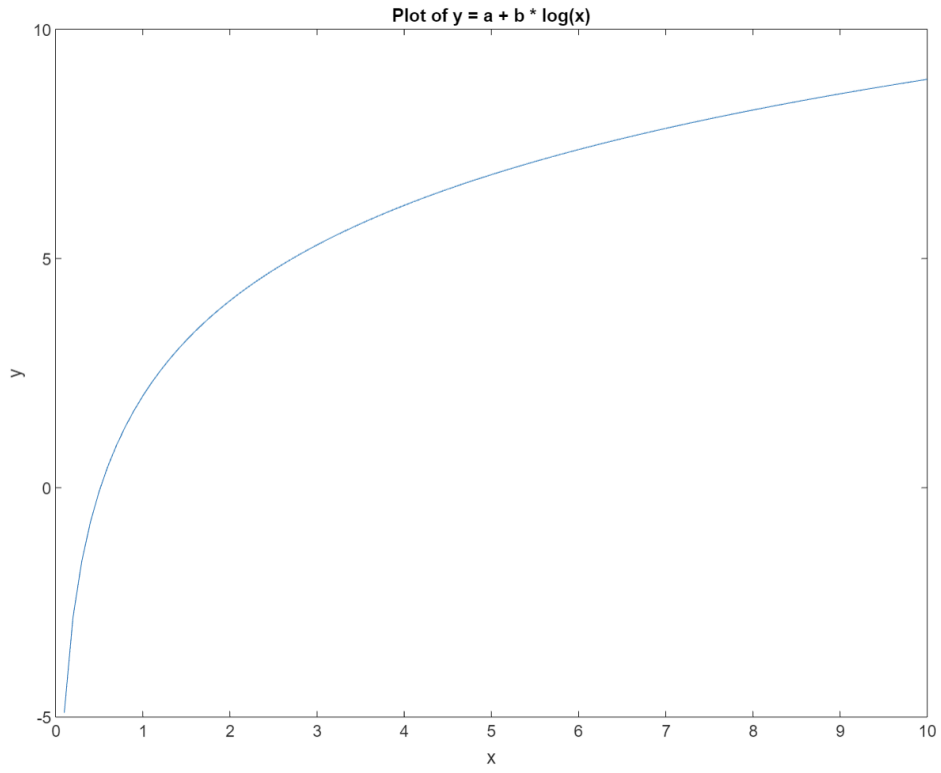

Figure 8.9: Example of a Plot for a Logarithmic Curve

Exponential Curve

The exponential curve represents an exponential relationship between the independent variable(s) and the dependent variable. It is characterized by the equation:

$$y = a \cdot e^{b \cdot x} \tag{8.3}$$

where:

- y is the dependent variable
- x is the independent variable
- a and b are coefficients to be determined
- e is the base of the natural logarithm (approximately 2.71828)

Exponential curves are suitable for fitting when the data exhibits rapid growth or decay.

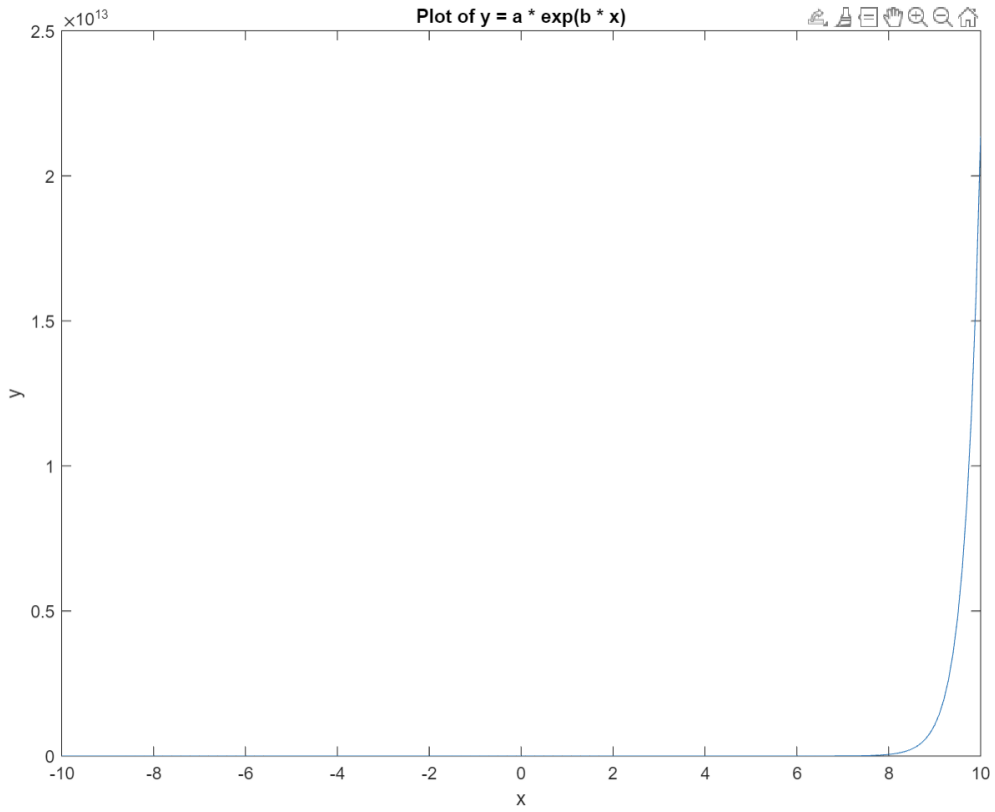

Figure 8.10: Example of a Plot for an Exponential Curve

By understanding the concept of curve fitting and the chosen generic curves, namely linear, logarithmic, and exponential curves, it is possible to analyze and model the relationships between variables in this study. From now on, the fitting based on the linear, logarithmic, and exponential curves will be referred to as linear fitting, logarithmic fitting, and exponential fitting, respectively.

8.3.2 MATLAB and Fitting Functions

In this section, the usage of MATLAB will be presented as a powerful tool for performing fittings and explore the main functions used to conduct different types of fitting analyses. MATLAB provides a comprehensive set of functions and capabilities for data analysis and modeling, making it a valuable resource for researchers and practitioners [70]. The key functions used for linear, logarithmic, and exponential fittings will be discussed.

Linear Fitting in MATLAB

MATLAB offers various functions to perform linear fitting. The primary function used is `polyfit`, which fits a polynomial of a specified degree to the data [71]. In the case of linear fitting, a first-degree polynomial (a straight line) is fitted to the data points. The syntax for using `polyfit` is as follows:

```
coefficients = polyfit(x, y, degree)
```

where:

- x and y are the independent and dependent variables, respectively
- *degree* specifies the degree of the polynomial (1 for linear fitting)

The output of `polyfit` is a set of coefficients representing the slope and y-intercept of the line. These coefficients can be used to evaluate the linear fit equation and analyze the goodness of fit.

Logarithmic Fitting in MATLAB

MATLAB provides the function `lsqcurvefit` to perform nonlinear least squares fitting [72]. For logarithmic fitting, a custom model function that captures the logarithmic relationship between the variables has been defined. The syntax for using `lsqcurvefit` for logarithmic fitting is as follows:

```
coefficients = lsqcurvefit(model, initialGuess, x, y)
```

where:

- *model* is a function handle representing the logarithmic model equation
- *initialGuess* is an initial guess for the model coefficients
- x and y are the independent and dependent variables, respectively.

The output of `lsqcurvefit` is a set of coefficients that provide the best fit to the data. These coefficients can be used to evaluate the logarithmic fit equation and assess the goodness of fit.

Exponential Fitting in MATLAB

Similar to logarithmic fitting, MATLAB's `lsqcurvefit` function is used for exponential fitting as well. A custom model function that represents the exponential relationship between the variables has been defined. The syntax for using `lsqcurvefit` for exponential fitting is as follows:

```
coefficients = lsqcurvefit(model, initialGuess, x, y)
```

where:

- *model* is a function handle representing the exponential model equation
- *initialGuess* is an initial guess for the model coefficients
- x and y are the independent and dependent variables, respectively.

The output of `lsqcurvefit` is a set of coefficients that provide the best fit to the data. These coefficients can be used to evaluate the exponential fit equation and assess the goodness of fit.

By utilizing these MATLAB functions, it is possible to perform linear, logarithmic, and exponential fittings on your data.

8.3.3 Error Rates and Model Evaluation

In this section, the focus is on the error rates and model evaluation metrics that were used to determine the best fit among the different models. These metrics provide valuable insights into the quality of the fitted models and assist in the selection of the most appropriate model for the given data. The coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC) will be discussed.

Coefficient of Determination (R-squared)

The coefficient of determination, commonly referred to as R-squared, measures the proportion of the variance in the dependent variable (for example, average CPU utilization) that can be explained by the independent variable (number of sensors). R-squared ranges from 0 to 1, where 0 indicates that the model does not explain any variability in the data, and 1 indicates a perfect prediction by the model [73].

A data set has n values marked $y_1, \ldots, y_n$ (collectively known as y_i or as a vector $y = [y_1, \ldots, y_n]^T$), each associated with a fitted (or modeled, or predicted) value $f_1, \ldots, f_n$ (known as f_i, or sometimes $\hat{y}_i$, as a vector f). Define the residuals as $e_i = y_i - f_i$ (forming a vector e). If $\bar{y}$ is the mean of the observed data:

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{8.4}$$

then the variability of the data set can be measured with two sums of squares formulas: The sum of squares of residuals, also called the residual sum of squares:

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2 \tag{8.5}$$

The total sum of squares (proportional to the variance of the data):

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2 \tag{8.6}$$

The most general definition of the coefficient of determination is:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \tag{8.7}$$

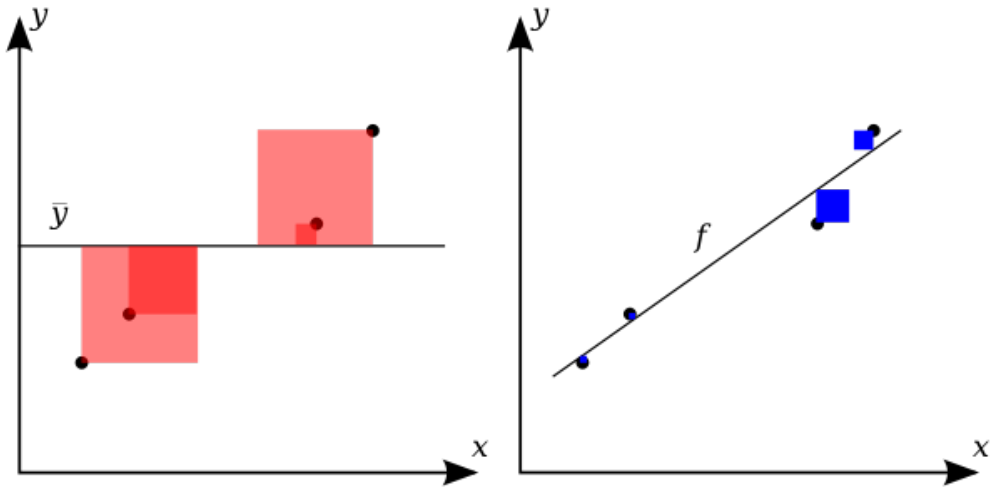

Figure 8.11: R-Squared Visual Explanation
The areas of the blue squares represent the squared residuals with respect to the linear regression. The areas of the red squares represent the squared residuals with respect to the average value.

Root Mean Squared Error (RMSE)

RMSE provides a measure of the average deviation between the predicted values of the model and the actual data points. It quantifies how well the model's predictions align with the observed data. RMSE is expressed in the same units as the dependent variable, and lower values indicate a better fit [74].

The root mean squared error (RMSE) can be calculated using the formula:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2} \tag{8.8}$$

In this formula, y represents the predicted values of the model, x represents the actual values of the dependent variable, and n represents the number of data points.

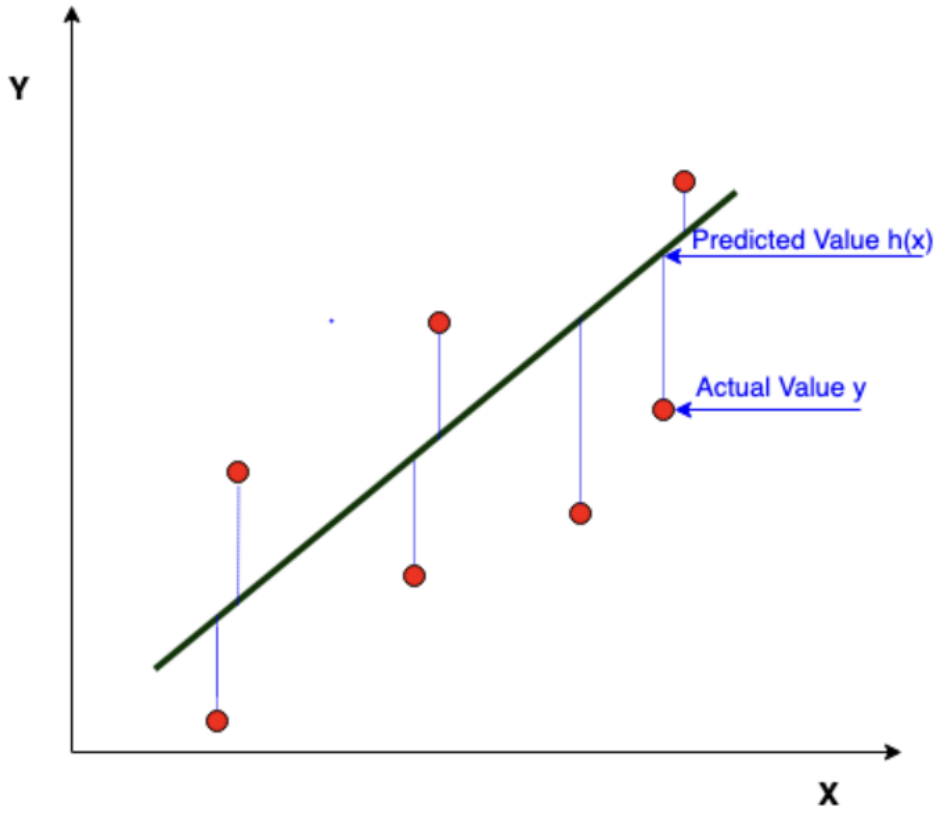

Figure 8.12: Predicted Values and Actual Values in RMSE

Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC)

AIC and BIC are information criteria used to compare the goodness of fit of different models. These criteria take into account both the goodness of fit and the complexity of the model, providing a balanced assessment. Lower values of AIC and BIC indicate a better fit, considering the trade-off between goodness of fit and model complexity [75][76].
The BIC is formally defined as:

$$\mathrm{BIC} = \ln(n) - 2\ln(\hat{L}) \tag{8.9}$$

where:

- $\hat{L}$ is the maximized value of the likelihood function of the model M, i.e., $\hat{L} = p(x|\hat{\theta}, M)$
- $\hat{\theta}$ are the parameter values that maximize the likelihood function

- x is the observed data
- n is the number of data points in x, the number of observations, or equivalently, the sample size
- k is the number of parameters estimated by the model. For example, in multiple linear regression, the estimated parameters are the intercept, the q slope parameters, and the constant variance of the errors; thus, $k = q + 2$

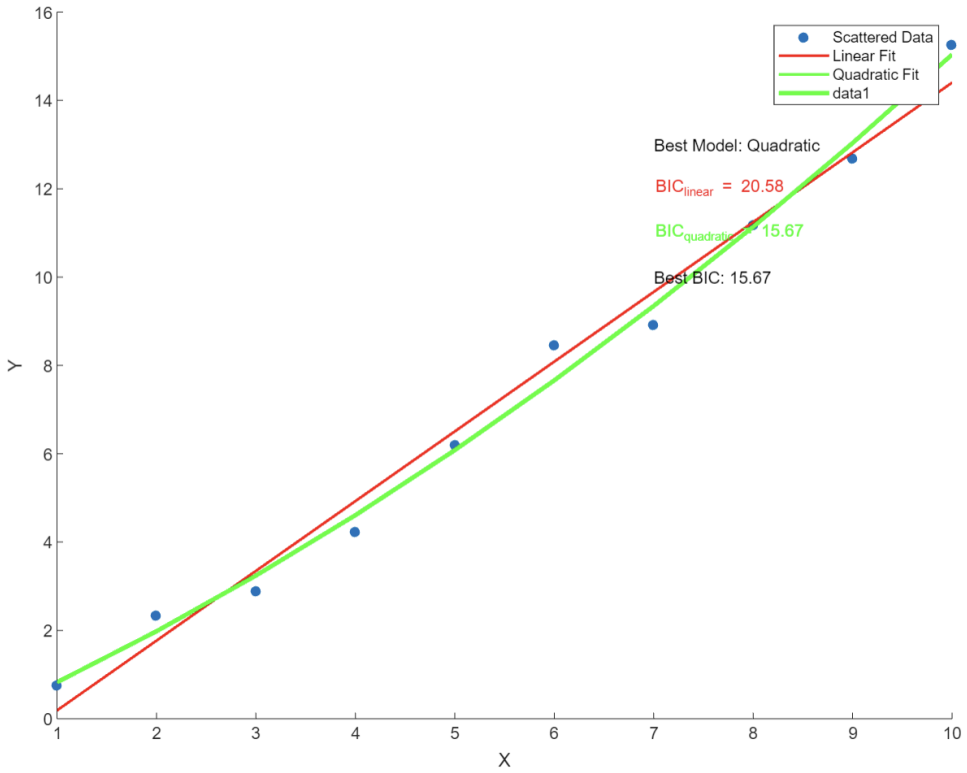

Figure 8.13: Different BIC for Different Curve Fittings

The AIC (Akaike Information Criterion) value of the model is defined as:

$$\text{AIC} = 2k - 2\ln(\hat{L}) \tag{8.10}$$

where:

- k is the number of estimated parameters in the model
- $\hat{L}$ is the maximized value of the likelihood function for the model

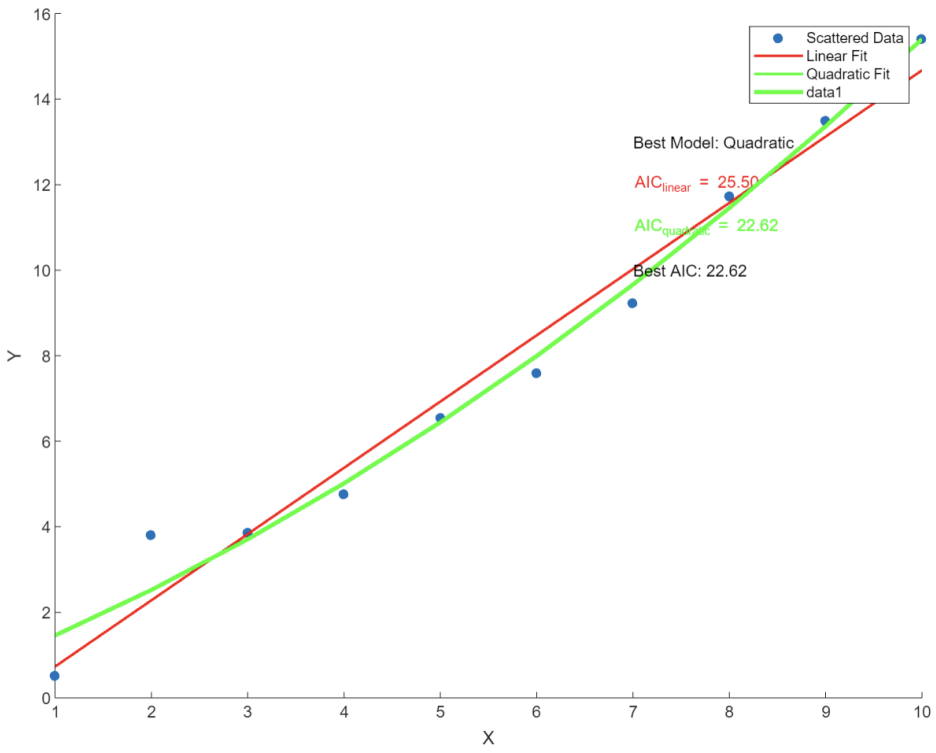

Figure 8.14: Different AIC for Different Curve Fittings

By considering these error rates and evaluation metrics, it is possbile to make informed decisions regarding the best fit for the collected data.

