University of Patras - Polytechnic School
Department of Electrical Engineering
and Computer Technology

Department: Electronics and Computers Department
Laboratory: NaN

Thesis

of the student of Department of Electrical Engineering
and Computer Technology of the Polytechnic School of the University of Patras

Anastasios Zampetis of Michael

record number: 228322

Subject

Data Search & Extraction with Microservices

Supervisor

Professor Dr. Evangelos Dermatas, University of Patras

Thesis Number:

Patras, February 2025

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η διπλωματική εργασία με θέμα

Αναζήτηση & Εξόρυξη Δεδομένων με Microservices

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

Αναστασίου Ζαμπέτη του Μιχαήλ

(Α.Μ.: 228322)

παρουσιάστηκε δημόσια και εξετάστηκε στο τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις

__/__/____

Ο Επιβλέπων

Ο Διευθυντής του Τομέα

ΔΡ. Ευάγγελος Δερματάς
Καθηγητής

Ο Συν-Επιβλέπων

NaN
NaN

NaN
NaN

CERTIFICATION

It is certified that the Thesis with Subject

Data Search & Extraction with Microservices

of the student of the Department of Electrical Engineering & Computer Technology

Anastasios Zampetis of Michael

(R.N: 228322)

Was presented publicly and defended at the Department of Electrical Engineering & Computer Technology at

__/__/___

The Supervisor

The Director of the Division

Dr. Evangelos Dermatas
Professor

The Co-Supervisor

NaN
NaN

Thesis details

Subject: **Data Search & Extraction with Microservices**

Student: **Anastasios Zampetis of Michael**

Supervising Team
Professor Dr. Evangelos Dermatas

NaN NaN

Laboratories:
NaN

Thesis research period:
September 2018 - September 2019

This thesis was written in LaTeX.

Abstract

Internet of Things (IoT) is an enabling technology for numerous domains worldwide, such as smart cities, manufacturing, logistics and critical infrastructure. On top of Internet of Things, the architectural paradigm shifts from cloud-centric to Edge-centric, offloading more and more functionality from the cloud to the Edge devices. Edge computing devices are transformed from simply aggregating data to performing data processing and decision making, accelerating the decentralization of the IoT domain. Given the considerable increase of the number of IoT devices and the size of the generated data, an increase of Edge devices with augmented functionality is expected. We propose an ”Edge as a service” scheme, where Edge devices will be able to procure unused resources and run services that are requested and consumed by IoT devices that belong to different stakeholders. In this work, we outline a high-level architecture of this scheme and give a reference implementation of a narrow part of the system, boasting high modularity and the extensive use of Open Source Technologies.

Acknowledgements

I would like to thank, first and foremost, Christos Tranoris, Researcher at the NAM group, who guided me at every step of the thesis, put up with incoherent messages and ideas during the most improper hours. I have the privilege to call him a mentor, a colleague a and a friend. I would also like to thank Professor Spiros Denazis for introducing me to the NAM group and for his insightful comments on this work. Most importantly, I want to thank John Gialelis who introduced me to the world of Academic Research and who has been my mentor for over 2 years. Finally, I want to thank the whole team of NAM group for the fruitful conversations, as also the online communities and developers of the projects: *EdgeX Foundry*, *Filecoin*, *loraserver.io*, *Node-RED* and *balena*, for their continuing assistance and guidance throughout the implementation. I dedicate this work to my Parents, *Despoina* and *George*, who laid the foundations for everything I have accomplished thus far, and to my friends and my companion, who have supported me, put up with me and pushed me to become a better human being.

Contents

List of Figures **xv**

List of Tables **xvii**

1 Design concepts **1**
- 1.1 Monolithic Architecture 1
- 1.2 Microservice Architecture 2
- 1.3 Containers 3
- 1.4 IoT device Simulators 5

2 Technologies **7**
- 2.1 Docker 7
- 2.2 Prometheus 8
- 2.3 Grafana 9
- 2.4 Message Queuing Telemetry Transport (MQTT) 10

3 Implementation **13**
- 3.1 Design and Architecture 13
- 3.2 Dataset 13
- 3.3 Sensor Simulator 15
- 3.4 MQTT Broker 15
- 3.5 Controller Node 15
- 3.6 Prometheus 15
- 3.7 Grafana 15
- 3.8 Orchestration 15

Bibliography **17**

List of Figures

1.1 Monolithic Architecture 2
1.2 Microservice Architecture 4
1.3 Containerized Applications 5
1.4 IoT ecosystem 6

2.1 Docker Workflow 8
2.2 Prometheus-Grafana Monitoring stack 10
2.3 MQTT protocol workflow 11

3.1 Implementation Diagram 14

List of Tables

3.1 Dataset Attribute Information 14

1. Design concepts

1.1 Monolithic Architecture

Before going into Microservices and the Microservice architecture, the Monolithic architecture approach must be explained first. The Monolithic architecture approach was, until recently, the preferred design option for software. In a Monolithic application, all the different components and functions of the business logic are combined into one indivisible program [1]. Generally, these components are the user interface, business rules, and data access. While individual components might be developed separately, they remain tightly coupled [2], and any change completed in any of them requires the whole program to be rebuilt and redeployed [3].

This tightly coupled nature creates a significant dependency problem. More often than not, development in one component requires functional changes in multiple others, adding to the development cost, complicating the build and testing process, and inducing delays in deployment. For example, a minor change in the user interface might necessitate updates to both the business logic and data access layers. Additionally, a single bug in any one component can potentially halt the entire application's operation and create a nightmarish situation for on-call engineers trying to figure out the root cause. This often results in multiple unrelated-to-the-issue teams joining in until root cause analysis is complete, leading to wasted resources and prolonged downtime. Such situations underscore the inherent fragility of the Monolithic approach in complex systems.

Another significant drawback of Monolithic applications is the large codebase they often entail. Over time, as the application grows, the codebase can become cumbersome to manage and increasingly difficult to understand for new developers [2]. Implementing even minor changes may require navigating through extensive, interconnected code, leading to higher development times and increased chances of introducing bugs. Maintenance becomes a challenge as technical debt accumulates, and the lack of modularity makes it harder to address specific areas without affecting the entire system.

Scalability is another major issue with Monolithic applications. Different components typically have conflicting resource requirements; for instance, one might be CPU-intensive while another is memory-intensive. Because of the unified design, all resource requirements must be handled together, making vertical scaling (increasing the power of a single server) impossible. Horizontal scaling (adding multiple copies of the application to distribute the load) remains the only option, but it is resource-intensive, inefficient, and often restricted due to the complexities of managing state across instances. This limitation can make Monolithic applications ill-suited for dynamic workloads or environments requiring rapid scaling.

Finally, Monolithic design allows for little to no flexibility when it comes to incorporating newer, state-of-the-art technologies. For instance, integrating a new database technology or a modern programming language might require a complete overhaul of the application, as its unified structure does not support gradual upgrades. Over time, this rigidity results in Monolithic applications becoming legacy systems. These systems often lag in performance and reliability compared to modern architectures and may eventually need to be completely redesigned and reconstructed, a process that is both time consuming and costly.

Despite these many drawbacks, Monolithic architecture is still favored for certain applications due to its core benefits. The most important one is performance. In most cases, Monolithic applications outperform their modular counterparts because all components run in the same memory space and avoid the overhead of inter-process communication [2]. The simplicity of having a single executable can lead to faster runtime and lower latency, making it suitable for scenarios where performance is paramount.

Initial design and implementation are also easier with a Monolithic approach, particularly for small to medium sized applications. Individual components are usually clearly defined at later stages, reducing the upfront complexity during the design phase. This simplicity extends to the unified build and deployment process, which simplifies configuration management, testing, and monitoring.

Monolithic architecture approach is particularly well-suited for smaller applications or projects with well-defined, stable requirements. It helps to get things up and running faster, making it an ideal choice for situations where time to market is critical. Furthermore, when development complexity and deployment time come second to performance, a Monolithic application typically has the edge over a modular approach.

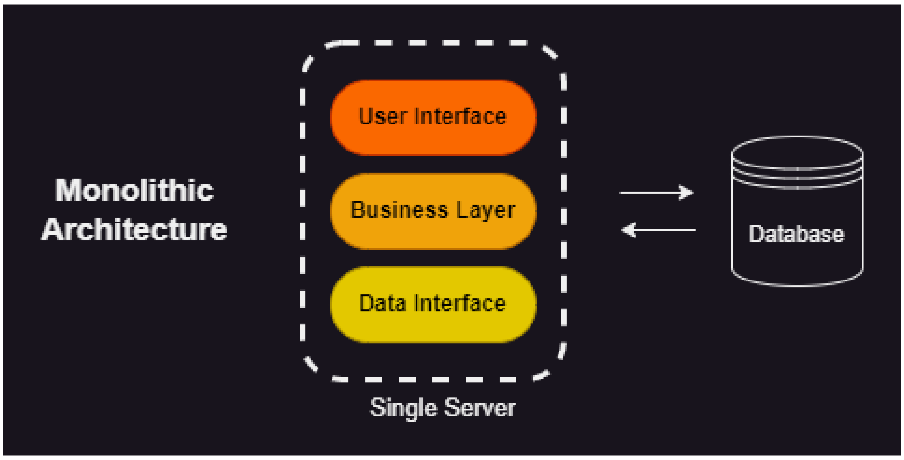

Figure 1.1: Monolithic Architecture

1.2 Microservice Architecture

Microservices and the Microservice Architecture have, in recent years, become one of the most popular design options for software applications. In the Microservice Architecture, the application is structured as a collection of independent services, called Microservices. Each Microservice corresponds to a distinct part of the business logic, executing a well-defined, unique process [1] [4]. These Microservices utilize lightweight communication mechanisms, such as API interfaces, allowing them to operate in unison and achieve the same final results as a Monolithic application but without being co-dependent. The independence of Microservices enables them to be built and tested separately, and they can be deployed and scaled independently as well. Each Microservice is designed to facilitate a single function of the application, making it focused, manageable, and easily comprehensible [5].

The advantages of this architectural approach compared to its Monolithic counterpart are significant and far-reaching. Each individual Microservice can be developed in isolation, often by different teams, without compromising or delaying the development of other parts of the application. This separation of concerns allows for parallel development, reducing bottlenecks and improving efficiency. Consequently, different components of the application can be updated and enhanced asynchronously, resulting in quicker and more frequent deployments. The build and deployment process is streamlined and resource-efficient since only the Microservice being updated needs to be redeployed, rather than the entire application.

Testing is another area where Microservice architecture shines. Since each Microservice operates independently, there is no need to build and test the entire application for changes made to one component. Instead, testing can be focused solely on the Microservice being modified. This leads to faster and more efficient testing cycles, allowing development teams to identify and resolve issues promptly. Similarly, debugging is simplified; when an error occurs, it is relatively easy to pinpoint the Microservice responsible for the failure. The responsible team can address the issue without disrupting the operation of the entire application, making the debugging process both quicker and more effective.

While these advantages are compelling, the most critical benefit of the Microservice Architecture is scalability. In the era of cloud-native applications, where the ability to scale up or down on demand is of utmost importance and operational costs are often usage-based, Microservice-based applications significantly outclass their Monolithic counterparts. Scaling in a Microservice application is versatile, as it is possible both vertically and horizontally. The entire application can be replicated if necessary, just like a Monolithic application. However, the true strength of Microservices lies in the ability to scale individual components. For example, if one Microservice experiences a spike in demand, only that specific service can be scaled up, optimizing resource usage.

Moreover, Microservice-based applications align seamlessly with cloud infrastructure. Since Microservices can be readily instantiated as needed, there is no requirement for maintaining multiple always-on instances to handle demand spikes, as is often the case with Monolithic applications. This elasticity in scaling reduces idle resource consumption and exponentially decreases operational costs.

Despite these numerous advantages, the Microservice architecture is not without its drawbacks. The initial development of Microservice applications requires careful and time-consuming planning and design. During the early stages of development, requirements and features are often not well defined, making it challenging to design Microservices effectively. Any missteps in the design can lead to significant complications later on.

Performance is another area where Microservice-based applications can fall short compared to Monolithic ones. Because Microservices rely on lightweight communication mechanisms, such as APIs, there is an inherent overhead in inter-service communication. This can result in latency, making Microservice applications less suitable for time-critical operations, such as load balancers or real-time data processing. For use cases requiring the lowest possible response times, Monolithic applications often have a performance edge.

Finally, Microservice architecture may not be the best choice for on-premises applications where customers are required to set up and manage everything manually. The complexity of configuring and maintaining multiple independent services can be overwhelming for end-users, especially those without extensive technical expertise. In such scenarios, a Monolithic application might be preferable due to its simplicity and ease of deployment [6].