8.3.4 MATLAB Code Example

This section is focused on the presentation of the MATLAB code used to perform linear, logarithmic, and exponential fitting on the different considered metrics. The code snippet that will be subsequently included is related to the average CPU utilization data, but it has been easily adapted to the other kind of metrics.

```
% Data
numSensors = [1 2 4 8 16 32 64];
cpuUtilization = [11.00 11.00 11.66 12.50 12.64 15.49 16.39];

% Linear fitting
linearCoefficients = polyfit(numSensors, cpuUtilization, 1);
linearFit = polyval(linearCoefficients, numSensors);
```

```
linearRsq = 1 - sum((cpuUtilization - linearFit).^2) / sum((
    cpuUtilization - mean(cpuUtilization)).^2);
linearRMSE = sqrt(mean((cpuUtilization - linearFit).^2));

% Logarithmic fitting
logarithmicModel = @(coefficients, x) coefficients(1) + coefficients
    (2) * log(x);
logarithmicInitialGuess = [0 0]; % Initial guess for coefficients
logarithmicCoefficients = lsqcurvefit(logarithmicModel,
    logarithmicInitialGuess, numSensors, cpuUtilization);
logarithmicFit = logarithmicModel(logarithmicCoefficients, numSensors
    );
logarithmicRsq = 1 - sum((cpuUtilization - logarithmicFit).^2) / sum
    ((cpuUtilization - mean(cpuUtilization)).^2);
logarithmicRMSE = sqrt(mean((cpuUtilization - logarithmicFit).^2));

% Exponential fitting
exponentialModel = @(coefficients, x) coefficients(1) * exp(
    coefficients(2) * x);
exponentialInitialGuess = [1 0]; % Initial guess for coefficients
exponentialCoefficients = lsqcurvefit(exponentialModel,
    exponentialInitialGuess, numSensors, cpuUtilization);
exponentialFit = exponentialModel(exponentialCoefficients, numSensors
    );
exponentialRsq = 1 - sum((cpuUtilization - exponentialFit).^2) / sum
    ((cpuUtilization - mean(cpuUtilization)).^2);
exponentialRMSE = sqrt(mean((cpuUtilization - exponentialFit).^2));

% Plotting
figure;
plot(numSensors, cpuUtilization, 'bo', 'MarkerSize', 8);
hold on;
plot(numSensors, linearFit, 'r-', 'LineWidth', 1.5);
plot(numSensors, logarithmicFit, 'g-', 'LineWidth', 1.5);
plot(numSensors, exponentialFit, 'm-', 'LineWidth', 1.5);
xlabel('Number of Sensors');
ylabel('CPU Utilization (%)');
legend('Data', 'Linear Fit', 'Logarithmic Fit', 'Exponential Fit');
title('Fitting of CPU Utilization');

% Display equations of fitting
linearEquation = sprintf('Linear Fit: y = %.8f * x + %.8f',
    linearCoefficients(1), linearCoefficients(2));
logarithmicEquation = sprintf('Logarithmic Fit: y = %.8f + %.8f * log
    (x)', logarithmicCoefficients(1), logarithmicCoefficients(2));
exponentialEquation = sprintf('Exponential Fit: y = %.8f * exp(%.8f *
     x)', exponentialCoefficients(1), exponentialCoefficients(2));
disp(linearEquation);
disp(logarithmicEquation);
```

```
45 disp(exponentialEquation);
46
47 % Display metrics
48 disp(['Linear Fit - R-squared: ' num2str(linearRsq)]);
49 disp(['Linear Fit - RMSE: ' num2str(linearRMSE)]);
50 disp(['Logarithmic Fit - R-squared: ' num2str(logarithmicRsq)]);
51 disp(['Logarithmic Fit - RMSE: ' num2str(logarithmicRMSE)]);
52 disp(['Exponential Fit - R-squared: ' num2str(exponentialRsq)]);
53 disp(['Exponential Fit - RMSE: ' num2str(exponentialRMSE)]);
54
55 % Calculate AIC and BIC
56 n = length(cpuUtilization);
57 linearAIC = n * log(linearRMSE^2) + 2 * (length(linearCoefficients) +
       1);
58 logarithmicAIC = n * log(logarithmicRMSE^2) + 2 * (length(
      logarithmicCoefficients) + 1);
59 exponentialAIC = n * log(exponentialRMSE^2) + 2 * (length(
      exponentialCoefficients) + 1);
60 linearBIC = n * log(linearRMSE^2) + log(n) * (length(
      linearCoefficients) + 1);
61 logarithmicBIC = n * log(logarithmicRMSE^2) + log(n) * (length(
      logarithmicCoefficients) + 1);
62 exponentialBIC = n * log(exponentialRMSE^2) + log(n) * (length(
      exponentialCoefficients) + 1);
63 disp(['Linear Fit - AIC: ' num2str(linearAIC)]);
64 disp(['Linear Fit - BIC: ' num2str(linearBIC)]);
65 disp(['Logarithmic Fit - AIC: ' num2str(logarithmicAIC)]);
66 disp(['Logarithmic Fit - BIC: ' num2str(logarithmicBIC)]);
67 disp(['Exponential Fit - AIC: ' num2str(exponentialAIC)]);
68 disp(['Exponential Fit - BIC: ' num2str(exponentialBIC)]);
```

8.4 Metrics And Tools

This section covers the metrics and tools used in the monitoring and analysis of the experimental architecture.

8.4.1 Prometheus Overview

Prometheus is an open-source monitoring system widely used in the DevOps and system monitoring domain. It serves as a data collection and storage system, enabling the gathering, storage, and analysis of time-series data. Prometheus employs a pull-based model to collect metrics from various sources, making it highly flexible and compatible with different applications and platforms [77].

It is possbile to retrieve the helm charts used to deploy Prometheus and Prometheus

Operator in the `/openwhisk/metrics` folder of the GitHub repository for this thesis[1].

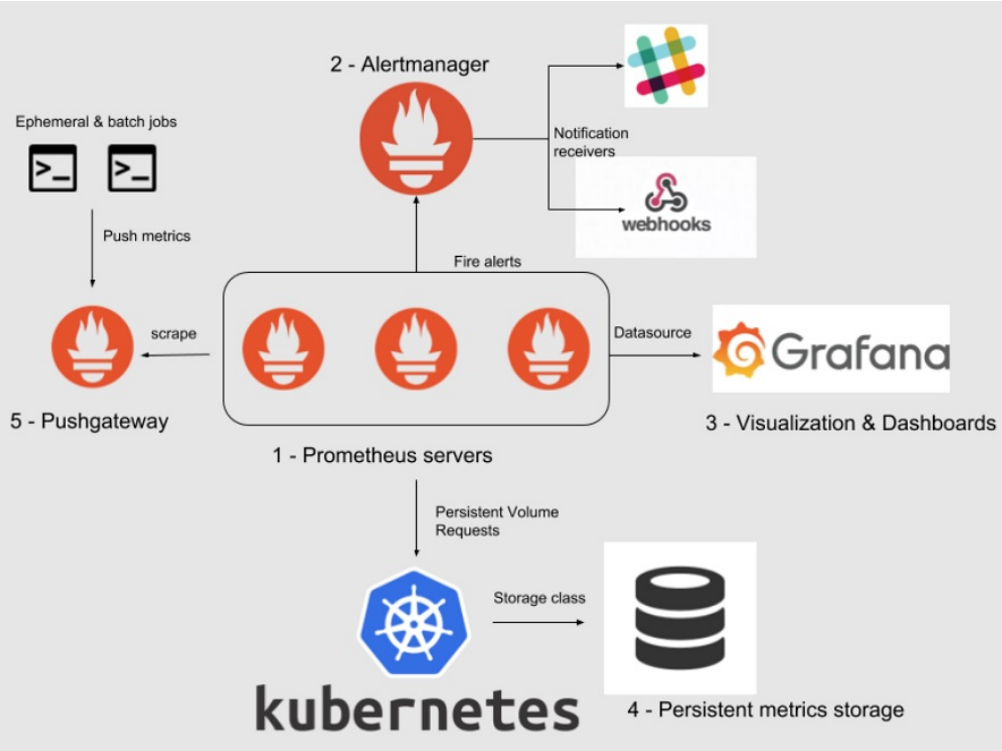

Figure 8.15: General Architecture with Prometheus

8.4.2 Exporting Metrics from OpenWhisk

In the context of the experimental architecture, OpenWhisk exports different metrics through the invoker, user events, and controller [78] on port 8080. These metrics provide valuable insights into the performance and behavior of the OpenWhisk platform. By exporting metrics from OpenWhisk, Prometheus can effectively collect and store these metrics for further analysis.

To enable OpenWhisk to export metrics, it is important to add this section to the `values.yaml` file used when deploying the OpenWhisk helm chart:

```
1 metrics:
```

[1] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/metrics`

```
    # set true to enable prometheus exporter
    prometheusEnabled: true
    # passing prometheus-enabled by a config file, required by
    openwhisk
    whiskconfigFile: "whiskconfig.conf"
    # set true to enable Kamon
    kamonEnabled: true
    # set true to enable Kamon tags
    kamonTags: true
    # set true to enable user metrics
    userMetricsEnabled: true
```

It it also important to create the following **ServiceMonitor**s and **PodMonitor**, adapting their code with the right namespace of the OpenWhisk deployment:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: openwhisk-controller
  namespace: openwhisk-second
spec:
  endpoints:
  - path: /metrics
    port: http
  selector:
    matchLabels:
      name: openwhisk-controller
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: openwhisk-user-events
  namespace: openwhisk-second
spec:
  endpoints:
  - path: /metrics
    port: http
  selector:
    matchLabels:
      name: openwhisk-user-events
---
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: openwhisk-invoker
  namespace: openwhisk-second
spec:
  podMetricsEndpoints:
```

```
34   - port: invoker
35     path: /metrics
36   selector:
37     matchLabels:
38       name: openwhisk-invoker
39
```

8.4.3 Exporting Metrics with k3s Exporter Stack

To gather the desired metrics for monitoring and analysis, a k3s exporter stack is utilized. This stack consists of multiple exporters that extract and expose specific metrics from the k3s cluster. The exporters used in this stack include:

- **Node Exporter**: The Node Exporter is responsible for collecting and exporting metrics related to the host machine. It provides information about CPU usage, memory utilization, disk I/O, and network statistics. By scraping the metrics exposed by the Node Exporter, it is possible to monitor the host-level resource consumption [79].
- **kube-state-metrics Exporter**: The kube-state-metrics Exporter collects metrics about the state of various Kubernetes objects, such as deployments, pods, services, and namespaces. These metrics include the number of running pods, the status of deployments, and resource allocations. By leveraging the kube-state-metrics Exporter, it is possible to gain insights into the overall health and status of the Kubernetes cluster [80]

These exporters work together to scrape and export the desired metrics from different components of the k3s cluster. The Prometheus server acts as the central data source, collecting and storing the metrics exposed by these exporters. It enables efficient monitoring and analysis of the cluster's performance.

It is possbile to retrieve the helm chart used to enable the collection of the node and kube metrics in the `/openwhisk/metrics/k3s-exporter-stack` folder of the GitHub repository for this thesis[2].

8.4.4 Prometheus as a Data Source for Grafana

To gather the desired information and visualize the metrics, Prometheus serves as the data source for Grafana. Grafana is a popular open-source platform for data

[2] `https://github.com/s294547/Exploring-FaaS-Solutions/tree/main/openwhisk/metrics/k3s-exporter-stack`

visualization and monitoring. By integrating Prometheus with Grafana, it becomes possible to create dashboards and visualize the collected metrics in a meaningful and informative way [81].

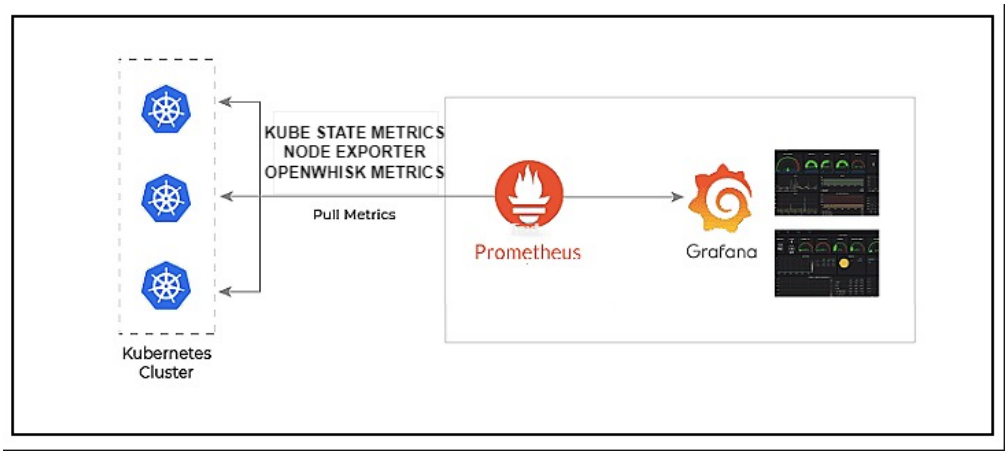

Figure 8.16: Prometheus as a Data Source for Grafana

8.4.5 Average Response Time and Maximum Response Time Metric

One of the key metrics considered for measuring scalability is the average response time. This metric provides insights into the performance of the system in terms of processing requests and generating responses. The average response time is calculated using the provided expression, which takes into account various factors such as action duration, wait time, and initialization time [82].

To measure the average response time, the PromQL query below is used:

```
rate(openwhisk_action_duration_seconds_sum{region=~"()",stack
=~"()",namespace=~"openwhisk-second",action=~"addReadingToDb",
initiator=~"guest"}[15m]) * 1000 / rate(
openwhisk_action_duration_seconds_count{region=~"()",stack=~"()",
namespace=~"openwhisk-second",action=~"addReadingToDb",initiator
=~"guest"}[15m])  + rate(openwhisk_action_waitTime_seconds_sum{
region=~"()",stack=~"()",namespace=~"openwhisk-second",action=~"
addReadingToDb",initiator=~"guest"}[15m]) * 1000 / rate(
openwhisk_action_waitTime_seconds_count{region=~"()",stack=~"()",
namespace=~"openwhisk-second",action=~"addReadingToDb",initiator
=~"guest"}[15m])   + rate(openwhisk_action_iniitTime_seconds_sum{
region=~"()",stack=~"()",namespace=~"openwhisk-second",action=~"
addReadingToDb",initiator=~"guest"}[15m]) * 1000 / rate(
openwhisk_action_initTime_seconds_count{region=~"()",stack=~"()",
namespace=~"openwhisk-second",action=~"addReadingToDb",initiator
=~"guest"}[15m])
```

This query is calculating an average response time for the action `addReadingToDb` in the OpenWhisk system, over a 15-minute time window. This expression combines the durations of the action execution, wait time, and initialization time. By calculating the rate of these durations and dividing them by their respective count rates, it is possible to obtain the average response time in milliseconds. The step for this query is set to 30 seconds, ensuring a value is computed every 30 seconds to determine the average response time.

By selecting the right time window in which a certain number of sensors has been used, it is also possible to retrieve the maximum response time of the average response times computed every 15 minutes. It is also possible to compute the overall average response time, computing the average of the average response times in the right time window.

8.4.6 CPU Utilization Metric

CPU utilization is another crucial metric for assessing scalability. It indicates the extent to which the system's CPU resources are being utilized. By measuring CPU utilization, it becomes possible to evaluate the efficiency and capacity of the system to handle increasing workloads [82].

To measure the average CPU utilization for the edge and centralized instance, the PromQL queries below are used:

```
sum(rate(container_cpu_usage_seconds_total{namespace="openwhisk-
second"}[1m]))  / sum (machine_cpu_cores)
```

```
sum(rate(container_cpu_usage_seconds_total{namespace="openwhisk-central"}[1m])) / sum (machine_cpu_cores)
```

This expression calculates the CPU utilization over a 1-minute time window and normalizing it by dividing it by the total number of CPUs. The selection of the namespace give the opportunity to choose the edge/centralized instance. The step for this query is set to 30 seconds, ensuring a value is computed every 30 seconds to determine the average CPU utilization.

By selecting the right time window in which a certain number of sensors has been used, it is also possible to retrieve the overall average CPU utilization from the average CPU utilization values computed every minute for each instance.

8.4.7 RAM Utilization Metric

RAM utilization is an essential metric for understanding the system's memory usage. It provides insights into how efficiently the system is managing and utilizing available memory resources. By monitoring RAM utilization, it becomes possible to assess the scalability of the architecture and ensure that memory resources are effectively allocated [82].

To measure the average RAM utilization for the edge and centralized instances, the PromQL queries below are used:

```
sum(container_memory_usage_bytes{namespace="openwhisk-second"}) - sum(container_memory_usage_bytes{pod="openwhisk-influxdb2-0", namespace="openwhisk-second"})
```

```
sum(container_memory_usage_bytes{namespace="openwhisk-central"}) - sum(container_memory_usage_bytes{pod="openwhisk-influxdb2-0", namespace="openwhisk-central"})
```

This expression calculates the RAM utilization for a certain instance (centralized or edge) by subtracting the space occupied by the data saved for the sensors or the aggregates on InfluxDB. The result represents the utilized RAM of an instance. The step for this query is set to 30 seconds, ensuring a value is computed 30 seconds to determine the average RAM utilization.

By selecting the right time window in which a certain number of sensors has been used, it is also possible to retrieve the overall average RAM utilization of

a certain instance from the average RAM utilization values computed every 30 seconds.

8.4.8 Network Troughput Metric

Network throughput is a crucial metric for monitoring the amount of data transferred between the centralized and edge instance. By measuring network throughput, it becomes possible to identify bottlenecks and optimize the network performance [82].

To measure the network throughput for the centralized CouchDB and InfluxDB instances, the following PromQL query is used:

```
1    irate(container_network_receive_bytes_total{device="interfaceID
    "}[$__rate_interval])*8 + irate(
    container_network_transmit_bytes_total{device="interfaceID"}[
    $__rate_interval])*8
```

This expression calculates the network throughput by summing the receive and transmit rates of bytes for a specific interface, represented by $interfaceID$. The rates are multiplied by 8 to convert them from bytes to bits. The *irate* function calculates the per-second rate of change for the metric over the given interval, specified by variable in square brackets. The step for this query is set to 15 seconds, ensuring a value is computed every 15 seconds to determine the network throughput.

This query is going to be executed four times on four different interfaces, which are the one of the centralized instance attached to CouchDB and InfluxDB. They are going to be summed up together, in order to get the overall network throughput.

By using this query for four different interfaces attached to the centralized CouchDB and InfluxDB, it is possible to monitor the network throughput between the edge and centralized instances. The exchanges between them occur when there is a cache miss, involving a reading in CouchDB, and whene there is the writing of aggregates in InfluxDB. Monitoring the network throughput helps assess the performance and efficiency of these data exchanges.

By selecting the appropriate time window, it is also possible to analyze the average network throughput for a certain period, providing insights into the overall network performance.

8.5 Evaluation Results and Analysis

In this section, the results obtained from the scalability and real-time evaluation are presented and analyzed in detail. The focus is on the metrics of response times, CPU utilization, and memory utilization at different message rates. The analysis aims to highlight trends and notable observations observed in the data.

8.5.1 Response Time Analysis

The response time analysis was conducted by computing the average response time for each number of sensors based on the averages calculated every 30 seconds. The formula used to compute the average response time for each number of sensors incorporates the duration, wait time, and initialization time metrics obtained from the OpenWhisk platform for the `addReadingToDb` action initiated by the `guest` user. The values are collected at a 30-second interval over a 15-minute window.

To provide a visual representation of the average response time at steady state for each number of sensors, the following images display the graphs:

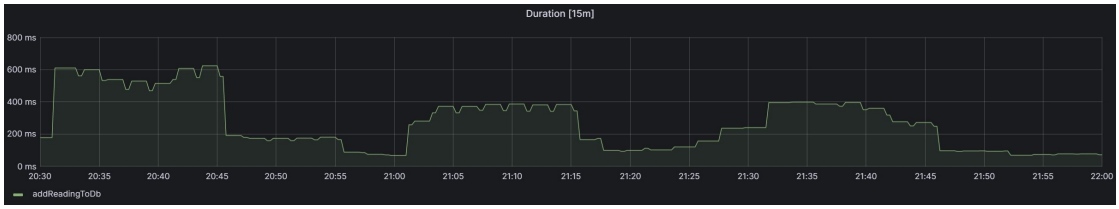

Figure 8.17: Average Response Time for 1 Sensor

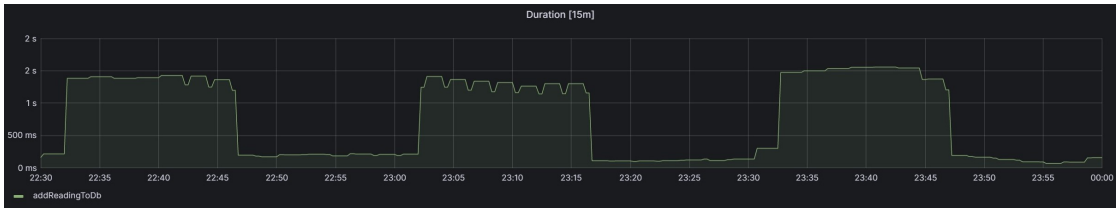

Figure 8.18: Average Response Time for 2 Sensors

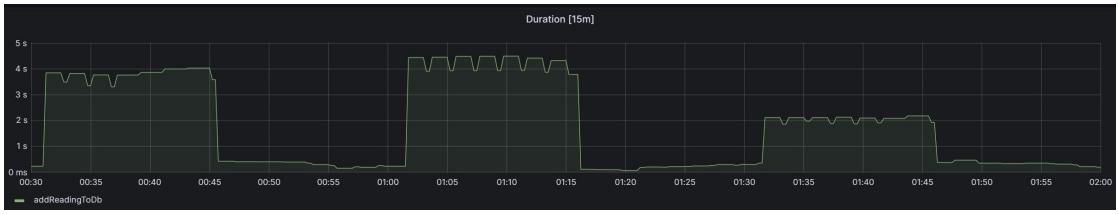

Figure 8.19: Average Response Time for 4 Sensors

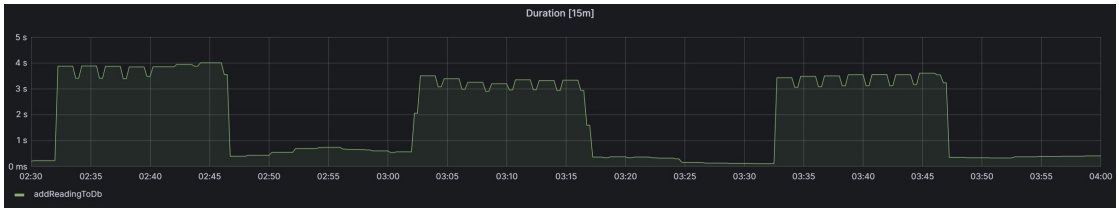

Figure 8.20: Average Response Time for 8 Sensors

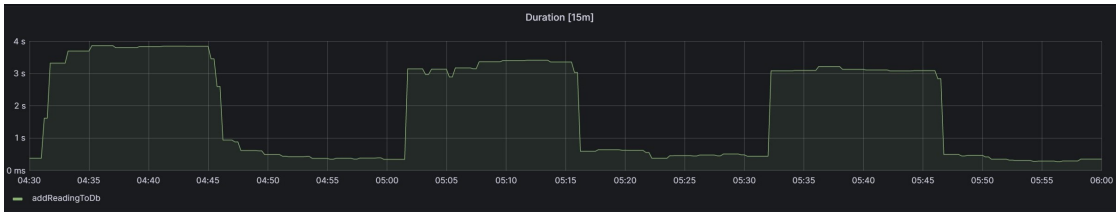

Figure 8.21: Average Response Time for 16 Sensors

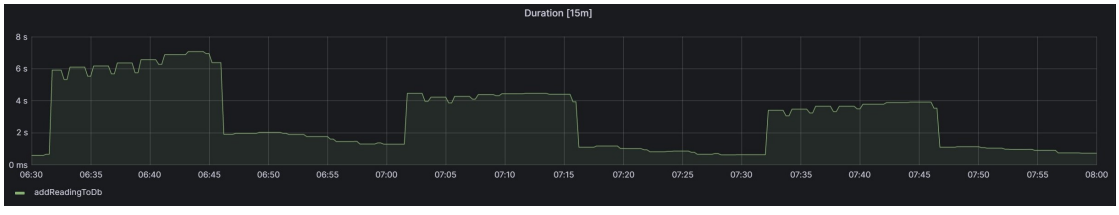

Figure 8.22: Average Response Time for 32 Sensors

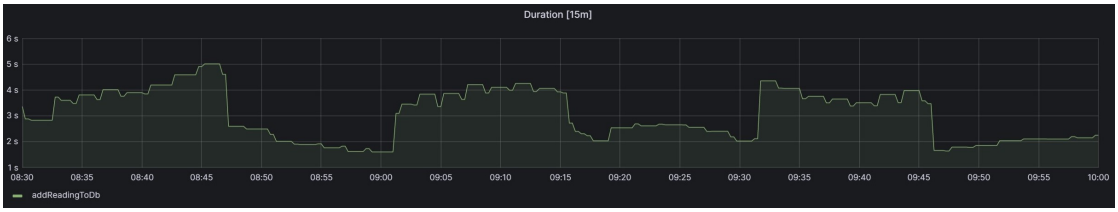

Figure 8.23: Average Response Time for 64 Sensors

The response times were computed as the average value for each tried number of sensors, providing a comprehensive view of the system's performance. It's important to note that in some regions, the average response time appears slightly higher. This can be attributed to the execution of additional OpenWhisk actions, specifically the actions to compute aggregates. They are triggered every half an hour to process and aggregate the collected sensor data. During their execution, they may temporarily increase the overall response time of the system. However, it is important to consider that these periodic actions have a minimal impact on the overall scalability and efficiency of the system.

The following table displays the average response time (in milliseconds) for each message rate:

Table 8.1: Average Response Time for Different Numbers of Sensors

Number of Sensors	Average Response Time (ms)
1	329.32
2	797.27
4	1825.76
8	1890.68
16	1863.93
32	2907.03
64	3040.28

Analysis of Average Response Time Trends

To gain insights into the relationship between the number of sensors and the average response time, various fitting techniques were performed using MATLAB. The aim was to find the best mathematical model that accurately represents the observed trend. Three different fitting approaches have been considered: linear, logarithmic, and exponential. The results of these fittings indicate the following relationships between the number of sensors (x) and the average response time (y) according to

each fitting model:

$$\text{Linear Fit: } y = 35.53961490 \cdot x + 1162.96270115 \tag{8.11}$$

$$\text{Logarithmic Fit: } y = 480.19 + 638.42 \cdot \log(x) \tag{8.12}$$

$$\text{Exponential Fit: } y = 1335.41141767 \cdot e^{0.01429924 \cdot x} \tag{8.13}$$

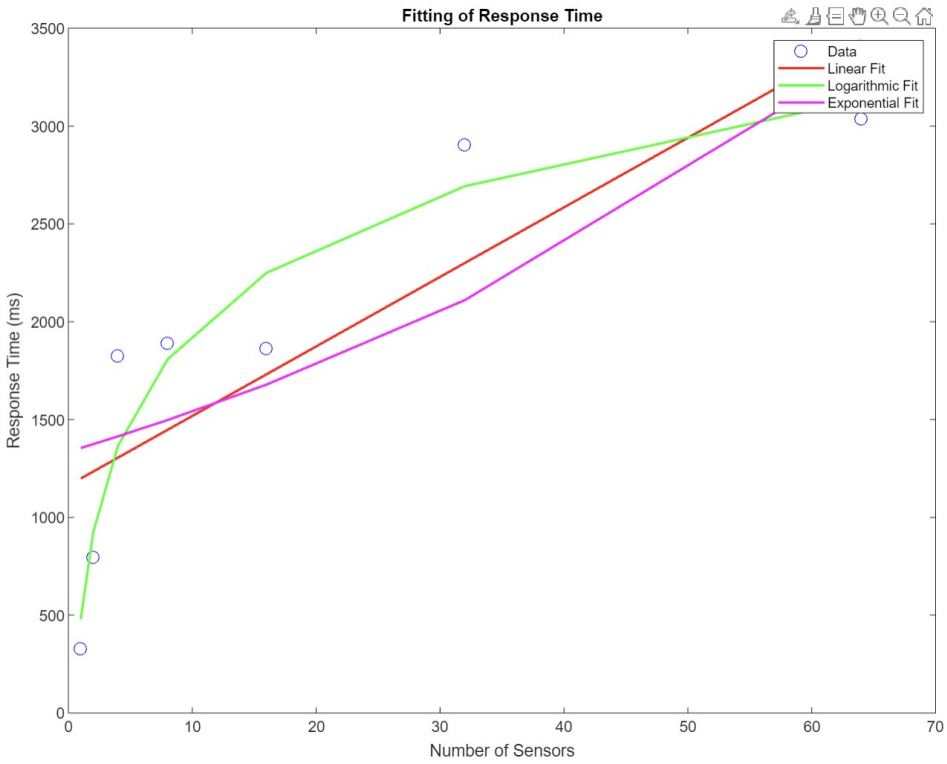

Figure 8.24: Plot for Average Response Time Fittings

To assess the quality of these fits, several metrics were calculated, including the coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC).

Table 8.2: Average Response Time Evaluation Metrics

Fitting	R-Squared	RMSE	AIC	BIC
Linear	0.67073	528.8014	93.7886	93.6263
Logarithmic	0.92234	256.8074	83.6766	83.5143
Exponential	0.58548	593.3237	95.4004	95.2381

The logarithmic fit demonstrated the highest R-squared value (0.92234), indicating a better fit compared to the linear and exponential fits. Additionally, the logarithmic fit had the lowest RMSE (256.8074), suggesting a smaller average deviation from the actual data points. Moreover, the logarithmic fit had the lowest AIC (83.6766) and BIC (83.5143) values, indicating a better fit in terms of information criteria.

Considering these evaluations, the logarithmic fit is deemed the most suitable model for representing the relationship between the number of sensors and the average response time in the developed system. This logarithmic equation provides a reliable estimation of the average response time based on the number of sensors.

The advantages of the logarithmic fitting in terms of scalability are noteworthy. The logarithmic equation suggests that as the number of sensors increases, the average response time experiences diminishing returns. This indicates that adding more sensors to the system has a diminishing impact on the response time improvement. Such a characteristic is favorable for scalability as it implies that the system can handle a growing number of sensors with a relatively stable average response time.

However, it is important to note that while the logarithmic fit provides valuable insights into the scalability of the system, the complexities introduced by the combination of edge instances and centralized instances should be considered, which may affect the accuracy of the measured average response time. Nonetheless, the logarithmic fit confirms the system's scalability, indicating that the average response time improves initially with the addition of sensors but at a decreasing rate. This understanding enables organizations to make informed decisions regarding resource allocation, capacity planning, and system optimization to effectively accommodate future growth.

In summary, the logarithmic fit obtained through MATLAB serves as compelling evidence of the system's scalability and its ability to handle increasing workloads. The logarithmic equation, with its diminishing returns, provides a reliable estimation of the average response time and demonstrates the system's capacity to efficiently accommodate a growing number of sensors.

Real-time Evaluation

To assess the real-time properties of the system, the maximum response time for each number of sensors was considered. Real-time requirements dictate that the response time should not exceed a certain threshold, ensuring that data is available

for the computation of aggregates every 30 minutes and that new data is received at least 2 minutes before the computation.

The selected threshold for the response time is of 10 seconds: it was determined to ensure that the system can process and save the data within the specified timeframe. This choice aligns with the requirement of having the previous data ready before the arrival of new data every two minutes. By adhering to this threshold, the system consistently handles incoming data in a timely manner.

The real-time nature of the application necessitates prompt data processing and saving. Setting a threshold of 10 seconds allows for efficient turnaround, ensuring the availability of data for analysis and subsequent processing without undue delays.

The decision to opt for a relatively low threshold provides a safety margin to account for potential variations or spikes in processing time. This ensures that even during periods of high workload or peak activity, the response time remains within an acceptable range.

Maintaining a response time below 10 seconds facilitates the desired level of responsiveness, allowing quick access to processed data. This enables informed decision-making and timely actions based on the latest information.

The following table displays the maximum response time (in milliseconds) for each number of sensors:

Table 8.3: Maximum Response Time for Different Numbers of Sensors

Number of Sensors	**Maximum Response Time (ms)**
1	623.00
2	1557.00
4	4487.00
8	4003.00
16	3681.00
32	7070.00
64	5007.00

Analyzing the maximum response time, it is possible to evaluate the real-time properties of the system:

- For one sensor, the maximum response time is **623.00** milliseconds, well below the threshold. This indicates that the system can process data from a

single sensor in real-time, ensuring timely computation of aggregates every 30 minutes.

- As the number of sensors increases, the maximum response time also increases. However, even for 64 sensors, the maximum response time is **5007.00** milliseconds, which is still within an acceptable range considering the real-time requirements.

Based on the maximum response times observed, it can be concluded that the system maintains its real-time properties, allowing for the computation of aggregates every 30 minutes. Additionally, the data is received with a sufficient time buffer of 2 minutes before the computation, ensuring the availability of the latest data.

8.5.2 CPU Utilization Analysis

The CPU utilization analysis focuses on evaluating how the system's CPU usage scales with the increasing number of sensors. The CPU utilization data was collected for each message rate, providing insights into the system's ability to handle the computational workload. The values are collected at a 30-second interval over a 1-minute window.

To provide a visual representation of the average CPU utilization at steady state for each number of sensors, the following images display the graphs for the centralized instance:

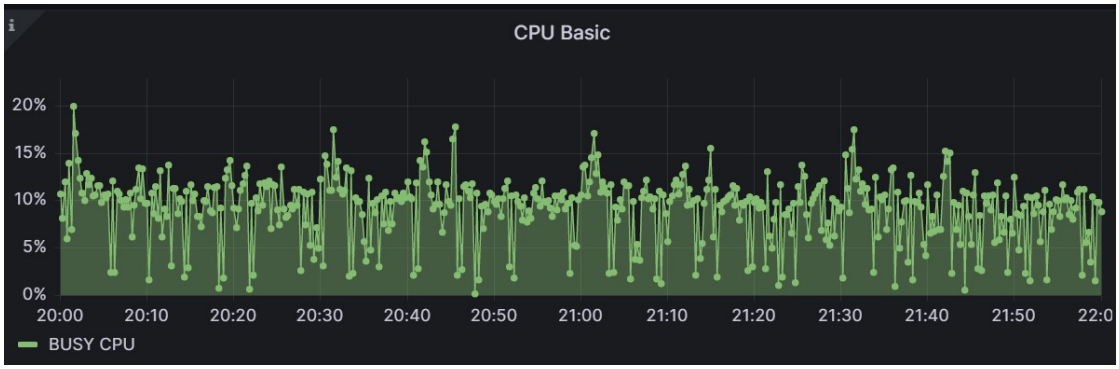

Figure 8.25: Average Centralized CPU Utilization for 1 Sensor

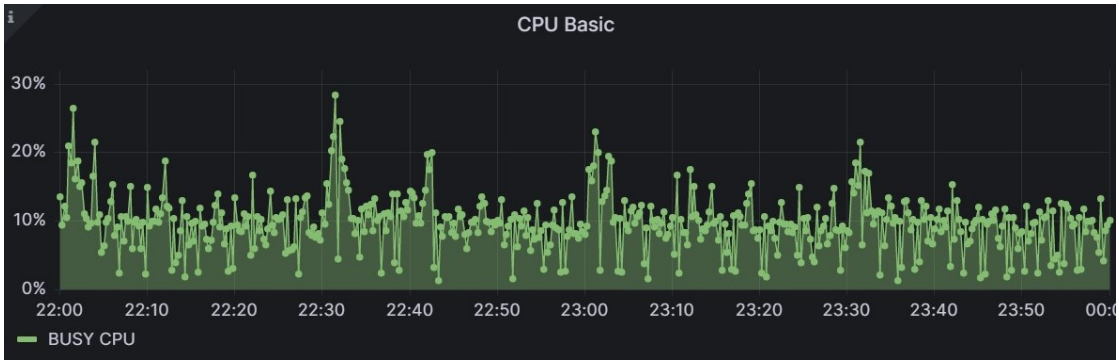

Figure 8.26: Average Centralized CPU Utilization for 2 Sensors

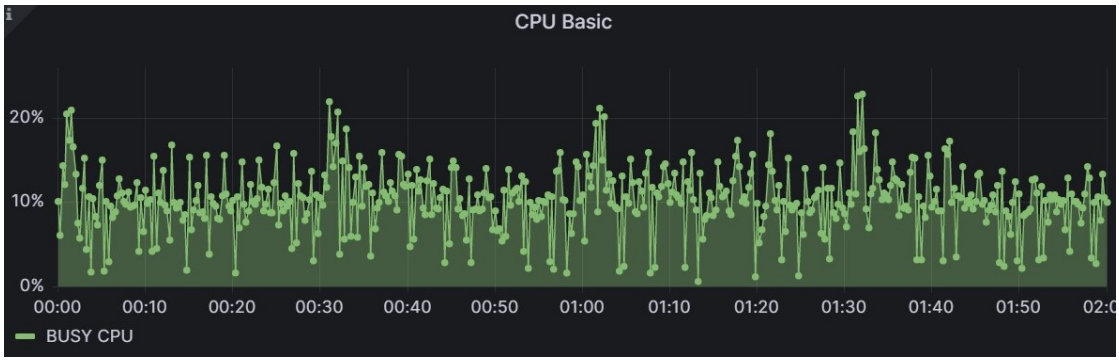

Figure 8.27: Average Centralized CPU Utilization for 4 Sensors

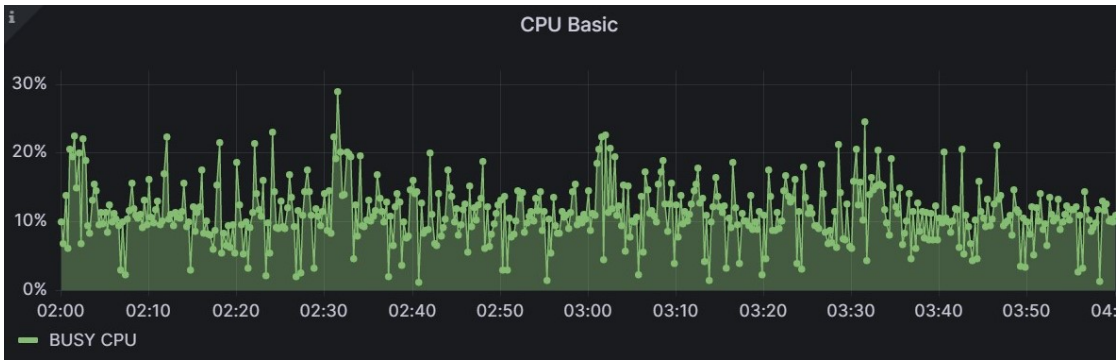

Figure 8.28: Average Centralized CPU Utilization for 8 Sensors