1.3 Containers

Hand in hand with the Microservice Architecture came containers. Containers are a form of virtualization similar to virtual machines (VMs), but unlike traditional virtual machines, containers share the host system's kernel while running in isolated user spaces. This architecture makes them significantly more lightweight, efficient, and versatile compared to VMs. By eliminating the need for a full guest operating system, containers reduce resource overhead dramatically, enabling more efficient utilization of host hardware. This efficiency translates to faster startup times, reduced memory and processing power requirements, and greater performance from the same hardware infrastructure compared to VMs [7].

The lightweight nature of containers is particularly advantageous in scalable environments where applications need to scale up or down quickly in response to demand. In contrast, VMs require a

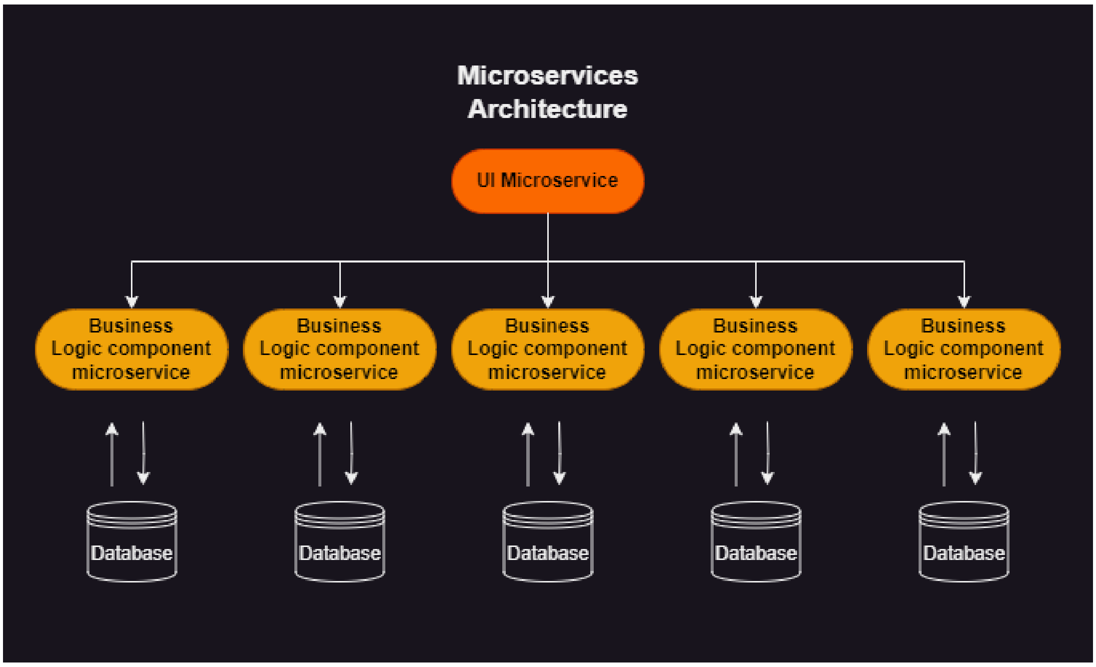

Figure 1.2: Microservice Architecture

separate guest operating system for each instance, which not only increases resource consumption but also lengthens deployment and startup times. Containers, on the other hand, are designed to spin up almost instantly, enabling rapid scaling and responsiveness.

Containers are created and deployed using container images, which are essentially templates. These images are built using ContainerFiles, which specify a base image and a sequence of instructions or steps to execute on top of it. Each step in the build process forms a new layer. Layering is a critical feature for image creation and deployment workflows, as it allows for reuse across different images. For instance, if multiple container images share common steps, such as installing a specific library, those layers need to be built only once. This reuse drastically reduces build times and conserves storage space, making containerization particularly well-suited to agile development practices where frequent builds and deployments are standard.

The distribution of container images is made straightforward through container registries, platforms where images can be stored, shared, and accessed. Docker Hub is the registry associated with the most popular container platform, Docker. The ability to easily distribute container images enhances collaboration and accelerates development cycles. Teams working in different environments can seamlessly pull images from registries, ensuring consistent runtime environments and reducing potential configuration mismatches.

One of the key strengths of containers lies in their ability to provide a consistent runtime environment. By encapsulating all dependencies and configurations within the container image, containers ensure that applications run identically across development, staging, and production environments. This consistency simplifies the testing and deployment process, minimizing release time and reducing the likelihood of environment-specific bugs.

However, as the use of containers grows, managing a large number of containers across multiple hosts becomes increasingly complex. Efficient deployment and management of containerized applications require some form of orchestration. Container orchestration platforms, such as Kubernetes, have emerged as the standard solution for automating the deployment, scaling, and management of containerized applications [8]. Such platforms ensure high availability by intelligently distributing containers across nodes, maintaining desired states even during failures, and handling incident recovery automatically.

Moreover, they provide capabilities such as load balancing and resource allocation, enabling developers to build resilient and scalable systems with minimal manual intervention. Another major feature of these platforms are rolling updates, which ensure that new versions of an application can be deployed without downtime and handle rollback processes in case of failure. These features are only possible by taking advantage of the lightweight, lightning-fast deployment of containers and not only enhance the performance and reliability of applications but also reduce the operational burden on development teams.

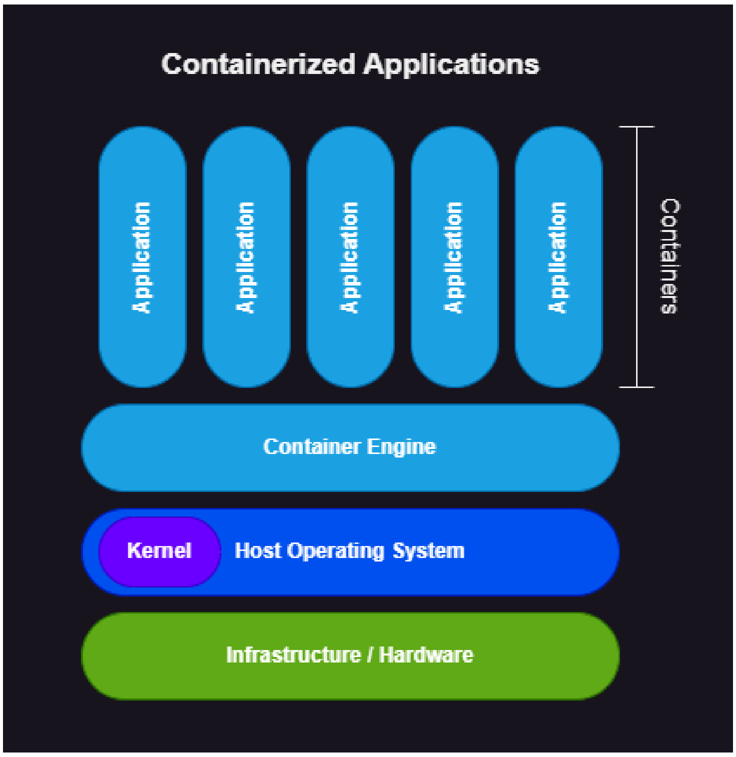

Figure 1.3: Containerized Applications

1.4 IoT device Simulators

There's little doubt that the Internet of Things (IoT) is here to stay. IoT has reshaped the way humans and machines interact with the environment, creating smarter, more connected systems that are now strongly influencing most industries. In recent years, an increasing number of everyday devices have been equipped with various sensors and internet connectivity, enabling them to gather and transmit large volumes of data. This exponential growth of IoT devices has brought about significant opportunities, but it also presents unique challenges.

The data generated by IoT devices is massive and diverse, requiring robust IoT applications to process, analyze, and utilize it effectively. These applications are designed to transform raw data into valuable insights, enhancing the functionality of devices, and promoting efficiency in operations. However, as with any software, IoT applications must undergo rigorous testing before deployment to ensure reliability, security, and optimal performance.

One approach to testing IoT applications is to create a physical IoT network composed of the actual devices and sensors specific to the application. This setup allows for real-world data generation and provides an environment for testing the application under realistic conditions. While this method has its merits, it is not hard to notice the significant challenges and limitations it poses.

First and foremost, creating an actual IoT network for testing can be prohibitively expensive. IoT ecosystems often consist of diverse and specialized devices, many of which may be costly to acquire or replicate at scale. For large and complex ecosystems, the financial overhead of assembling and maintaining such a network can be quite substantial, making this approach impractical in most cases.

Furthermore, IoT applications, particularly in the early stages of development, are subject to frequent redesigns and changes. These iterations are part of the development process as requirements change, features are added, and feedback is provided. However, when an IoT network is used for testing, any significant change in the application may necessitate corresponding changes to the physical network. This adds to the development cost and introduces delays, as modifying or expanding an IoT network is both time-consuming and resource-intensive.

Even when an IoT network is already in place for deployment purposes, using it for application testing poses additional risks. Testing on a live network can expose sensitive data to potential security vulnerabilities.

To address these challenges, a more efficient and safer alternative is the use of synthetic data. Synthetic data simulates the behavior and output of real IoT devices, providing a realistic representation of IoT network activity without requiring a physical network. This approach allows developers to create virtual IoT environments tailored to specific applications, enabling thorough testing under controlled conditions.

By generating synthetic data, developers can replicate complex IoT ecosystems at a minimum cost compared with physical networks. These virtual environments can be easily modified to accommodate changes in the application, supporting agile development without the need for expensive hardware adjustments. Additionally, synthetic data eliminates the security risks associated with using real-world data during testing, as no actual devices or sensitive information are involved.

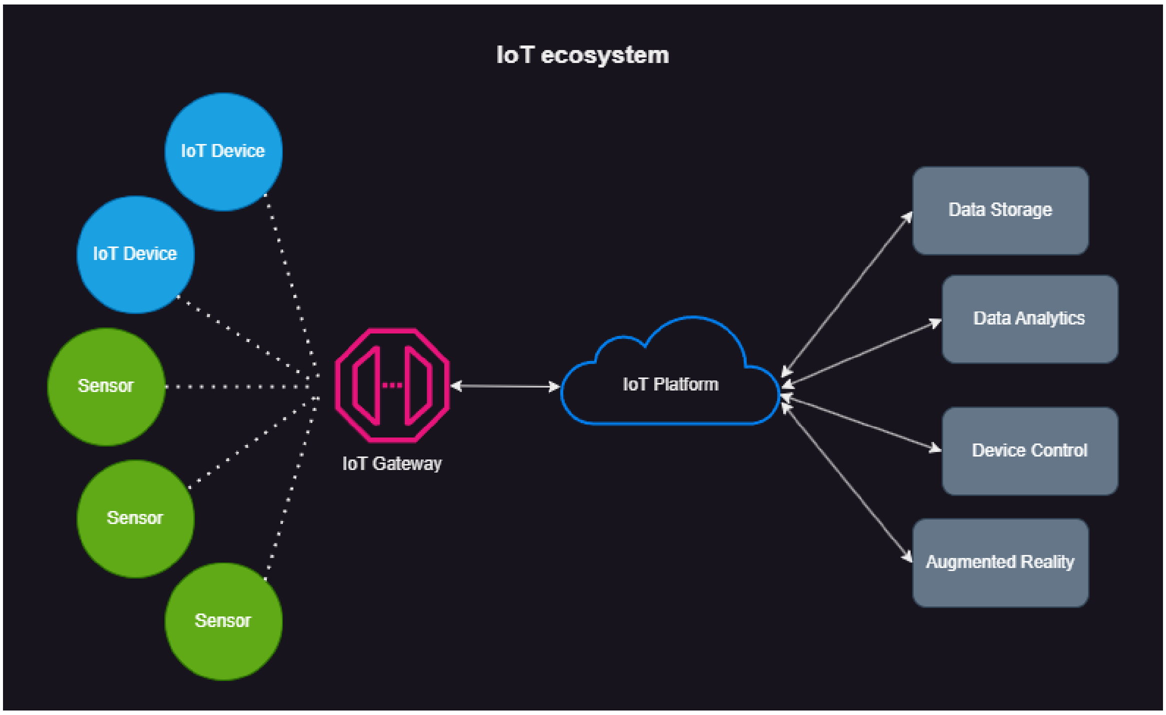

Figure 1.4: IoT ecosystem

2. Technologies

2.1 Docker

When discussing containers, Docker cannot be left out of the discussion. Docker is an open-source platform designed to automate the creation, deployment, scaling, and management of containerized applications. It is, by far, the most widely adopted tool for containerization. Docker allows the packaging of applications and their dependencies into lightweight and easily portable containers that can be deployed consistently across a wide range of environments. This portability ensures that applications behave identically, whether running on a developer's laptop, a staging environment, or a production server.

To provision containers, Docker uses images. Docker images are read-only templates used to create containers, similar to .iso files for virtual machines, but are more lightweight and versatile. Docker images bundle everything an application needs to run, including the operating system, application code, dependencies, libraries, and configuration metadata. The metadata often includes the entry point script, a set of commands executed when the container is instantiated.

An important feature of Docker images is their layered architecture. Each layer represents a distinct change, such as adding a file, installing a package, or modifying a configuration. This layered design allows developers to build images on top of existing ones, significantly reducing build times, image sizes, and data transfer requirements.