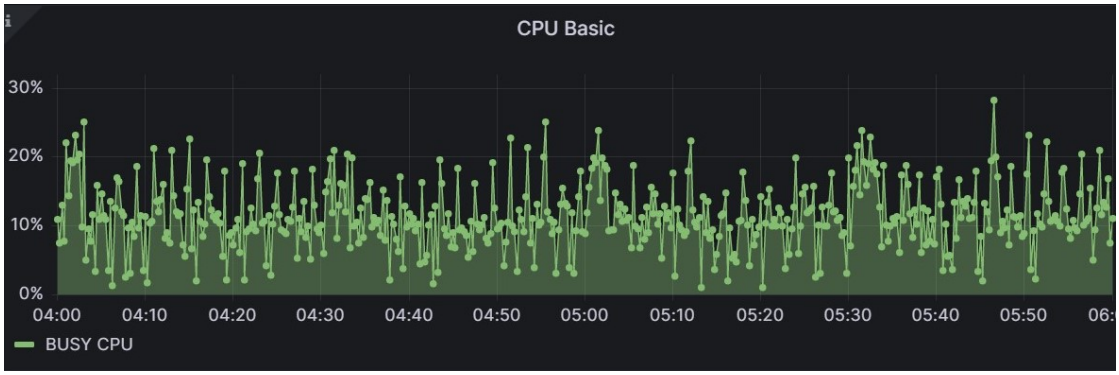

Figure 8.29: Average Centralized CPU Utilization for 16 Sensors

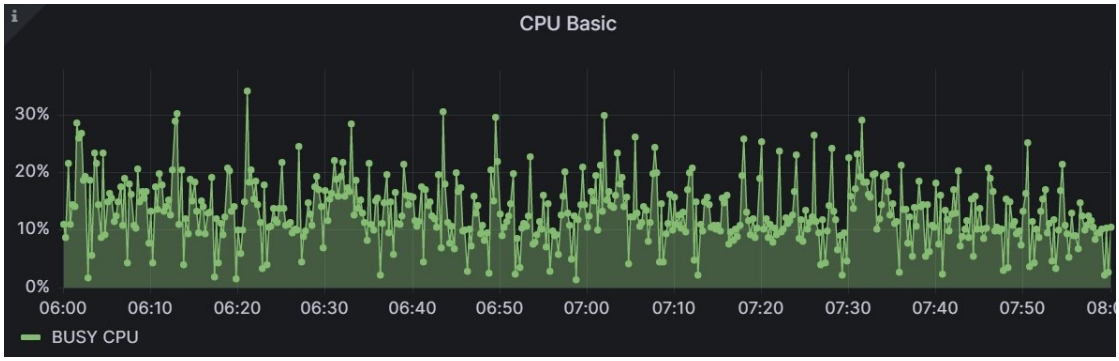

Figure 8.30: Average Centralized CPU Utilization for 32 Sensors

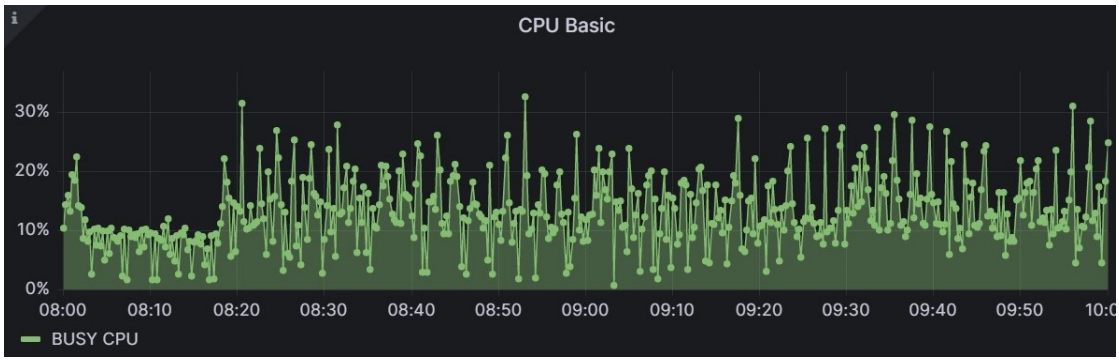

Figure 8.31: Average Centralized CPU Utilization for 64 Sensors

To provide a visual representation of the average CPU utilization at steady state for each number of sensors, the following images display the graphs for the edge instance:

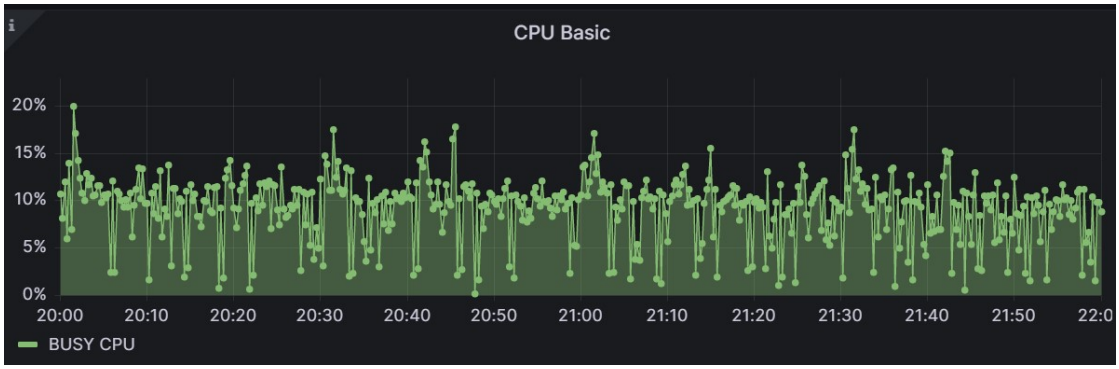

Figure 8.32: Average Edge CPU Utilization for 1 Sensor

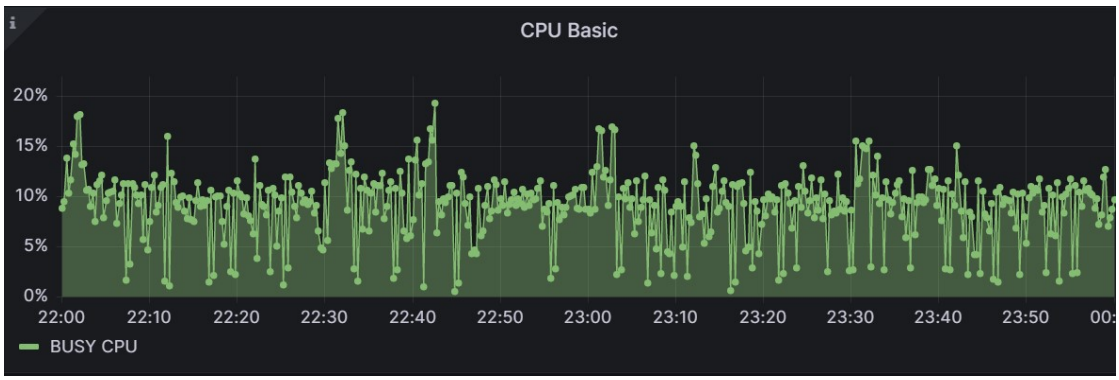

Figure 8.33: Average Edge CPU Utilization for 2 Sensors

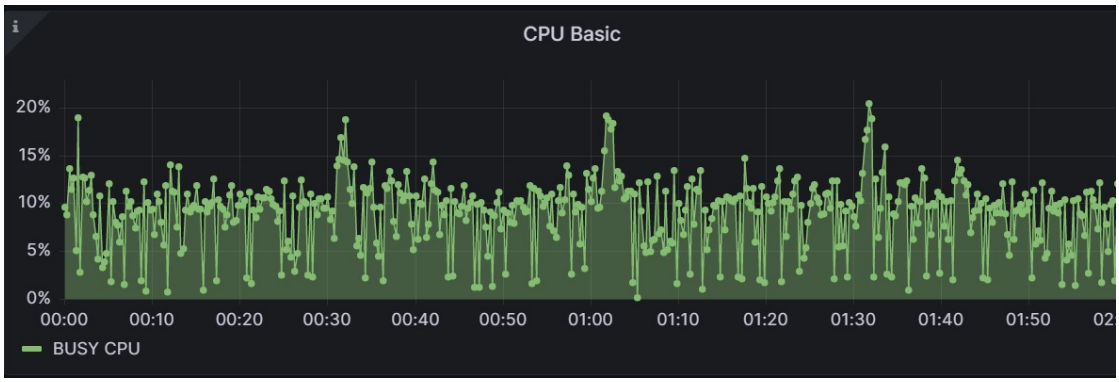

Figure 8.34: Average Edge CPU Utilization for 4 Sensors

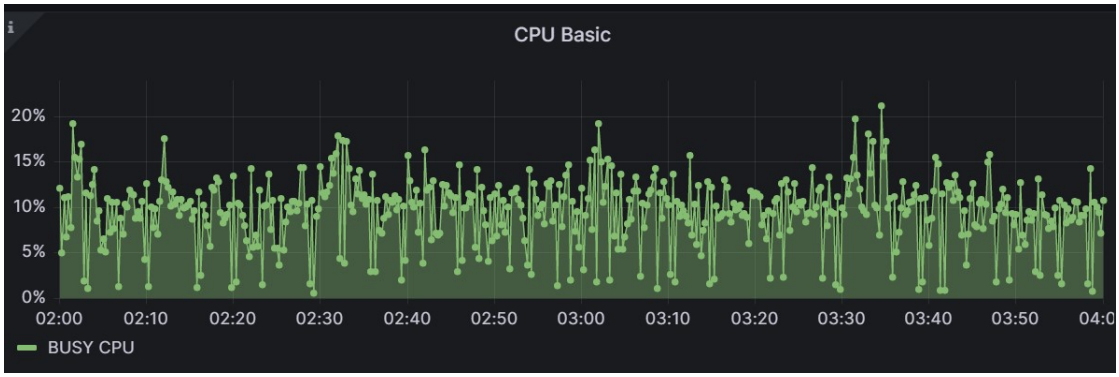

Figure 8.35: Average Edge CPU Utilization for 8 Sensors

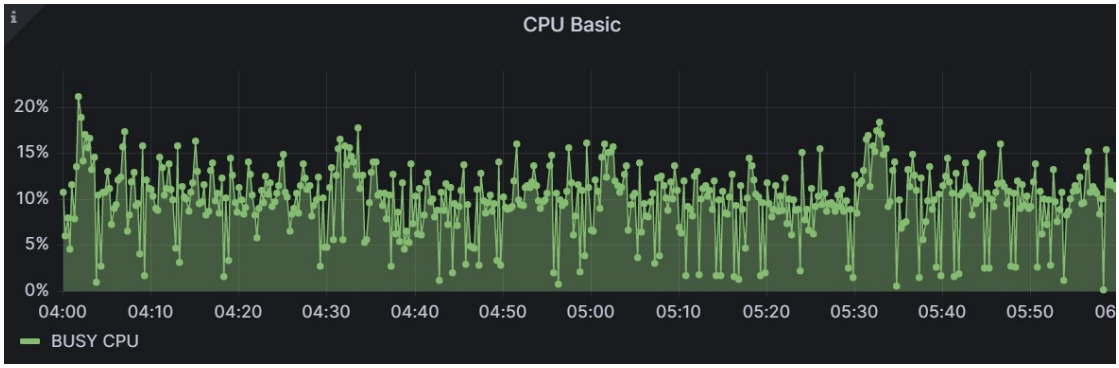

Figure 8.36: Average Edge CPU Utilization for 16 Sensors

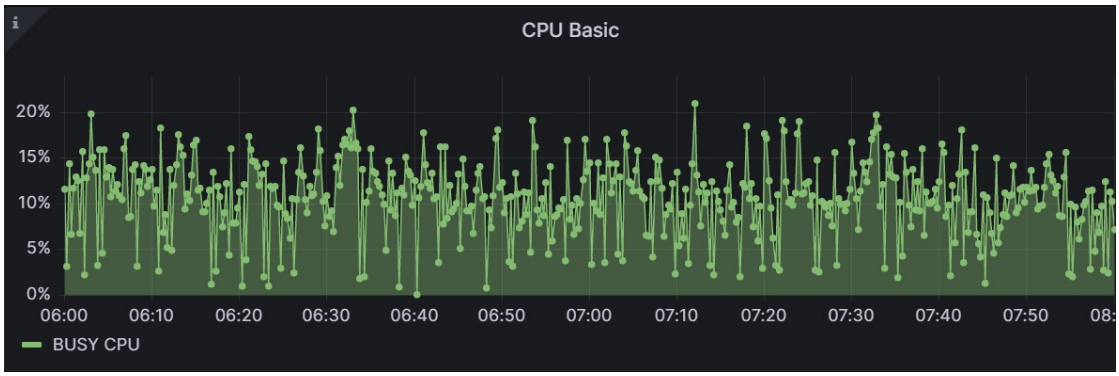

Figure 8.37: Average Edge CPU Utilization for 32 Sensors

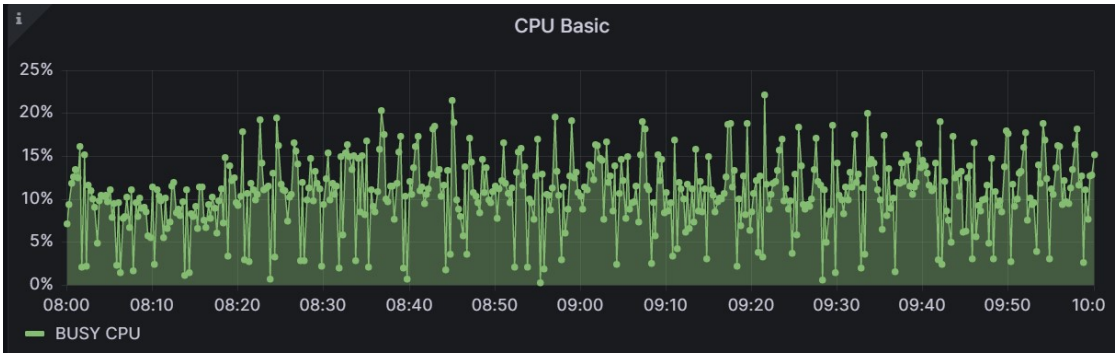

Figure 8.38: Average Edge CPU Utilization for 64 Sensors

The CPU utilization were computed as the average value for each tried number of sensors, providing a comprehensive view of the system's performance. It's important to note that in some regions, the average CPU utilization slightly higher. This can be attributed to the execution of additional OpenWhisk actions, specifically the actions to compute aggregates. They are triggered every half an hour to process and aggregate the collected sensor data. During their execution, they may temporarily increase the overall CPU utilization of the edge and centralized instance. However, it is important to consider that these periodic actions have a minimal impact on the overall scalability and efficiency of the system.

The following table displays the average CPU utilization (in percentage) for each message rate:

Table 8.4: Average CPU Utilization for Different Numbers of Sensors

Number of Sensors	Average Centralized CPU Utilization (%)	Average Edge CPU Utilization (%)
1	11.00	10.49
2	11.00	10.25
4	11.66	10.71
8	12.50	11.48
16	12.64	10.86
32	15.49	12.66
64	16.39	13.14

Analysis of Average CPU Utilizarion Trends

CENTRALIZED INSTANCE

To gain insights into the relationship between the number of sensors and the average CPU utilization, various fitting techniques have been performed using MATLAB. The aim was to find the best mathematical model that accurately represents the observed trend. Three different fitting approaches have been considered: linear, logarithmic, and exponential.

The results of these fittings indicate the following relationships between the number of sensors (x) and the average CPU utilization (y) according to each fitting model:

$$\text{Linear Fit: } y = 0.09 \cdot x + 11.34 \tag{8.14}$$

$$\text{Logarithmic Fit: } y = 10.15 + 1.35 \cdot \log(x) \tag{8.15}$$

$$\text{Exponential Fit: } y = 11.49 \cdot e^{0.01 \cdot x} \tag{8.16}$$

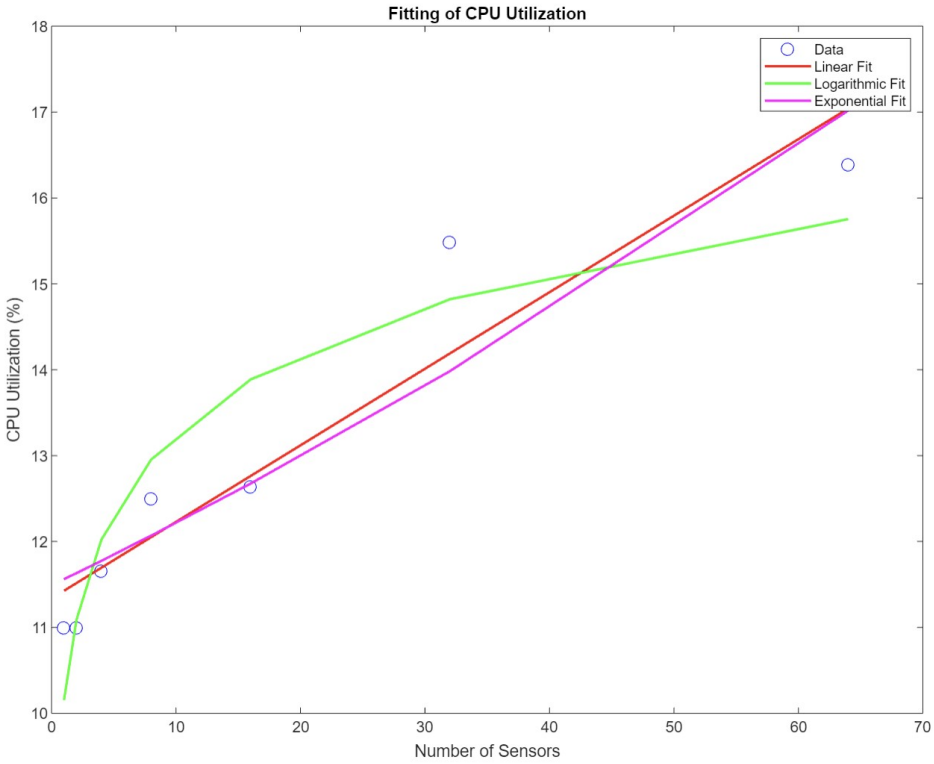

Figure 8.39: Plot for Average Centralized CPU Utilization Fittings

To evaluate the quality of these fits, several metrics were calculated, including the coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC).

Table 8.5: Fitting Results for Centralized CPU Utilization

Fitting	R-Squared	RMSE	AIC	BIC
Linear	0.90011	0.63044	-0.45863	-0.6209
Logarithmic	0.87549	0.70385	1.0834	0.92117
Exponential	0.87151	0.71503	1.3039	1.1417

The linear fit demonstrated the highest R-squared value (0.90011), indicating a better fit compared to the logarithmic and exponential fits. Additionally, the linear fit had the lowest RMSE (0.63044), suggesting a smaller average deviation from the actual data points. Moreover, the linear fit had the lowest AIC (-0.45863) and BIC (-0.6209) values, indicating a better fit in terms of information criteria.

Considering these evaluations, the linear fit is deemed the most suitable model for representing the relationship between the number of sensors and the average CPU utilization in the developed system. This linear equation provides a reliable estimation of the average CPU utilization based on the number of sensors.

The advantages of the linear fitting in terms of scalability are noteworthy. The small coefficient of 0.09 in the linear equation indicates that for every additional sensor added to the system, the average CPU utilization increases by only 0.09%. This suggests a linear relationship with a relatively low rate of resource consumption growth. Such a characteristic is favorable for scalability as it implies that the system can handle an expanding sensor network without experiencing a significant surge in resource demands.

However, it is important to note that while the linear fit provides valuable insights into the scalability of the system, it also important to consider the complexities introduced by the combination of edge instances and centralized instances, which may affect the accuracy of the measured CPU utilization. Nonetheless, the linear fit confirms the system's scalability, indicating that the average CPU utilization increases proportionally with the number of sensors. This understanding enables organizations to make informed decisions regarding resource allocation, capacity planning, and system optimization to effectively accommodate future growth.

In summary, the linear fit obtained through MATLAB serves as compelling evidence of the system's scalability and its ability to handle increasing workloads. The linear

equation, with its small coefficient, provides a reliable estimation of the average CPU utilization and demonstrates the system's capacity to efficiently accommodate a growing number of sensors.

EDGE INSTANCE

To gain insights into the relationship between the number of sensors and the average CPU utilization, various fitting techniques were performed using MATLAB. The aim was to find the best mathematical model that accurately represents the observed trend. Three different fitting approaches have been considered: linear, logarithmic, and exponential.

The results of these fittings indicate the following relationships between the number of sensors (x) and the average CPU utilization (y) according to each fitting model:

$$\text{Linear Fit: } y = 0.04 \cdot x + 10.56 \tag{8.17}$$

$$\text{Logarithmic Fit: } y = 9.99 + 0.67 \cdot \log(x) \tag{8.18}$$

$$\text{Exponential Fit: } y = 10.67 \cdot e^{0.001 \cdot x} \tag{8.19}$$

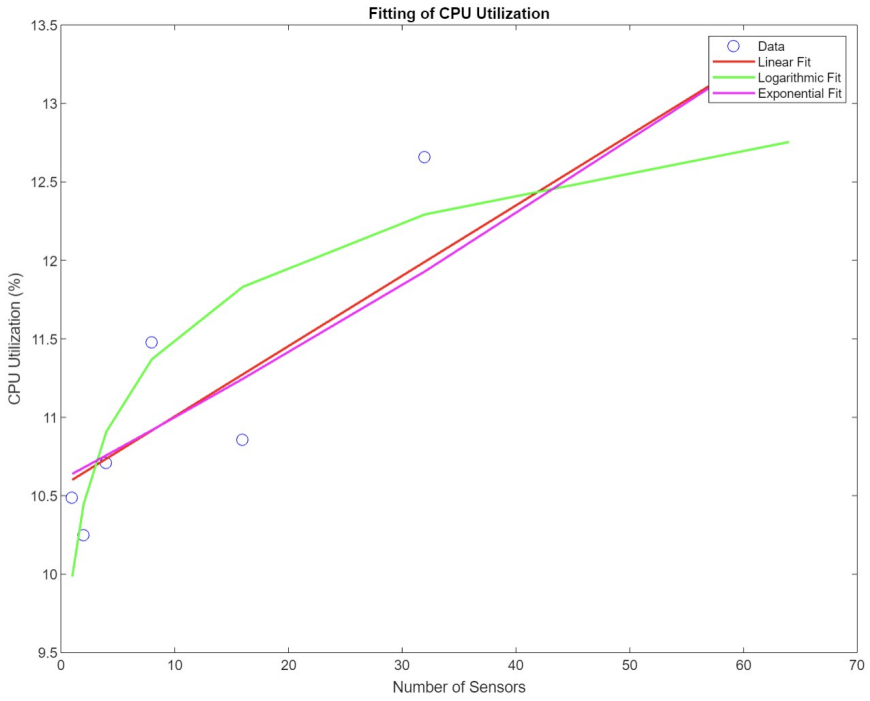

Figure 8.40: Plot for Average Edge CPU Utilization Fittings

To evaluate the quality of these fits, several metrics were calculated, including the coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC).

Table 8.6: Fitting Results for Edge CPU Utilization

Fitting	R-Squared	RMSE	AIC	BIC
Linear	0.84213	0.4122	-6.4076	-6.5698
Logarithmic	0.79134	0.47388	-4.4552	-4.6175
Exponential	0.82905	0.42893	-5.8503	-6.0126

The linear fit demonstrates a relatively high R-squared value (0.84213), indicating a good fit compared to the logarithmic and exponential fits. The RMSE for the linear fit is the lowest among the three models (0.4122), suggesting a smaller average deviation from the actual data points. Moreover, the linear fit has the lowest AIC (-6.4076) and BIC (-6.5698) values, indicating a better fit in terms of information criteria.

Based on these evaluations, the linear fit is considered the most suitable model for representing the relationship between the number of sensors and the average CPU utilization in the system. The linear equation provides a reliable estimation of the average CPU utilization based on the number of sensors.

The advantages of the linear fitting in terms of scalability are noteworthy. The small coefficient of 0.04 in the linear equation indicates that for every additional sensor added to the system, the average CPU utilization increases by only 0.04%. This suggests a linear relationship with a relatively low rate of resource consumption growth. Such a characteristic is favorable for scalability as it implies that the system can handle an expanding sensor network without experiencing a significant surge in resource demands.

However, it is important to note that while the linear fit provides valuable insights into the scalability of the system, the complexities introduced by the combination of edge instances and centralized instances should also be considered, which may affect the accuracy of the measured CPU utilization. Nonetheless, the linear fit confirms the system's scalability, indicating that the average CPU utilization increases proportionally with the number of sensors. This understanding enables organizations to make informed decisions regarding resource allocation, capacity planning, and system optimization to effectively accommodate future growth.

In summary, the linear fit obtained through MATLAB serves as compelling evidence

of the system's scalability and its ability to handle increasing workloads. The linear equation, with its small coefficient, provides a reliable estimation of the average CPU utilization and demonstrates the system's capacity to efficiently accommodate a growing number of sensors.

RAM Utilization Analysis

The RAM utilization analysis focuses on evaluating how the system's RAM usage scales with the increasing number of sensors. The RAM utilization data was collected for each message rate, providing insights into the system's ability to handle the computational workload. The values are collected at a 30-second interval.

To provide a visual representation of the average RAM utilization at steady state for each number of sensors, the following images display the graphs for the centralized instance:

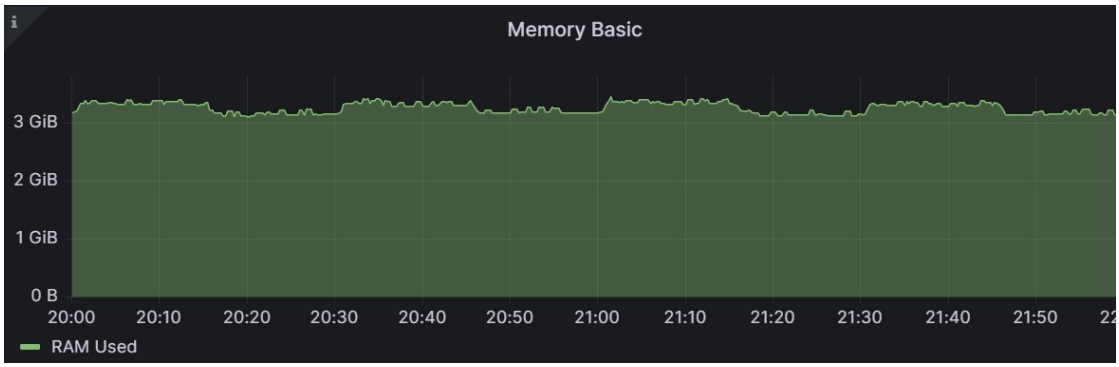

Figure 8.41: Average Centralized RAM Utilization for 1 Sensor

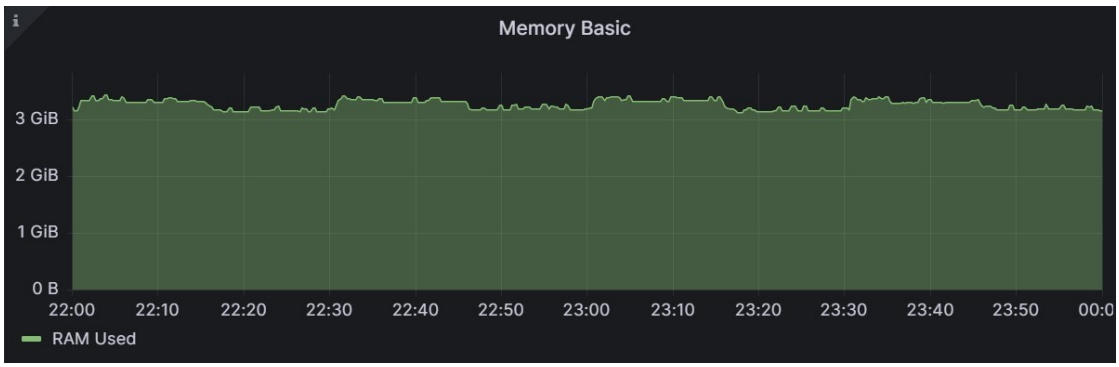

Figure 8.42: Average Centralized RAM Utilization for 2 Sensors

Figure 8.43: Average Centralized RAM Utilization for 4 Sensors

Figure 8.44: Average Centralized RAM Utilization for 8 Sensors

Figure 8.45: Average Centralized RAM Utilization for 16 Sensors

Figure 8.46: Average Centralized RAM Utilization for 32 Sensors

Figure 8.47: Average Centralized RAM Utilization for 64 Sensors

To provide a visual representation of the average RAM utilization at steady state for each number of sensors, the following images display the graphs for the edge instance:

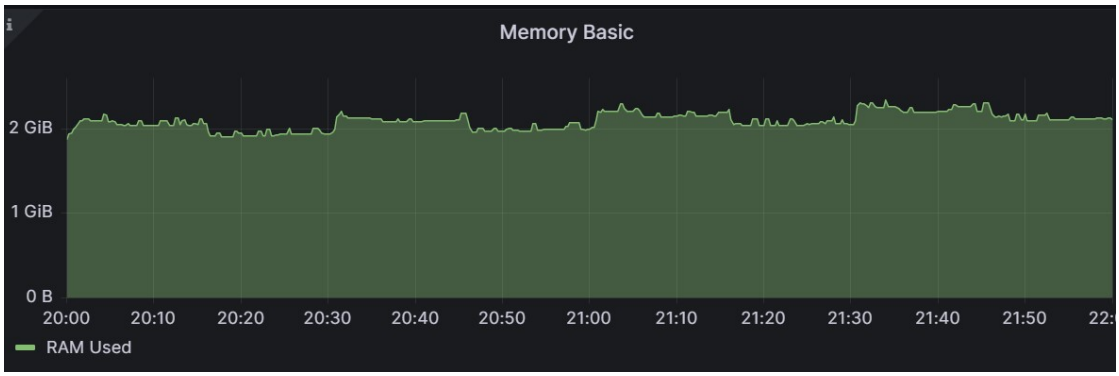

Figure 8.48: Average Edge RAM Utilization for 1 Sensor

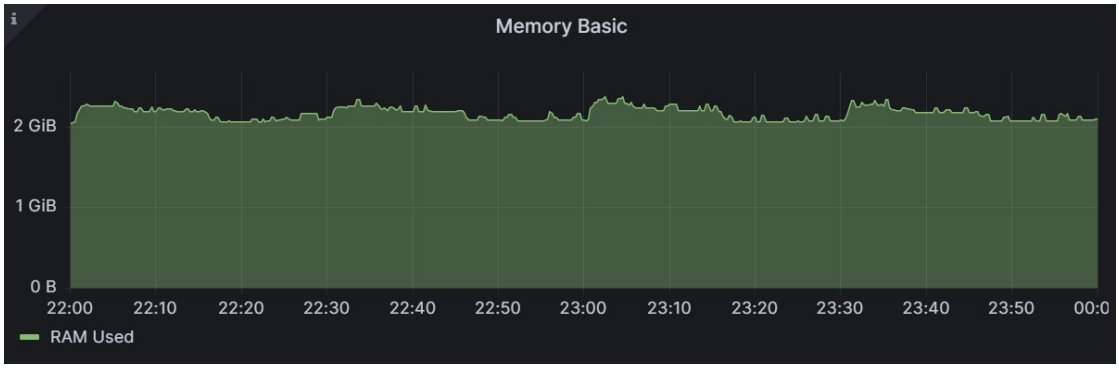

Figure 8.49: Average Edge RAM Utilization for 2 Sensors

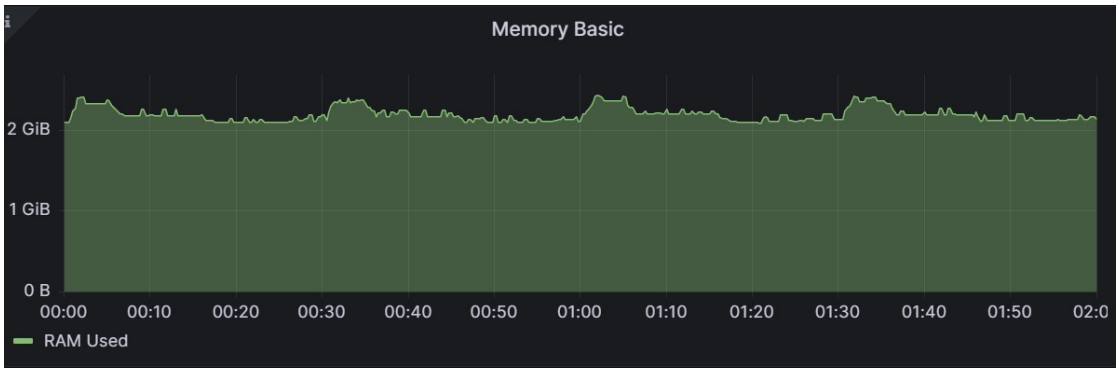

Figure 8.50: Average Edge RAM Utilization for 4 Sensors

Figure 8.51: Average Edge RAM Utilization for 8 Sensors

Figure 8.52: Average Edge RAM Utilization for 16 Sensors

Figure 8.53: Average Edge RAM Utilization for 32 Sensors

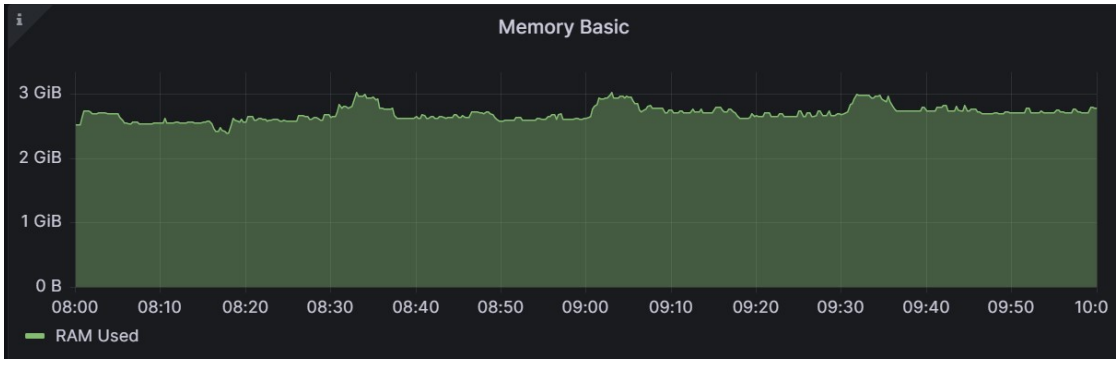

Figure 8.54: Average Edge RAM Utilization for 64 Sensors

The RAM utilizations were computed as the average value for each tried number of sensors, providing a comprehensive view of the system's performance. It's important to note that in some regions, the average RAM utilization slightly