The runtime environment responsible for building, running, and managing containers is the Docker Engine, and it consists of three main components. The first is the Docker Daemon, a background service responsible for managing Docker objects, such as containers, images, volumes, and networks. Next is the Docker Command-Line Interface (CLI), which provides a way to interact with Docker through terminal commands. Finally, the REST API enables programmatic access to Docker's functionalities.

Docker images are created using Dockerfiles, which act as blueprints for the image creation process. A Dockerfile contains step-by-step instructions for building an image, including the base image, commands to configure the environment, install dependencies, and metadata such as port configurations and the entry point script. This declarative approach ensures reproducibility, as anyone with the Dockerfile can recreate the same image, ensuring consistency across teams.

Since images are meant to be portable and used over multiple environments, remote registries to store and fetch images from are crucial. Docker Hub is a free, widely used registry provided by the wider Docker ecosystem.

Hand in hand with containers, Docker enables the creation and management of other resources critical for smooth operation. Volumes are a mechanism for persisting data generated and used by containers. Unlike ephemeral container storage, volumes ensure data remains intact even after container deletion. Docker also creates networks, enabling container interconnectivity and communication between containers and the outside world.

For handling deployments of multiple containers in a programmatic way, Docker Compose can be utilized. Docker Compose is a tool that utilizes simple YAML files to manage multi-container deployments or applications by defining services, networks, and volumes required. This approach reduces complexity and enhances reproducibility, making it easier to manage applications with multiple interconnected components.

Finally, Docker provides a native container orchestration platform, Docker Swarm, but on a production level, it is outclassed by other, more robust and feature-rich solutions, such as bare-metal Kubernetes or cloud Kubernetes services like Amazon Elastic Kubernetes Service (EKS) and Google Kubernetes Engine (GKE) that offer seamless integration and scalability. [9]

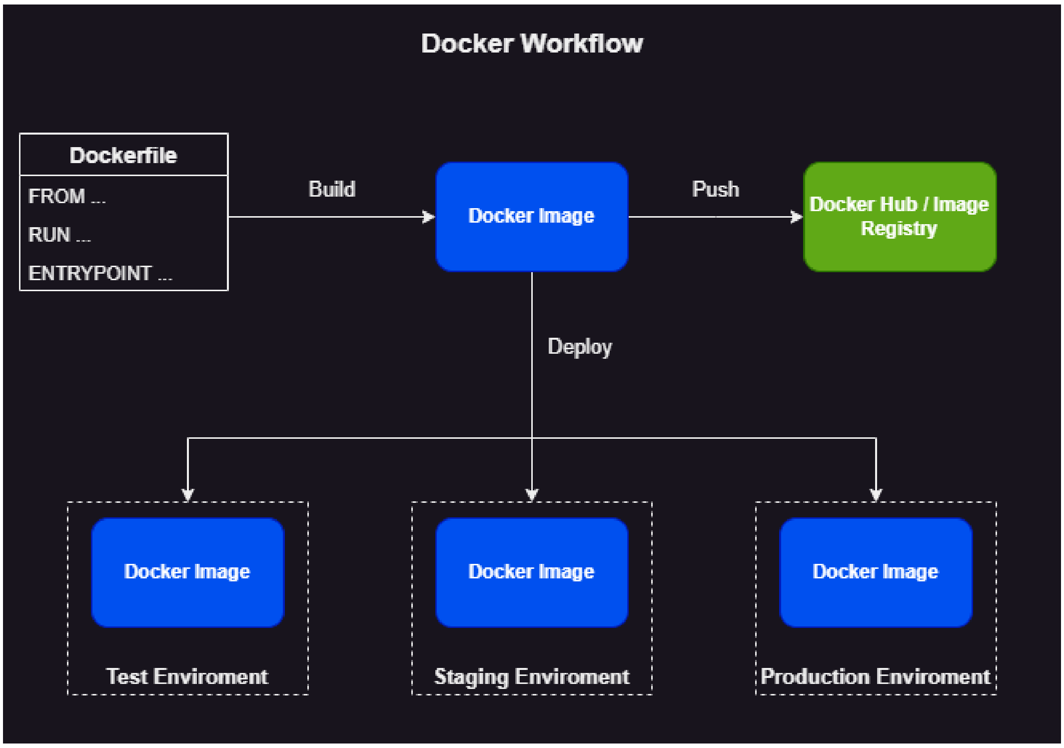

Figure 2.1: Docker Workflow

2.2 Prometheus

Prometheus is an open-source monitoring and alerting toolkit designed to provide flexible, reliable, and robust monitoring for any type of numerical data. It is considered a foundational tool for observability and a key component of modern monitoring stacks. Prometheus excels in collecting, storing, and querying time-series metrics, making it an ideal choice for monitoring infrastructure, applications, and containerized environments.

Prometheus collects and stores metrics as time-series data. Every metric is stored alongside a timestamp, allowing users to track and analyze changes over time. This feature is invaluable for identifying trends, diagnosing performance bottlenecks, and understanding long-term system behavior. Prometheus operates using a pull-based model, periodically scraping selected targets for metrics. This model ensures efficiency by fetching only the required data and enhances security by not requiring external systems to push data directly into Prometheus.

Metrics from various sources are exposed through Exporters, which transform raw data into a Prometheus-readable format. Prometheus includes a large ecosystem of pre-built exporters for common targets such as hardware systems, databases, and cloud platforms. Additionally, creating custom exporters is straightforward, making Prometheus highly adaptable to diverse use cases.

For retrieving, filtering, and manipulating time-series data, Prometheus Query Language (PromQL) is used. PromQL allows users to extract meaningful insights, create advanced visualizations, and define custom alerting rules. Prometheus also integrates seamlessly with Alertmanager, a companion tool for managing alerts. Users can define alerting rules based on PromQL expressions to trigger notifications when specific conditions, such as threshold breaches or anomalies, are met. Alertmanager routes these alerts to appropriate channels, such as email, Slack, or webhooks.

For short-lived jobs that may terminate before Prometheus can scrape their metrics, the PushGateway provides a solution. This intermediary gateway allows these jobs to push metrics temporarily, ensuring no data loss. Prometheus also supports Service Discovery, enabling automatic detection of scrape targets in dynamic environments like Kubernetes. This eliminates the need for manual configuration, making it ideal for large-scale, constantly changing systems.

Prometheus employs a highly optimized, custom-built database for storing time-series data. This database supports fine-grained retention policies, allowing users to define which metrics to retain and for how long, thereby optimizing storage usage. Prometheus further enhances usability with its ability to label metrics using key-value pairs. These labels provide additional context and facilitate filtering and aggregation, enabling multidimensional analysis of metrics.

Although Prometheus offers basic visualization capabilities, it integrates natively with Grafana, the industry-leading visualization platform. This integration allows users to build interactive, feature-rich dashboards. While Prometheus is optimized for metrics collection, it is not suitable for other types of data, such as logs. To complement Prometheus, tools like Loki are often used to handle log data, creating a comprehensive observability stack. [10, 11]

2.3 Grafana

Usually, when utilizing Prometheus to gather metrics, Grafana is employed as the visualization tool of choice. Grafana is an open-source platform for monitoring and observability, designed to enable users to visualize, query, and analyze data from an extensive range of data sources. By transforming raw metrics into actionable insights, Grafana facilitates the creation of complex, interactive dashboards, making it an essential component of most modern monitoring and observability implementations.

These dashboards are highly customizable, combining a diverse variety of visualizations, such as line graphs, heatmaps, gauges, tables, and single-stat panels. This flexibility allows teams to represent various types of information, including historical data, real-time system statuses, and long-term trends, in a visually appealing and intuitive manner.

Grafana supports integration with a broad array of data sources, including Prometheus, Loki, PostgreSQL, MySQL, and cloud services. Its ability to query multiple sources simultaneously enables advanced, multifaceted analyses, where data from disparate systems can be combined and manipulated for deeper insights.

Grafana also has support for variable creation, which allows users to dynamically assign values from the sourced data. This capability enables the construction of dynamic dashboards with parameterized views that adapt to different environments, datasets, or conditions without requiring additional customization. Such dynamic behavior unlocks templating, where reusable dashboards can be developed, significantly enhancing efficiency and consistency across monitoring setups.

Grafana excels at centralizing observability, providing a unified view of the health and performance of systems, services, and applications. Its real-time visualizations empower teams to detect and address issues as they emerge, enabling faster response times and minimizing downtime.

At the same time, the ability to chart and analyze historical data enables the identification of recurring trends and the prediction of potential problems, thus allowing the implementation of proactive measures to mitigate risks.

Lastly, Grafana offers alerting capabilities, same as Prometheus with Alertmanager, that can push notifications on various channels when a set of conditions is met [12, 13].

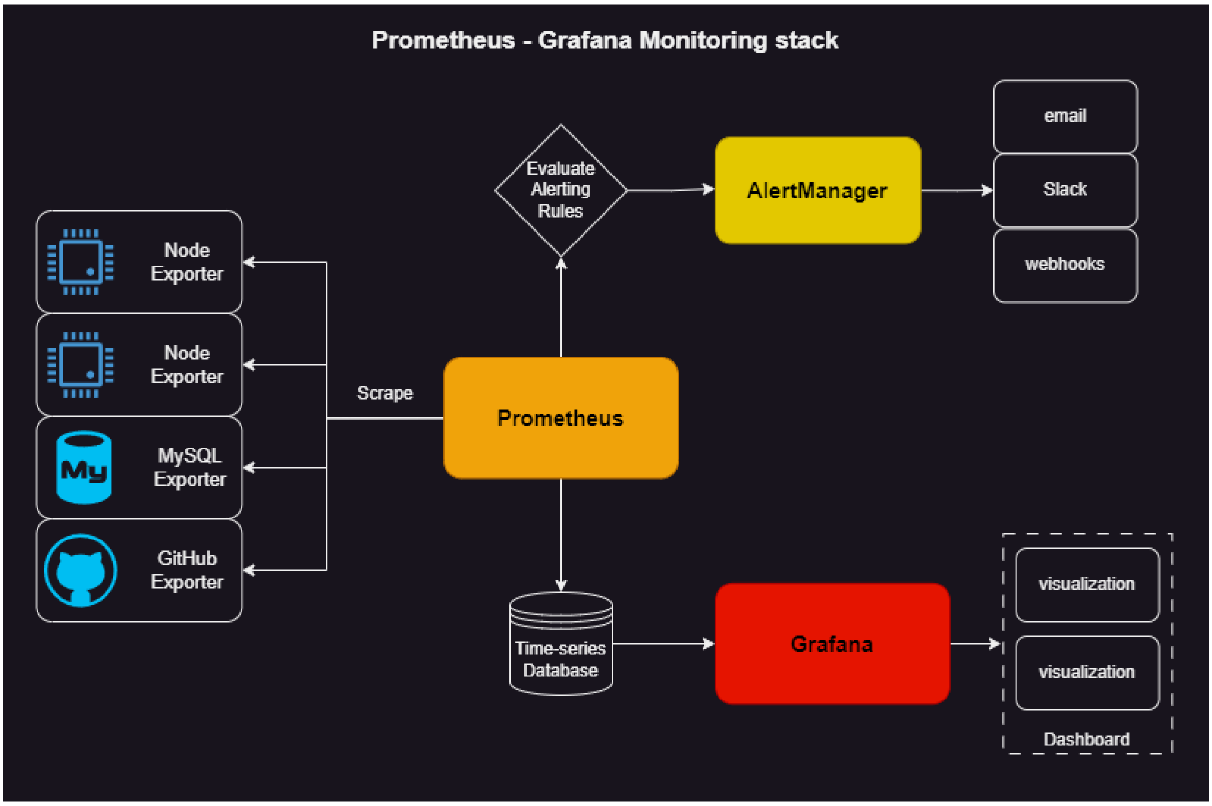

Figure 2.2: Prometheus-Grafana Monitoring stack

2.4 Message Queuing Telemetry Transport (MQTT)

Message Queuing Telemetry Transport (MQTT) is a lightweight, publish/subscribe model messaging protocol specifically designed for devices with limited computational resources and unreliable, low-bandwidth networks. Its efficiency and reliability make it a standard choice for IoT device communication and similar constrained environments.

At its core, MQTT operates using the publish/subscribe model, a decoupled communication architecture where publishers (senders) and subscribers (receivers) exchange messages through a central broker. The MQTT broker serves as the backend system responsible for managing message coordination between clients. Its key responsibilities include receiving and filtering messages, identifying clients subscribed to specific topics, and forwarding messages to those subscribers. Popular MQTT brokers are EMQX, HiveMQ and Mosquitto. Mosquitto was preferred in this implementation for it's lightweight nature, which fits well in a microservice approach.