higher. This can be attributed to the execution of additional OpenWhisk actions, specifically the actions to compute aggregates. They are triggered every half an hour to process and aggregate the collected sensor data. During their execution, they may temporarily increase the overall RAM utilization of the edge and centralized instance. However, it is important to consider that these periodic actions have a minimal impact on the overall scalability and efficiency of the system.

The following table displays the average RAM utilization (in GiB) for each message rate:

Table 8.7: Average RAM Utilization for Different Numbers of Sensors

Number of Sensors	Average Centralized RAM Utilization (GiB)	Average Edge RAM Utilization (GiB)
1	3.26	2.06
2	3.27	2.18
4	3.25	2.22
8	3.27	2.28
16	3.30	2.35
32	3.34	2.54
64	3.35	2.74

Analysis of Average RAM Utilization Trends

CENTRALIZED INSTANCE
To gain insights into the relationship between the number of sensors and the average RAM utilization, various fitting techniques have been performed using MATLAB. The objective was to identify the best mathematical model that accurately represents the observed trend. Three different fitting approaches have been explored: linear, logarithmic, and exponential.

The results of these fittings indicate the following relationships between the number of sensors (x) and the average RAM utilization (y) according to each fitting model:

$$\text{Linear Fit: } y = 0.0015929043 \cdot x + 3.2625287356 \tag{8.20}$$

$$\text{Logarithmic Fit: } y = 3.24 + 0.02 \cdot \log(x) \tag{8.21}$$

$$\text{Exponential Fit: } y = 3.2627531228 \cdot e^{0.0004794407 \cdot x} \tag{8.22}$$

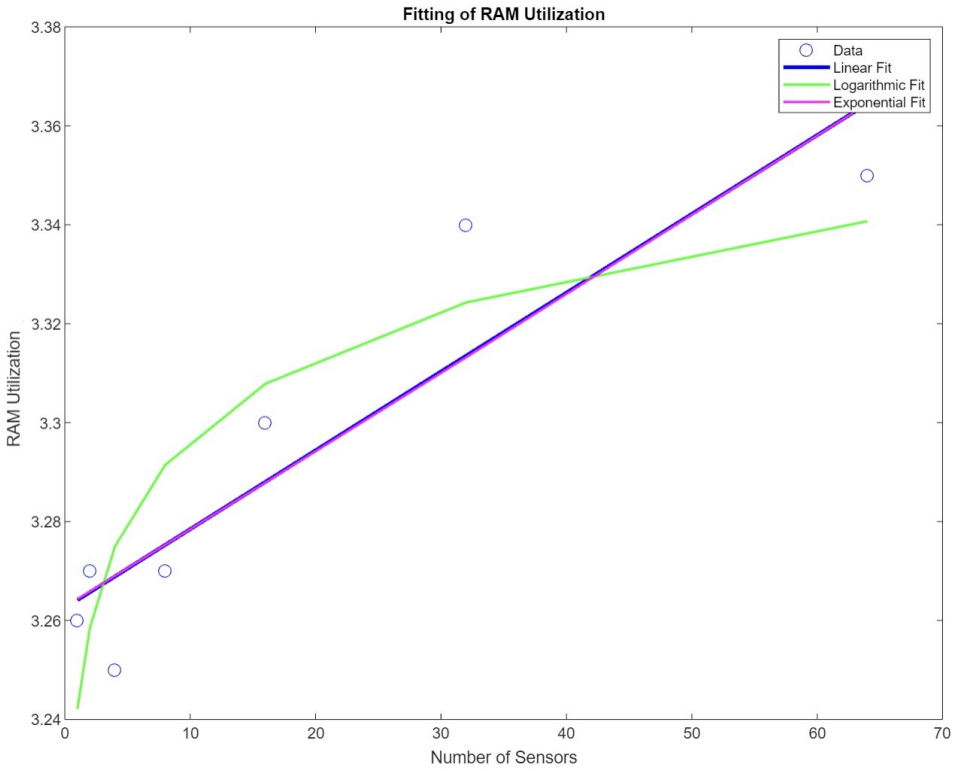

Figure 8.55: Plot for Average Centralized RAM Utilization Fittings

To assess the quality of these fits, several metrics were calculated, including the coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC).

Table 8.8: Fitting Results for Centralized RAM Utilization

Fitting	R-Squared	RMSE	AIC	BIC
Linear	0.84443	0.014519	-53.2518	-53.4141
Logarithmic	0.79669	0.016599	-51.3782	-51.5405
Exponential	0.84202	0.014631	-53.1444	-53.3066

The linear fit demonstrated the highest R-squared value (0.84443), indicating a better fit compared to the logarithmic and exponential fits. Additionally, the linear fit had the lowest RMSE (0.014519), suggesting a smaller average deviation from the actual data points. Moreover, the linear fit had the lowest AIC (-53.2518) and BIC (-53.4141) values, indicating a better fit in terms of information criteria.

Considering these evaluations, the linear fit is considered the most suitable model for representing the relationship between the number of sensors and the average RAM utilization in the developed system. The linear equation provides a reliable estimation of the average RAM utilization based on the number of sensors.

The advantages of the linear fitting in terms of scalability are noteworthy. The small coefficient of 0.0015929043 in the linear equation indicates that for every additional sensor added to the system, the average RAM utilization increases by only 0.0015929043 units. This suggests a linear relationship with a relatively low rate of resource consumption growth. Such a characteristic is favorable for scalability as it implies that the system can handle an expanding sensor network without experiencing a significant surge in RAM demands.

However, it is important to note that while the linear fit provides valuable insights into the scalability of the system, the complexities introduced by the combination of edge instances and centralized instances should also be considered, which may affect the accuracy of the measured RAM utilization. Nonetheless, the linear fit confirms the system's scalability, indicating that the average RAM utilization increases proportionally with the number of sensors. This understanding enables organizations to make informed decisions regarding resource allocation, capacity planning, and system optimization to effectively accommodate future growth.

In summary, the linear fit obtained through MATLAB serves as compelling evidence of the system's scalability and its ability to handle increasing workloads. The linear equation, with its small coefficient, provides a reliable estimation of the average RAM utilization and demonstrates the system's capacity to efficiently accommodate a growing number of sensors.

EDGE INSTANCE

To gain insights into the relationship between the number of sensors and the average RAM utilization, various fitting techniques have been performed using MATLAB. The objective was to identify the best mathematical model that accurately represents the observed trend. Three different fitting approaches have been explored: linear, logarithmic, and exponential.

The results of these fittings indicate the following relationships between the number of sensors (x) and the average RAM utilization (y) according to each fitting model:

$$\text{Linear Fit: } y = 0.0097506562 \cdot x + 2.1616666667 \tag{8.23}$$

$$\text{Logarithmic Fit: } y = 2.03 + 0.15 \cdot \log(x) \tag{8.24}$$

$$\text{Exponential Fit: } y = 2.1713004038 \cdot e^{0.0038981277 \cdot x} \tag{8.25}$$

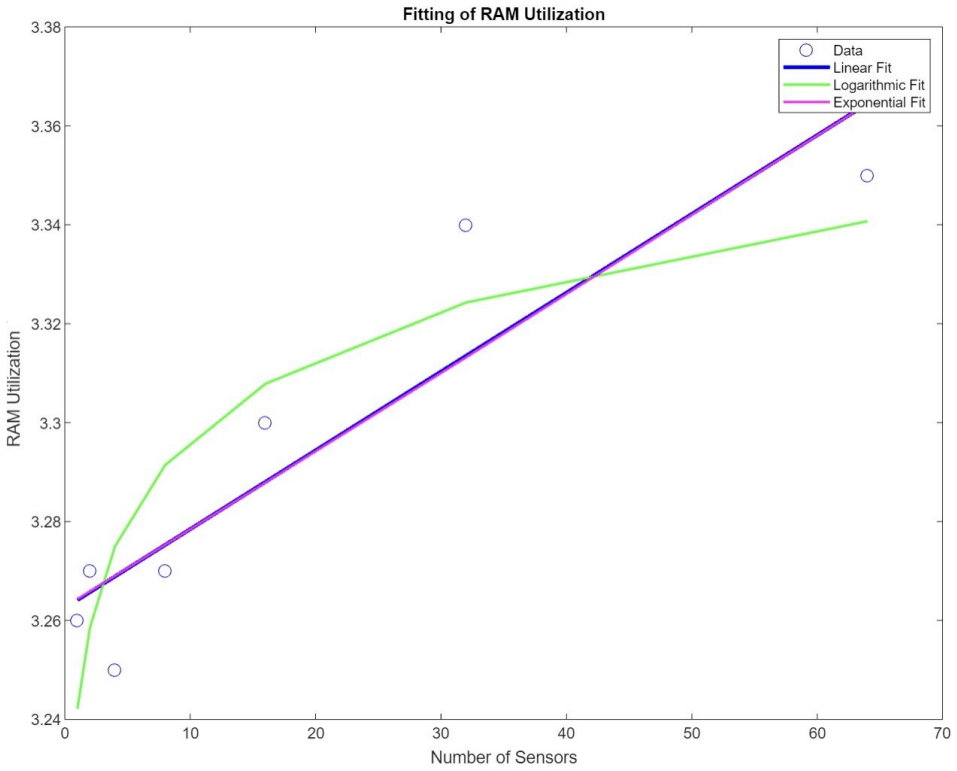

Figure 8.56: Plot for Average Edge RAM Utilization Fittings

To assess the quality of these fits, several metrics were calculated, including the coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC).

Table 8.9: Fitting Results for Edge RAM Utilization

Fitting	R-Squared	RMSE	AIC	BIC
Linear	0.93186	0.055993	-34.3554	-34.5176
Logarithmic	0.92612	0.058305	-33.7889	-33.9512
Exponential	0.9182	0.061349	-33.0766	-33.2388

The linear fit demonstrates the highest R-squared value (0.93186), indicating a better fit compared to the logarithmic and exponential fits. Additionally, the linear fit has the lowest RMSE (0.055993), suggesting a smaller average deviation from the actual data points. Moreover, the linear fit has the lowest AIC (-34.3554) and BIC (-34.5176) values, indicating a better fit in terms of information criteria.

Considering these evaluations, the linear fit is deemed the most suitable model for representing the relationship between the number of sensors and the average RAM utilization in the developed system. This linear equation provides a reliable estimation of the average RAM utilization based on the number of sensors.

The advantages of the linear fitting in terms of scalability are noteworthy. The small coefficient of 0.0097506562 in the linear equation indicates that for every additional sensor added to the system, the average RAM utilization increases by only 0.0097506562 units. This suggests a linear relationship with a relatively low rate of resource consumption growth. Such a characteristic is favorable for scalability as it implies that the system can handle an expanding sensor network without experiencing a significant surge in RAM demands.

However, it is important to note that while the linear fit provides valuable insights into the scalability of the system, the complexities introduced by the combination of edge instances and centralized instances should also be considered, which may affect the accuracy of the measured RAM utilization. Nonetheless, the linear fit confirms the system's scalability, indicating that the average RAM utilization increases proportionally with the number of sensors. This understanding enables organizations to make informed decisions regarding resource allocation, capacity planning, and system optimization to effectively accommodate future growth.

In summary, the linear fit obtained through MATLAB serves as compelling evidence of the system's scalability and its ability to handle increasing workloads. The linear equation, with its small coefficient, provides a reliable estimation of the average RAM utilization and demonstrates the system's capacity to efficiently accommodate a growing number of sensors.

8.5.3 Network Throughput Analysis

The network throughput analysis focuses on evaluating how the network traffic between the edge and centralized instance scales with the increasing number of sensors. The network throughput data was collected for each message rate, providing insights into the system's ability to handle the increasing traffic. The values are collected at a 15-seconds interval.

To provide a visual representation of the average network throughput of each relevant interface that receives/sends data at steady state for each number of sensors, the following images display the graphs:

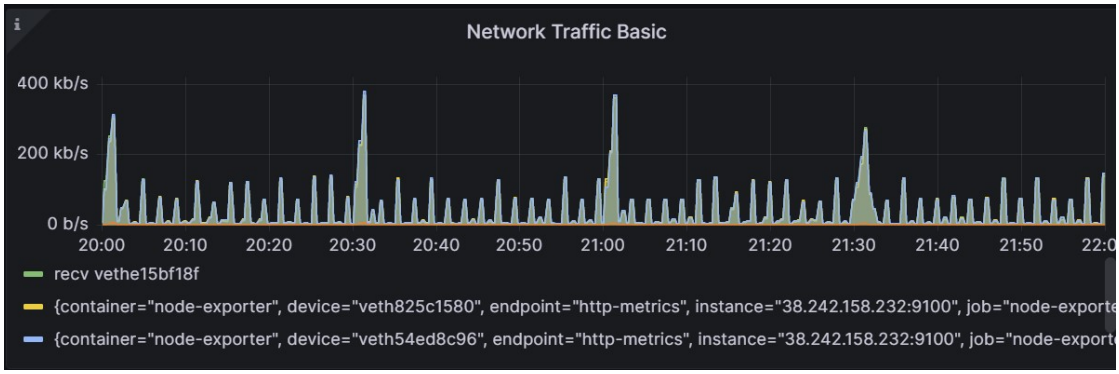

Figure 8.57: Average Network Throughput for 1 Sensor

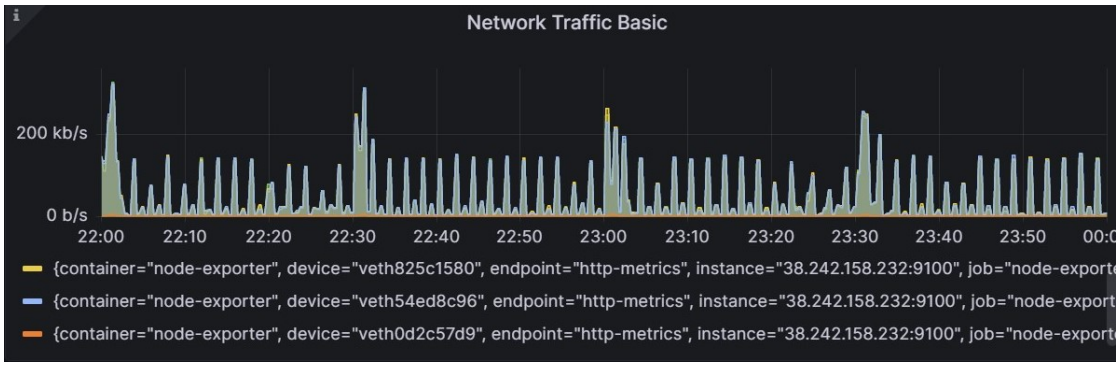

Figure 8.58: Average Network Throughput for 2 Sensors

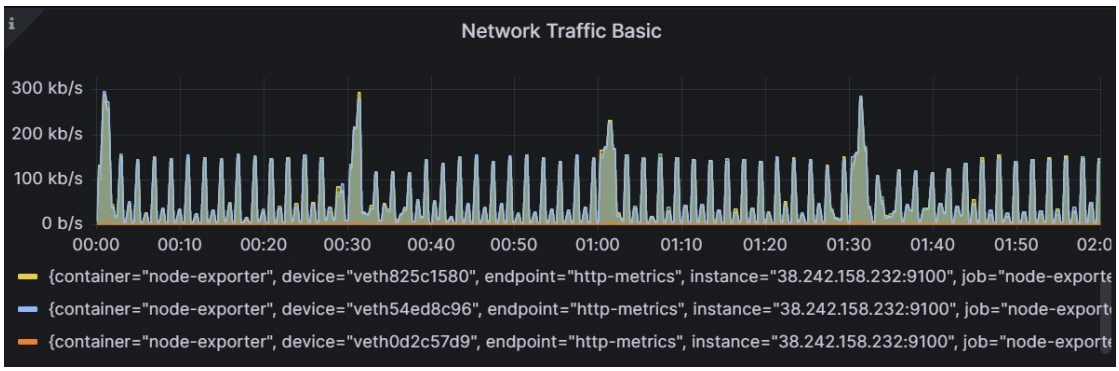

Figure 8.59: Average Network Throughput for 4 Sensors

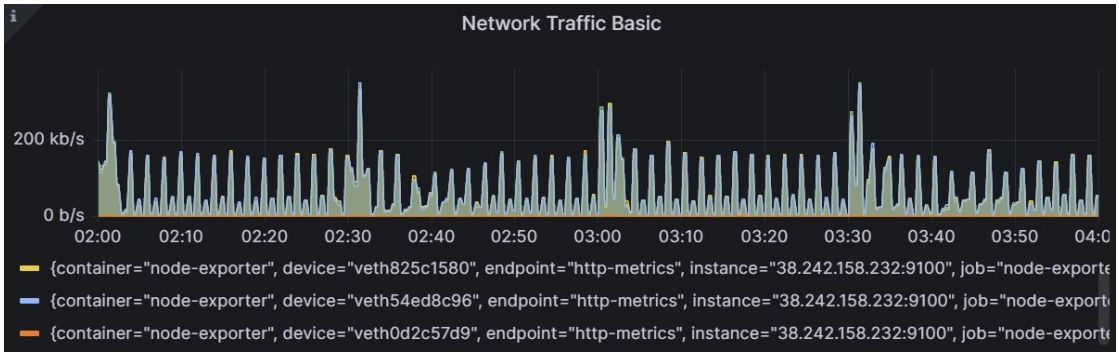

Figure 8.60: Average Network Throughput for 8 Sensors

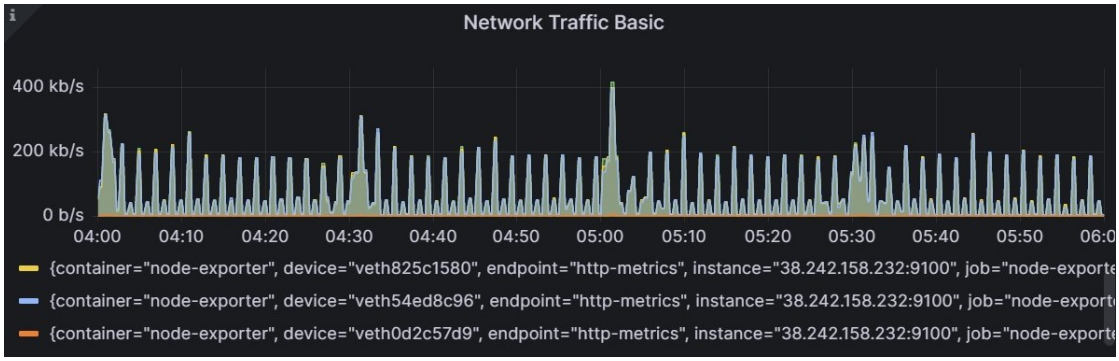

Figure 8.61: Average Network Throughput for 16 Sensors

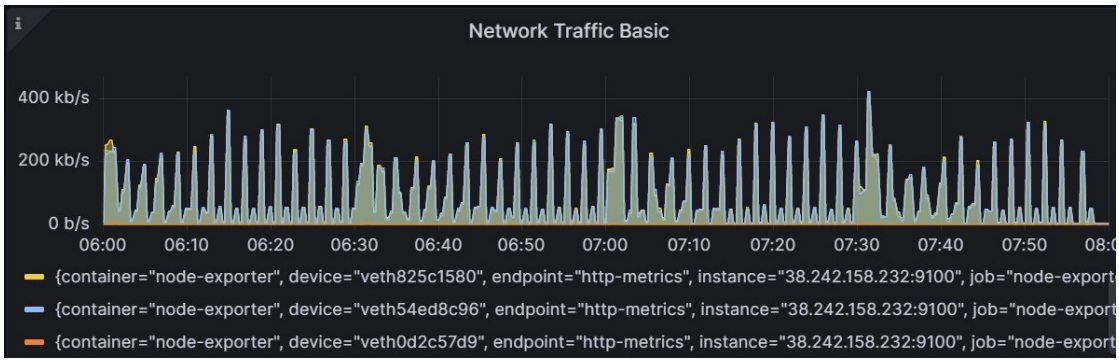

Figure 8.62: Average Network Throughput for 32 Sensors

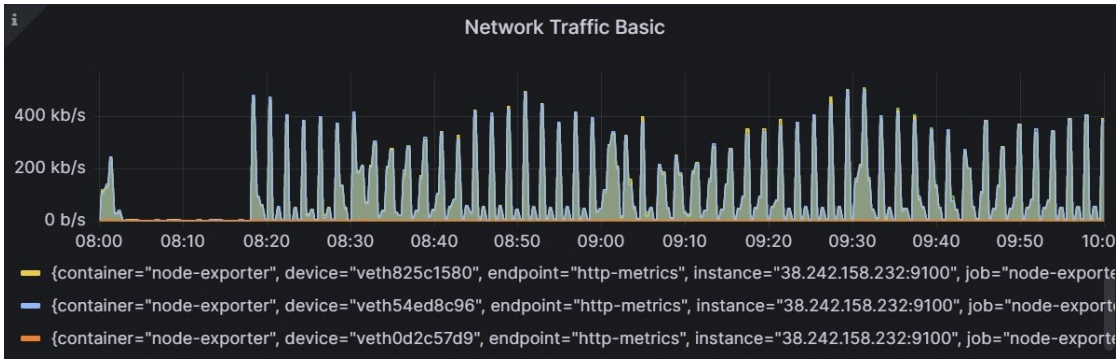

Figure 8.63: Average Network Throughput for 64 Sensors

The network throughput was calculated as the average value for different numbers of sensors, providing a comprehensive performance overview. In some regions, the average throughput is slightly higher due to additional OpenWhisk actions, mainly for computing aggregates and storing data in the centralized InfluxDB. These actions temporarily increase the network throughput but have minimal impact on overall scalability and efficiency. The increase in throughput can also be attributed to cache misses in edge instances, requiring data retrieval from the centralized CouchDB.

The following table displays the average network throughput (in kb/s) for each message rate. It is important to remember that this average network throughput has been obtained by summing the average network throughput of four different virtual interfaces, which are related to the centralized InfluxDB and CouchDB instances.

Table 8.10: Average Network Throughput for Different Numbers of Sensors

Number of Sensors	veth e15bf18f (kb/s)	veth 825c1580 (kb/s)	veth 54ed8c96 (kb/s)	veth 0d2c57d9 (kb/s)	Total (kb/s)
1	38.99988	40.457	40.46125	2.037	121.9551
2	49.46092	51.55988	51.22196	2.045875	154.2886
4	55.26138	56.92196	57.05838	2.064208	171.3059
8	64.02867	65.73738	65.96988	2.117833	197.8538
16	73.29508	74.87588	74.44017	2.092375	224.7035
32	96.01183	98.32721	97.85879	2.095958	294.2938
64	112.8953	114.7505	114.8362	2.072125	344.5541

Analysis of Average Overall Network Troughput Trends

To gain insights into the relationship between the number of sensors and the average overall network troughput between the centralized and edge instance, various fitting techniques have been performed using MATLAB. The objective was to identify the best mathematical model that accurately represents the observed trend. Three different fitting approaches have been explored: linear, logarithmic, and exponential.

The results of these fittings indicate the following relationships between the number of sensors (x) and the overall average network troughput (y) according to each fitting model:

$$\text{Linear Fit: } y = 3.28996672 \cdot x + 155.87557529 \tag{8.26}$$

$$\text{Logarithmic Fit: } y = 108.29 + 51.59 \cdot \log(x) \tag{8.27}$$

$$\text{Exponential Fit: } y = 167.15622250 \cdot e^{0.01215438 \cdot x} \tag{8.28}$$

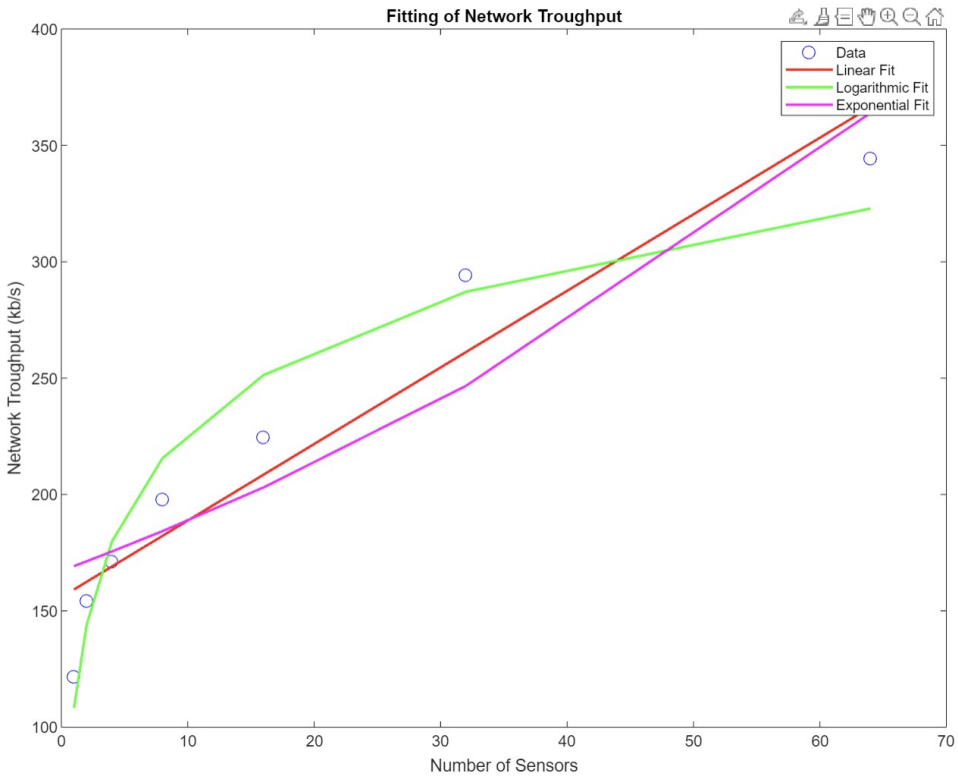

Figure 8.64: Plot for Average Network Throughput Fittings