In typical communication, MQTT clients (which can act as publishers, subscribers, or both) establish a connection with the broker using an MQTT connection. The broker confirms the connection, ensuring both entities are ready to exchange messages. MQTT requires a TCP/IP stack on both clients and brokers for communication, and clients never connect directly with each other but only with the broker. Once connected, the client can either publish messages, subscribe to specific messages, or do both. The broker filters incoming messages using topics, which are structured hierarchically, similar to folder directories in a filesystem. The broker only sends messages to subscribers from the topics they have explicitly subscribed to.

The MQTT protocol is widely regarded as a standard for IoT data transmission and for good reason. Firstly, it requires minimal hardware resources, so it can even be used by small battery-powered microcontrollers. MQTT control messages and MQTT message headers are quite small, reducing network overhead and ensuring efficient use of bandwidth. MQTT has build-in features like quick device reconnections and quality-of-service (QoS) levels, which ensure reliable message delivery even on the unreliable, low-bandwidth and high latency cellular networks IoT devices usually operate on. The decouple nature of MQTT, combined with the low bandwidth requirements allows it to easily handle large number of clients, making it very scalable.

Despite its many strengths, MQTT does have a few limitations. The broker acts as the central node, and subsequently as a single point of failure, so any disruption or failure of the broker results in a complete communication breakdown. This risk can be mitigated through high-availability setups. Also the maximum payload is 256MB and large payloads can impact performance but in an IoT scenario. However, such large payloads are uncommon. Lastly, MQTT lacks native encryption but modern authentication protocols such as OAuth and TLS can be easily integrated [14].

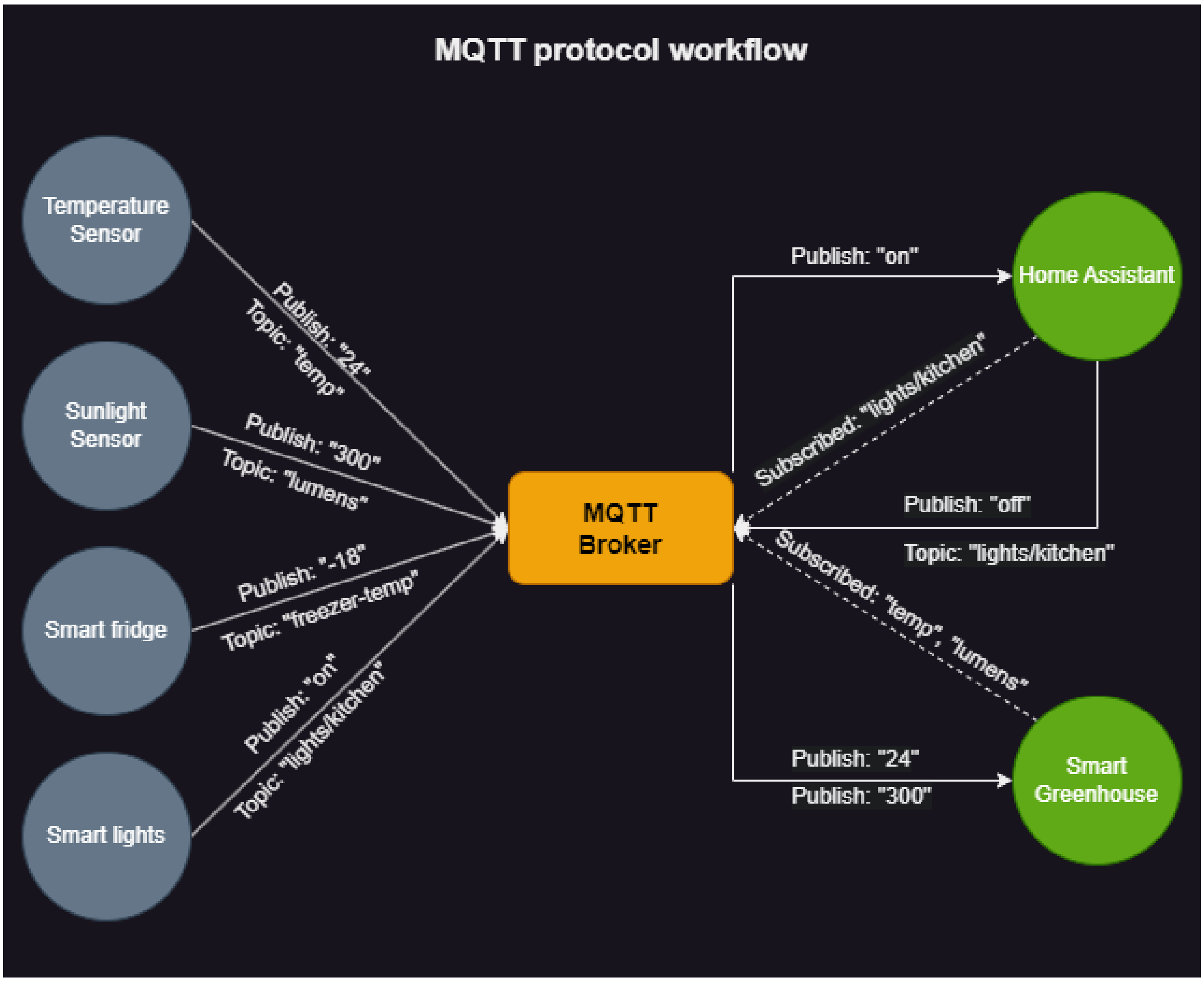

Figure 2.3: MQTT protocol workflow

3. Implementation

3.1 Design and Architecture

The design of the implementation leverages a microservices architecture to simulate real-world data collection and processing. The system was designed to emulate a sensor-driven environment, utilizing containerized services for data generation, communication, transformation, and visualization.

To simulate real sensors or microcontrollers, a Bash application was developed. This application processes a real dataset of air quality data and extracts the data corresponding to specific timestamps. The extracted data is then published to an MQTT broker. This Bash application is packaged in a Docker image to enable scalability and reusability. Multiple instances of this sensor simulation image are deployed as separate Docker containers, each simulating an independent sensor.

To enable communication using the MQTT protocol between the simulated sensors and the data-processing layer, an MQTT broker was deployed. The MQTT broker, also packaged in a Docker container, receives data from the sensor containers and forwards it to the subscribed clients.

A controller service was implemented as a Python application. This application subscribes to the MQTT topics published by the sensors, processes the incoming data, and converts it into a format compatible with Prometheus. The transformed data is then exposed through an HTTP endpoint. The controller application is also containerized and deployed as a separate Docker container.

Prometheus was deployed in a Docker container and configured to periodically scrape the data exposed by the controller service and to retain it for an appropriate amount of time.

Finally, Grafana was deployed as the visualization layer of the system, again in its own Docker container. Grafana was configured to use Prometheus as its data source. Custom dashboards were created to visualize the air quality data collected from the simulated sensors, enabling real-time monitoring and analysis.

The entire system was deployed using Docker Compose to allow for a easy orchestration and management of all the containerized services and for a straightforward, scalable and reproducible deployment.

3.2 Dataset

The dataset used contains the responses of a gas multisensor device deployes on the field in an Italian city. Hourly responses averages are recorded along with gas concentrations references from a certified analyzer. The dataset contains 9357 instances of hourly averaged responses from an array of 5 metal oxide chemical sensors embedded in an Air Quality Chemical Multisensor Device. Ground Truth hourly averaged concentrations for CO, Non Metanic Hydrocarbons, Benzene, Total Nitrogen Oxides (NOx) and Nitrogen Dioxide (NO2) were provided by a co-located reference certified analyzer.

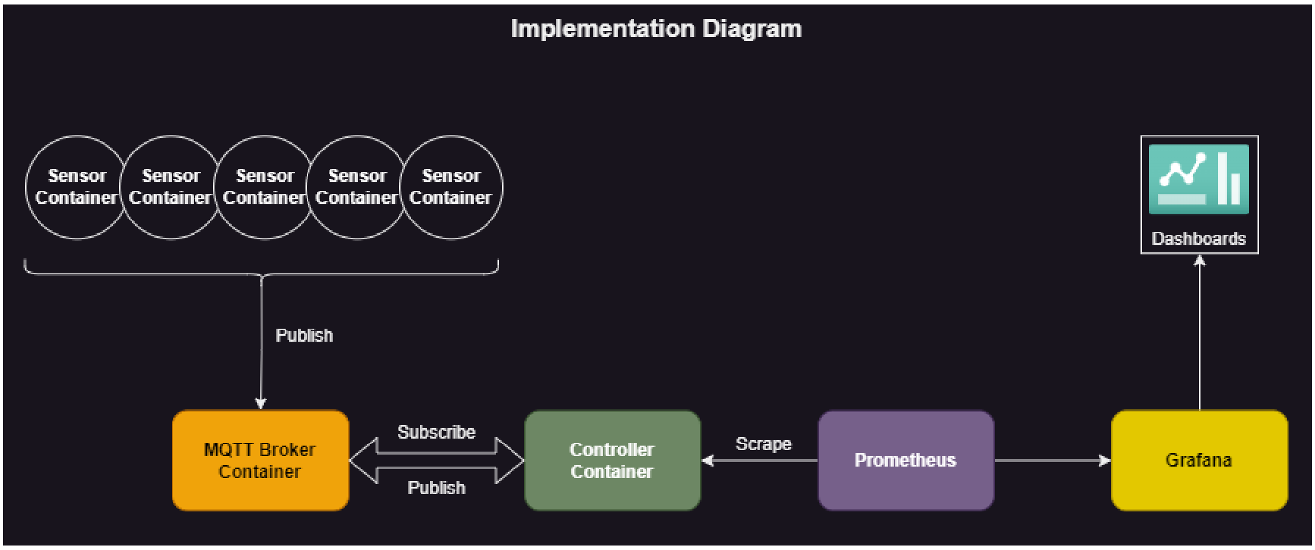

Figure 3.1: Implementation Diagram

Attribute Information	
0	Date (DD/MM/YYYY)
1	Time (HH.MM.SS)
2	True hourly averaged concentration CO in mg/m^3 (reference analyzer)
3	PT08.S1 (tin oxide) hourly averaged sensor response (nominally CO targeted)
4	True hourly averaged overall Non Metanic HydroCarbons concentration in $microg/m^3$ (reference analyzer)
5	True hourly averaged Benzene concentration in $microg/m^3$ (reference analyzer)
6	PT08.S2 (titania) hourly averaged sensor response (nominally NMHC targeted)
7	True hourly averaged NOx concentration in ppb (reference analyzer)
8	PT08.S3 (tungsten oxide) hourly averaged sensor response (nominally NOx targeted)
9	True hourly averaged NO2 concentration in $microg/m^3$ (reference analyzer)
10	PT08.S4 (tungsten oxide) hourly averaged sensor response (nominally NO2 targeted)
11	PT08.S5 (indium oxide) hourly averaged sensor response (nominally O3 targeted)
12	Temperature in °C
13	Relative Humidity (%)
14	AH Absolute Humidity

Table 3.1: Dataset Attribute Information

3.3 Sensor Simulator

3.4 MQTT Broker

3.5 Controller Node

3.6 Prometheus

3.7 Grafana

3.8 Orchestration

Bibliography

[1] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. pages 150–153, 04 2020.

[2] Ivy Wigmore Rahul Awati. What is monolithic architecture? https://www.techtarget.com/whatis/definition/monolithic-architecture, 2022.

[3] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. From monolithic systems to microservices: A comparative study of performance. *Applied Sciences*, 10(17), 2020.

[4] Martin Fowler James Lewis. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, 2014.

[5] A. Chandrinos. *Upgrading Existing Allergy Symptoms Monitoring Lab Infrastructure (AllergyMap) to Containerized Environment*. Master's thesis, Department of Computer Engineering and Informatics, University of Patras, 2021.

[6] Dan Ackerson Tomas Fernandez. When microservices are a bad idea. https://semaphoreci.com/blog/bad-microservices, 2022.

[7] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, May 2015.

[8] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), mar 2014.

[9] Sriharimalapati. Containerization with docker. https://medium.com/@sriharimalapati/containerization-with-docker-6cdc2cd5889b, 2024.

[10] Yash Jani. Unified monitoring for microservices: Implementing prometheus and grafana for scalable solutions. *Journal of Artificial Intelligence, Machine Learning and Data Science*, 2(1):848–852, March 2024.

[11] Pragathi, Hrithik Maddirala, and Sneha. Implementing an effective infrastructure monitoring solution with prometheus and grafana. *Int. J. Comput. Appl.*, 186(38):7–15, September 2024.

[12] Abhishek Singh. A data visualization tool- grafana. 01 2023.

[13] Sneha M. Pragathi B.C., Hrithik Maddirala. Implementing an effective infrastructure monitoring solution with prometheus and grafana. *International Journal of Computer Applications*, 186(38):7–15, Sep 2024.

[14] Muhammad Masdani and Denny Darlis. A comprehensive study on mqtt as a low power protocol for internet of things application. *IOP Conference Series: Materials Science and Engineering*, 434:012274, 12 2018.