To assess the quality of these fits, several metrics were calculated, including the coefficient of determination (R-squared), root mean squared error (RMSE), Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC).

Table 8.11: Fitting Results for Network Throughput

Fitting	R-Squared	RMSE	AIC	BIC
Linear	0.90611	22.4901	49.5831	49.4208
Logarithmic	0.94936	16.517	45.2615	45.0992
Exponential	0.84517	28.8804	53.0843	52.922

The logarithmic fit demonstrated the highest R-squared value (0.94936), indicating a better fit compared to the linear and exponential fits. Additionally, the linear fit had the lowest RMSE (16.517), suggesting a smaller average deviation from the actual data points. Moreover, the logerithmic fit had the lowest AIC (45.2615) and BIC (45.0992) values, indicating a better fit in terms of information criteria.

Considering these evaluations, the logarithmic fit is considered the most suitable model for representing the relationship between the number of sensors and the average overall network troughput between the centralized and edge instance. The logarithmic equation provides a reliable estimation of the average overall network troughput based on the number of sensors.

The logarithmic fitting for network throughput provides several advantages in terms of scalability:

- **Moderate Growth Rate**: The logarithmic relationship indicates a diminishing rate of increase in network throughput as the number of sensors increases. This allows for smoother scaling and better resource management without experiencing a significant surge in resource demands.

- **Stable Growth Pattern**: The logarithmic fitting represents a stable growth pattern for network throughput, providing a more gradual and predictable increase in performance as the number of sensors grows. This stability aids in capacity planning, resource allocation, and system optimization.

In summary, the logarithmic fit obtained through MATLAB serves as compelling evidence of the system's scalability and its ability to handle increasing workloads. The logarithmic equation, provides a reliable estimation of the average overall network troughput and demonstrates the system's capacity to efficiently accommodate a growing number of sensors.

8.6 Final Considerations

In this section, the feasibility of the experimental architecture for monitoring multiple data centers in a Fog Computing environment was evaluated. The focus was on assessing the scalability and real-time properties of the system, which are critical factors in determining its effectiveness. A systematic evaluation methodology was employed to measure response times, CPU utilization, memory utilization and network troughput between the centralized and edge instance at different message rates, enabling an analysis of the system's performance and capacity to scale up.

The chosen approach of utilizing OpenWhisk actions and triggers proved to align well with the scalability and real-time requirements of the system. With automatic scaling based on the incoming workload, the OpenWhisk actions ensured that the system could handle increasing data from sensors while maintaining real-time response rates. This dynamic scalability eliminated the need for manual intervention or infrastructure adjustments, enhancing the feasibility of the solution.

Regarding response times, simulations were conducted by varying the message rate to mimic growing sensor counts. The measured response times provided insights into how the system handled the increasing workload. Curve fitting was performed to analyze the relationship between the message rate and response times. The results revealed that the best fit for response times was logarithmic, indicating a diminishing growth pattern as the message rate increased. Additionally, the maximum response time for the `addReadingToDb` action was consistently below 10 seconds, even as the number of sensors increased. This indicates that the system was able to process and store sensor readings within an acceptable time frame, ensuring efficient data handling and demonstrating the real-time properties of the architecture.

Similarly, CPU and memory utilization were monitored to understand the system's resource consumption patterns. Curve fitting was applied to analyze the relationship between the message rate and resource utilization. The analysis showed that CPU and memory utilization followed a linear growth pattern as the message rate increased.

Additionally, the analysis of network throughput between the edge and centralized instances was conducted. Similar to response times and resource utilization, curve fitting was performed to determine the relationship between the message rate and network throughput. The analysis indicated that the best fit for network throughput was logarithmic, suggesting a diminishing growth pattern as the message rate increased.

These findings demonstrate the scalability of the experimental architecture within the practical limitations of the combined approach. The logarithmic fit for response times and network throughput indicates efficient data processing and resource utilization, while the linear fit for CPU and memory utilization suggests a proportional scaling with the increasing message rate.

Prometheus and Grafana were utilized as monitoring and visualization tools to gather and analyze the metrics. These tools played a crucial role in providing valuable insights into the system's performance, facilitating effective data visualization and interpretation.

In summary, the evaluation results showcase the scalability, and real-time capabilities of the experimental architecture for monitoring multiple data centers. The system effectively handles increasing workloads while maintaining real-time response rates. The metrics and tools employed offer compelling evidence of the system's performance, with the analysis highlighting trends and observations that validate the feasibility of the proposed solution.

In conclusion, the evaluation of scalability and real-time properties provides a comprehensive understanding of the system's performance and its capacity to handle growing workloads. The experimental architecture, based on OpenWhisk actions and triggers, demonstrates itself as a feasible solution for monitoring multiple data centers in a Fog Computing environment.

Chapter 9

Conclusions

9.1 Research Objectives Overview

The main objectives of this research were to assess the scalability, real-time properties, and ease of management of the proposed serverless architecture, as well as to evaluate the effective integration between different components, in order to demonstrate the feasibility of the system. By achieving these objectives, valuable insights have been provided into the potential benefits and challenges of implementing such an architecture in real-world scenarios while ensuring a cohesive and well-integrated system for edge device monitoring.

9.2 Methodology Overview

Through a systematic research approach, a comprehensive literature review was conducted to establish a strong foundation and gain a deep understanding of serverless computing, fog computing, edge computing, and their applications in IoT environments. Based on the findings from the literature review, OpenWhisk was carefully selected as the preferred serverless platform due to its compatibility with the research objectives and functional requirements specific to edge device monitoring in a fog computing environment.

The system design was developed to outline the architecture and components of the proposed solution, emphasizing the utilization of computing capabilities, confinement of computation to the edge clusters, and incorporation of a centralized management component. The implementation phase involved deploying OpenWhisk on both the edge clusters and the centralized management component. The ESP-8266 sensor was utilized to collect temperature, gas percentage, and humidity data from each data center, ensuring comprehensive monitoring of their "health"

status.

9.3 Feasibility Evaluation Overview

To evaluate the feasibility of the implemented system, experiments and measurements were conducted. The primary focus was on assessing the system's scalability and real-time responsiveness as the number of sensors increased. Key metrics, including average response time, average network throughput, average CPU utilization, and average RAM utilization, were measured and analyzed.

The key findings from the evaluation demonstrate the scalability and real-time capabilities of the experimental architecture for monitoring multiple data centers in a Fog Computing environment. The chosen approach of utilizing OpenWhisk actions and triggers proved to align well with the scalability and real-time requirements of the system. With automatic scaling based on the incoming workload, the system effectively handled increasing data from sensors while maintaining real-time response rates. The dynamic scalability eliminated the need for manual intervention or infrastructure adjustments, enhancing the feasibility of the solution.

Furthermore, the analysis of response times, CPU utilization, memory utilization, and network throughput revealed important insights. The response times exhibited a logarithmic growth pattern as the message rate increased, indicating efficient data processing. Additionally, the maximum response time remained consistently below 10 seconds, demonstrating the system's ability to meet real-time requirements even as the number of sensors increased. Similarly, CPU and memory utilization followed a linear growth pattern, suggesting proportional scaling with the increasing message rate. The network throughput also showed a logarithmic growth pattern, indicating efficient data transfer between the edge and centralized instances.

Prometheus and Grafana, utilized as monitoring and visualization tools, played a crucial role in gathering and analyzing the metrics, providing valuable insights into the system's performance. These tools facilitated effective data visualization and interpretation, enhancing the understanding of the system's behavior under different workloads.

In conclusion, the evaluation of scalability and real-time properties demonstrates that the experimental architecture, based on OpenWhisk actions and triggers, is

a feasible solution for monitoring multiple data centers in a Fog Computing environment. The system effectively handles increasing workloads while maintaining real-time response rates. The metrics and tools employed offer compelling evidence of the system's performance, validating the feasibility of the proposed solution.

9.4 Proven Benefits

The implemented serverless architecture for edge device monitoring in a fog computing environment has demonstrated several proven benefits. These are the key advantages and positive outcomes of the proposed solution based on the evaluation and analysis conducted throughout the research:

- **Scalability and Real-Time Responsiveness**: One of the primary benefits of the implemented solution is its scalability and real-time responsiveness. The architecture, built on OpenWhisk actions and triggers, effectively handles increasing workloads by dynamically scaling resources based on the incoming data from sensors. This automatic scaling mechanism eliminates the need for manual intervention and ensures that the system can adapt to changing demands. The evaluation results have shown that the system maintains real-time response rates even with a growing number of sensors, making it highly suitable for monitoring multiple data centers in a fog computing environment.
- **Efficient Resource Utilization**: The implemented solution also demonstrates efficient resource utilization. The analysis of metrics, including CPU utilization, memory utilization, and network throughput, has revealed that the system scales proportionally with the increasing message rate. This efficient resource utilization ensures that computational resources are effectively allocated to handle the incoming workload, resulting in optimal performance and reduced resource waste.
- **Centralized Management and Monitoring**: The incorporation of a centralized management component in the architecture brings several benefits. It allows for centralized control and monitoring of the edge clusters, simplifying the management and configuration of the system. Additionally, the use of CouchDB, a centralized database, ensures that updating the action code is efficient and seamless. When a change is made to the action code, all the actions across the edge instances are automatically updated, eliminating the need for manual updates on each edge instance. This centralized approach to action code updates simplifies the maintenance and ensures consistency across the system.
- **Decentralized Communication**: Another benefit of the architecture is the fact that the centralized management instance does not make requests to the

edge instances. Instead, the edge instances make requests to the centralized management component when necessary. This decentralized communication approach improves system efficiency and reduces unnecessary overhead.

- **Flexibility and Adaptability**: The serverless architecture offers flexibility and adaptability, making it suitable for dynamic and evolving environments. The use of OpenWhisk actions and triggers allows for easy integration with other components and services, enabling seamless communication and data exchange. This flexibility enables the system to accommodate future enhancements and changes, making it a future-proof solution for edge device monitoring in fog computing environments.
- **Reproducibility and Extensibility**: The organization and documentation of the research materials in the GitHub repository facilitate reproducibility and extensibility. This documentation, combined with the code and resources shared in the repository, allows other researchers and practitioners to build upon the implemented solution, further advancing the field of fog computing and serverless edge device monitoring.

In summary, the implemented serverless architecture has demonstrated several proven benefits, including scalability, real-time responsiveness, efficient resource utilization, centralized management and monitoring, flexibility, and reproducibility. These benefits contribute to the viability and effectiveness of the proposed solution for edge device monitoring in a fog computing environment, opening up opportunities for its application in real-world scenarios.

9.5 Limitations and Future Studies

However, it is important to acknowledge the limitations of the experimental setup. Due to physical constraints and resource limitations, challenges were faced in deploying additional edge instances. Consequently, the focus was on demonstrating scalability by increasing the workload through the addition of more sensors on a single edge instance. While the ability to distribute the workload across multiple edge instances was limited, this approach still allowed for the assessment of the system's capability to handle increasing demands and provided valuable insights into its scalability within the constraints of the experimental setup.

It is also important to note that the limitation imposed by OpenWhisk's restriction on triggering more than 60 actions per minute was encountered during the evaluation. This constraint impacted the maximum number of sensors that could effectively be monitored using the implemented architecture. While the developed system demonstrated scalability and real-time capabilities within these constraints,

it is crucial to understand that the scalability performance may vary when the number of sensors exceeds the limit dictated by the serverless platform.

To further enhance the research in this area, future studies should focus on testing scalability by adding additional edge instances and sensors. This would enable a more comprehensive evaluation of the system's performance under different scenarios. Additionally, it would be beneficial to revisit the research hypothesis or research questions and assess how the findings align with the initial expectations. This evaluation could involve reflecting on whether the research supports or contradicts existing theories or hypotheses.

Future studies should also explore alternative serverless platforms or strategies to overcome the limitation of 60 triggers per minute to enable a more comprehensive evaluation of the system's scalability.

It is important to mention that the focus in this study was on demonstrating feasibility, so other studies should aim to make optimizations in resource consumption and minimize latency. Exploring other Function-as-a-Service (FaaS) tools would also be valuable to test the optimality of the system.

9.6 Final Statements

In closing, the research presented in this thesis has demonstrated the practical feasibility of a serverless architecture for edge device monitoring in a fog computing environment. The evaluation results highlight the scalability, real-time capabilities, and effective integration of the proposed solution. The findings have significant implications for the field of fog computing and IoT, offering insights into the potential benefits and challenges of implementing serverless architectures in real-world scenarios.

Appendix A

GitHub Repository

The GitHub repository for the thesis, located at `https://github.com/s294547/Exploring-FaaS-Solutions`, provides organized and detailed content within its various folders. Each folder within the repository includes a `README.md` file, which serves as a guide and explanation of the folder's contents and highlights the most important aspects. The main materials related to the thesis can be found in the `/openwhisk` folder.

The `/openwhisk/deploy` folder contains the final materials and code used for the thesis. This folder represents the culmination of the research and includes the deployment script, configuration files, and other relevant resources necessary for replicating the solution presented in the thesis.

The `/openwhisk/prototypes` folder contains different intermediate materials, they can't be considered as definitive. These materials may include experimental prototypes, code snippets, or early iterations of the implemented solution. They provide insights into the development process and demonstrate the evolution of ideas and concepts throughout the research.

The `/openwhisk/metrics` folder is dedicated to the materials related to the metrics harvest. Here, it is possible to find data collection tools, and analysis files specifically focused on capturing and evaluating performance metrics of the solution. These materials contribute to a comprehensive understanding of the system's performance and effectiveness.

By exploring the repository's folders and their respective `README.md` files, readers can navigate through the different stages of the research, access the main thesis materials, and delve into supporting resources and documentation. This organization facilitates easy access to the relevant content and enhances the reproducibility

and comprehension of the thesis work.

Bibliography

[1] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. «Trends in big data analytics». English (US). In: *Journal of Parallel and Distributed Computing* 74.7 (July 2014). Funding Information: Ananth Grama is the Director of the Computational Science and Engineering program and Professor of Computer Science at Purdue University. He also serves as the Associate Director of the Center for Science of Information. Ananth received his B. Engg from Indian Institute of Technology, Roorkee (1989), his M.S. from Wayne State University (1990), and Ph.D. from the University of Minnesota (1996). His research interests lie in parallel and distributed systems, numerical methods, large-scale data analysis, and their applications. Ananth is a recipient of the National Science Foundation CAREER award (1998), University Faculty Scholar Award (2002–07), and is a Fellow of the American Association for the Advancement of Sciences (2013)., pp. 2561–2573. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.01.003 (cit. on p. 1).

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. «Internet of Things (IoT): A vision, architectural elements, and future directions». In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660 (cit. on p. 1).

[3] Alessio Botta, Walter de Donato, Valeria Persico, and Antonio Pescapé. «Integration of cloud computing and internet of things: a survey». In: *Future Generation Computer Systems* 56 (2016), pp. 684–700 (cit. on p. 1).

[4] Alessandro Bocci, Stefano Forti, Gian-Luigi Ferrari, and Antonio Brogi. «Secure FaaS orchestration in the fog: how far are we?» In: *Computing* 103.4 (2021), pp. 1025–1056. DOI: 10.1007/s00607-021-00924-y (cit. on p. 2).

[5] Kevin Ashton. «That 'Internet of Things' Thing». In: *RFID Journal* (2009) (cit. on p. 7).

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. «The Internet of Things: A survey». In: *Computer Networks* 54.15 (2010), pp. 2787–2805 (cit. on p. 7).

[7] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. «Internet of Things (IoT): A vision, architectural elements, and future directions». In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660 (cit. on p. 7).

[8] Shancang Li, Lei Da Xu, and Shanshan Zhao. «The internet of things: A survey». In: *Information Systems Frontiers* 17.2 (2015), pp. 243–259 (cit. on p. 7).

[9] Shuo Wang, Jiafu Wan, Di Zhang, Daqiang Li, and Chunxiao Zhang. «Towards smart factory for industry 4.0: A self-organized multi-agent system with big data based feedback and coordination». In: *Computer Networks* 101 (2016), pp. 158–168 (cit. on p. 7).

[10] Muhammad Imran, Muhammad Junaid, Irfan Aslam, Muhammad Rizwan, and Imran Javaid. «Fog computing: A taxonomy, survey and future directions». In: *arXiv preprint arXiv:1611.05539* (2016), pp. 1–4 (cit. on p. 8).

[11] IBM. *Function-as-a-Service (FaaS)*. Retrieved from IBM website. n.d. URL: https://www.ibm.com/topics/faas#:~:text=FaaS%2C%20or%20Function%2Das%2D,building%20and%20launching%20microservices%20applications (cit. on pp. 10, 11).

[12] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, Antonio Brogi, and Frank Leymann. «FaaSten your decisions: A classification framework and technology review of function-as-a-Service platforms». In: *Journal of Systems and Software* 175 (2021), p. 110906. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110906. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000030 (cit. on pp. 13, 15, 20, 22).

[13] Andrew St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly Media, 2004 (cit. on p. 13).

[14] Eric von Hippel and Georg von Krogh. «Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science». In: *Organization Science* 14.2 (2003), pp. 209–223 (cit. on p. 14).

[15] Wen Li, Na Meng, Li Li, and Haipeng Cai. «Multi-Language Software Projects on GitHub». In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2021, pp. 256–256. DOI: 10.1109/ICSE-Companion52605.2021.00119 (cit. on p. 14).

[16] Sean Mealin and Emerson Murphy-Hill. «An exploratory study of blind software developers». In: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2012, pp. 71–74 (cit. on p. 14).

[17] Khaled Albusays and Stephanie Ludi. «Eliciting programming challenges faced by developers with visual impairments: exploratory study». In: *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. 2016, pp. 82–85 (cit. on p. 14).

[18] Kattiana Constantino, Shurui Zhou, Mauricio Souza, Eduardo Figueiredo, and Christian Kästner. «Understanding Collaborative Software Development: An Interview Study». In: *Proceedings of the 15th International Conference on Global Software Engineering*. ICGSE '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 55–65. ISBN: 9781450370936. DOI: 10.1145/3372787.3390442. URL: https://doi.org/10.1145/3372787.3390442 (cit. on p. 14).

[19] Noela Jemutai Kipyegen and William P K Korir. «Importance of Software Documentation». In: *International Journal of Computer Science Issues (IJCSI)* 10.5 (Sept. 2013), pp. 223–228 (cit. on p. 15).

[20] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. *Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing*. 2022. arXiv: 2109.13492 [cs.DC] (cit. on p. 15).

[21] Scott Rogers. *Under the Microscope: Software Observability in a Distributed Architecture*. Blog post. Accessed on June 14, 2023. Nov. 2020. URL: https://tanzu.vmware.com/developer/blog/under-the-microscope-software-observability-in-a-distributed-architecture/ (cit. on p. 15).

[22] OpenFaaS. *OpenFaaS - Serverless Functions Made Simple*. Retrieved from https://www.openfaas.com/. n.d. (Cit. on p. 17).

[23] Apache OpenWhisk. *Apache OpenWhisk – Open Source Serverless Cloud Platform*. Retrieved from https://openwhisk.apache.org/. n.d. (Cit. on pp. 17, 34).

[24] Knative. *Knative - Kubernetes-Based Platform to Build, Deploy, and Manage Modern Serverless Workloads*. Retrieved from https://knative.dev/. n.d. (Cit. on p. 18).

[25] AWS Lambda. *AWS Lambda*. Retrieved from https://aws.amazon.com/lambda/. n.d. (Cit. on p. 18).

[26] Google Cloud Functions. *Google Cloud Functions*. Retrieved from https://cloud.google.com/functions. n.d. (Cit. on p. 19).

[27] Azure Functions. *Azure Functions*. Retrieved from https://azure.microsoft.com/en-us/services/functions/. n.d. (Cit. on p. 19).

[28] Andrew Morin, Jennifer Urban, and Piotr Sliz. «A Quick Guide to Software Licensing for the Scientist-Programmer». In: *PLOS Computational Biology* 8.7 (July 2012), pp. 1–7. DOI: 10.1371/journal.pcbi.1002598. URL: https://doi.org/10.1371/journal.pcbi.1002598 (cit. on p. 26).

[29] Mishal Roomi. *6 Advantages and Disadvantages of Closed Source Software | Drawbacks & Benefits of Closed Source Software*. Website. Accessed on June 14, 2023. May 2021. URL: https://www.hitechwhizz.com/2021/05/6-advantages-and-disadvantages-drawbacks-benefits-of-closed-source-software.html (cit. on p. 26).

[30] Apache OpenWhisk. *OpenWhisk – How the System Works*. https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works. Retrieved from Apache OpenWhisk GitHub repository. Accessed 2023 (cit. on pp. 34, 39).

[31] Apache OpenWhisk. *OpenWhisk – OpenWhisk CLI*. https://github.com/apache/openwhisk/blob/master/docs/cli.md. Retrieved from: https://github.com/apache/openwhisk/blob/master/docs/cli.md. Accessed 2023 (cit. on p. 36).

[32] Apache OpenWhisk. *OpenWhisk – OpenWhisk Actions*. https://github.com/apache/openwhisk/blob/master/docs/actions.md. Retrieved from: https://github.com/apache/openwhisk/blob/master/docs/actions.md. Accessed 2023 (cit. on p. 37).

[33] Apache OpenWhisk. *OpenWhisk – Creating triggers and rules*. https://github.com/apache/openwhisk/blob/master/docs/triggers_rules.md. Retrieved from: https://github.com/apache/openwhisk/blob/master/docs/triggers_rules.md. Accessed 2023 (cit. on p. 42).

[34] OpenWhisk. *Using and creating OpenWhisk packages*. GitHub. Retrieved from: https://github.com/apache/openwhisk/blob/master/docs/packages.md (cit. on p. 45).

[35] *Building a Sample MQTT-based Application on OpenWhisk*. https://blog.zhaw.ch/splab/2019/03/15/building-a-sample-mqtt-based-application-on-openwhisk/ (cit. on pp. 46, 65, 105).

[36] *Advanced OpenWhisk Alarm Schedules*. https://jamesthom.as/2017/10/advanced-openwhisk-alarm-schedules/ (cit. on pp. 46, 67).

[37] Adobe. *OpenWhisk – ShardingContainerPoolBalancer*. https://github.com/adobe-apiplatform/incubator-openwhisk/blob/master/core/controller/src/main/scala/org/apache/openwhisk/core/loadBalancer/ShardingContainerPoolBalancer.scala. Retrieved from: https://github.com/adobe-apiplatform/incubator-openwhisk/blob/master/core/controller/s Accessed 2023 (cit. on p. 46).

[38] C. Toma, A. Alexandru, M. Popa, and A. Zamfiroiu. «IoT Solution for Smart Cities' Pollution Monitoring and the Security Challenges». In: *Sensors* 19.8 (2019), pp. 1–20. DOI: 10.3390/s19081913 (cit. on p. 49).

[39] L. Colizzi, D. Caivano, C. Ardito, G. Desolda, A. Castrignanò, M. Matera, ..., and H. Shi. «Agricultural Internet of Things and Decision Support for Precision Smart Farming». In: *Agricultural Internet of Things and Decision Support for Precision Smart Farming*. Springer, 2020, pp. 1–33. DOI: 10.1007/978-3-030-23605-7_1 (cit. on pp. 49, 50).

[40] Claudio Cicconetti, Marco Conti, and Andrea Passarella. «Architecture and performance evaluation of distributed computation offloading in edge computing». In: *Simulation Modelling Practice and Theory* 101 (2020). Modeling and Simulation of Fog Computing, p. 102007. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2019.102007. URL: https://www.sciencedirect.com/science/article/pii/S1569190X19301406 (cit. on p. 54).

[41] Kartheek. «How to Deploy Highly Scalable MQTT Broker to AWS (HiveMQ)». In: *Geek Culture* (Mar. 2021). URL: https://medium.com/geekculture/how-to-deploy-highly-scalable-mqtt-broker-to-aws-hivemq-2b88ce0c7f43 (cit. on p. 54).

[42] Charles Mahler. «8 Real-World MQTT Use Cases». In: *InfluxDB, Community, Iot* (Sept. 2022). URL: https://www.influxdata.com/blog/mqtt-use-cases/ (cit. on p. 54).

[43] InfluxData. *A Guide to MQTT*. https://www.influxdata.com/mqtt (cit. on p. 54).

[44] Chiara Caiazza, Silvia Giordano, Valerio Luconi, and Alessio Vecchio. «Edge computing vs centralized cloud: Impact of communication latency on the energy consumption of LTE terminal nodes». In: *Computer Communications* 194 (2022), pp. 213–225. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2022.07.026. URL: https://www.sciencedirect.com/science/article/pii/S0140366422002730 (cit. on p. 56).

[45] Microsoft Azure. *What is edge computing?* https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-edge-computing/ (cit. on p. 56).

[46] InfluxData. *Why Time Series Matters for Metrics, Real-Time and Sensor Data*. https://www.influxdata.com/resources/why-time-series-matters-for-metrics-real-time-and-sensor-data/ (cit. on pp. 56, 58).

[47] Luis Miguel Rodriguez Cortes, Edward Paul Guillen, and Wilson Rojas Reales. «Serverless Architecture: Scalability, Implementations and Open Issues». In: *2022 6th International Conference on System Reliability and Safety (ICSRS)*. 2022, pp. 331–336. DOI: 10.1109/ICSRS56243.2022.10067577 (cit. on p. 59).

[48] *Grafana Labs*. https://grafana.com/ (cit. on pp. 61, 69).

[49] *Eclipse Mosquitto*. https://mosquitto.org/ (cit. on p. 65).

[50] *InfluxDB*. https://www.influxdata.com/products/influxdb/ (cit. on p. 68).

[51] *Apache CouchDB*. https://couchdb.apache.org/ (cit. on p. 68).

[52] *Kubernetes*. https://kubernetes.io/docs/home/ (cit. on p. 76).

[53] *Docker*. https://docs.docker.com/ (cit. on p. 77).

[54] *Helm*. https://helm.sh/docs/ (cit. on p. 78).

[55] *Chartmuseum*. https://github.com/helm/chartmuseum (cit. on p. 79).

[56] *Traefik*. https://traefik.io/ (cit. on p. 79).

[57] *Arduino IDE*. https://andprof.com/tools/what-is-arduino-software-ide-and-how-use-it/ (cit. on p. 80).

[58] *k3s*. https://docs.k3s.io/ (cit. on p. 81).

[59] *ESP8266 - Technical Reference*. https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf (cit. on p. 82).

[60] *CouchDB – Helm Chart*. https://github.com/apache/couchdb-helm (cit. on p. 99).

[61] *OpenWhisk – Helm Chart*. https://github.com/apache/openwhisk-deploy-kube/tree/master/helm/openwhisk (cit. on p. 101).

[62] *Mosquitto – Helm Chart*. https://github.com/SINTEF/mosquitto-helm-chart (cit. on p. 103).

[63] *InfluxDB – Helm Chart*. https://helm.influxdata.com/ (cit. on p. 104).

[64] *Grafana Operator – Documentation*. https://grafana-operator.github.io/grafana-operator/docs/ (cit. on p. 108).

[65] *What Are Kubernetes Custom Resource Definitions (CRDs)?* https://www.howtogeek.com/devops/what-are-kubernetes-custom-resource-definitions-crds/ (cit. on p. 108).

[66] Gartner. *Scalability*. [Online]. Available: urlhttps://www.gartner.com/en/information-technology/glossary/scalability. Retrieved 2023 (cit. on p. 110).

[67] Forbes. *How to use Real Time Data?* [Online]. Available: urlhttps://www.forbes.com/sites/bernardmarr/2022/03/14/how-to-use-real-time-data-key-examples-and-use-cases/?sh=2d90d1667f4d. Retrieved 2023 (cit. on p. 111).

[68] *OpenWhisk – OpenWhisk System Details.* `https://github.com/apache/openwhisk/blob/master/docs/reference.md` (cit. on p. 115).

[69] *Wikipedia – Curve Fitting.* `https://en.wikipedia.org/wiki/Curve_fitting` (cit. on p. 119).

[70] *MathWorks – Matlab.* `https://it.mathworks.com/products/matlab.html` (cit. on p. 123).

[71] *MathWorks – polyfit.* `https://it.mathworks.com/help/matlab/ref/polyfit.html` (cit. on p. 123).

[72] *MathWorks – lsqcurvefit.* `https://it.mathworks.com/help/optim/ug/lsqcurvefit.html` (cit. on p. 123).

[73] Wikipedia. *Coefficient of Determination.* [Online]. Available: urlhttps://en.wikipedia.org/wiki/Coefficient$_o f_d etermination$. Retrieved 2023 (cit. on p. 125).

[74] Wikipedia. *Root Mean Square Deviation.* [Online]. Available: urlhttps://en.wikipedia.org/wiki/Root-mean-square$_d eviation$. Retrieved 2023 (cit. on p. 126).

[75] Wikipedia. *Akaike Information Criterion.* [Online]. Available: urlhttps://en.wikipedia.org/wiki/Akaike$_i nformation_c riterion$. Retrieved 2023 (cit. on p. 127).

[76] Wikipedia. *Bayesian Information Criterion.* [Online]. Available: urlhttps://en.wikipedia.org/wiki/Bayesian$_i nformation_c riterion$. Retrieved 2023 (cit. on p. 127).

[77] *Prometheus – Overview.* `https://prometheus.io/docs/introduction/overview/` (cit. on p. 131).

[78] *OpenWhisk – OpenWhisk Metric Support.* `https://github.com/apache/openwhisk/blob/master/docs/metrics.md` (cit. on p. 132).

[79] *Prometheus – Monitoring Linux host metrics with the Node Exporter.* `https://prometheus.io/docs/guides/node-exporter/` (cit. on p. 134).

[80] *Kubernetes – kube-state-metrics.* `https://github.com/kubernetes/kube-state-metrics` (cit. on p. 134).

[81] *Grafana Labs – Prometheus Data Source.* `https://grafana.com/docs/grafana/latest/datasources/prometheus/` (cit. on p. 135).

[82] *10 application performance metrics and how to measure them*. https://www.techtarget.com/searchapparchitecture/tip/5-application-performance-metrics-all-dev-teams-should-track (cit. on pp. 135–138).