Bachelor's Thesis

Service meshes like Istio

submitted by
Akbota Aitbayeva

in partial fulfilment of the requirements for the degree of
Bachelor of Science (BSc)

Vienna, 2022

Degree programme code as it appears on the student record sheet:	UA 033 521
Degree programme as it appears on the student record sheet:	Bachelor's degree programme Computer Science
Supervisor:	Univ.-Prof. Dr.Uwe Zdun
Co-Supervisor:	Amine El Malki, BSc. MBA MSc.

Abstract

This thesis aims to illustrate how service meshes can be integrated into a distributed system. Therefore, we first consider the concepts of microservices, containerization and container orchestration and understand their interrelationships. As a result, we discuss the concept of service mesh and gain insight into its purpose and potential application. We then introduce Istio, an implementation of the service mesh, and integrate it into the the microservice-based Internet of Things system.

Contents

List of Tables

List of Figures

Listings

1 Motivation

In this time of digital transformation, many businesses are using innovative technologies to increase revenue and expand their customer base. Social media, mobile technologies, cloud computing, big data and Internet of Things are key to any business seeking rapid innovation. However, these technologies pose serious problems for traditional software development. Large applications are no longer the primary focus; instead, quick-win point solutions that are tailored to the unique requirements of the business are being developed in the most agile manner. Moreover, it is preferable to maximize the possibility of replacing parts in modern architectures and to minimize the cost of replacing them [1].

Microservices and service grids are two of the concepts that have gained the most prominence in the field of systems engineering over the past few years.

A *microservices architecture* is a more granular approach to software architecture than most other approaches. A microservices-based application consists of several small microservices. Each microservice represents a single application service, which may include data management, validation, business logic, and even a user interface. For example, an e-commerce application might include separate microservices for the product catalog, customer management, and delivery process, and each microservice can be developed using its own technology stack. Microservices can be hosted in containers, virtual machines or serverless functions. These services are independent, yet interconnected through inter-service communication. The advantage of this design approach is that it can scale well without degrading overall performance, causing bottlenecks or downtime due to excessive dependencies. Consequently, this approach improves application resiliency because a fault in one service will not necessarily affect the others.

A *service mesh* is a tool that helps in the deployment and management of microservices. It provides methods for monitoring application resources so that over- or under-utilized components can be automatically reconfigured. Service meshes complement the application by providing automatic code health checks, as well as ways to manually update the application configuration when changes occur.

Microservices and service meshes offer significant advantages over traditional approaches to application design. However, there are still concerns about these approaches due to the lack of examples of their effective implementation in practice. The downside, for example, is that they may require more work to implement and maintain. From such architectural considerations, several tools have emerged that make it much easier for developers to implement them. These include containerization tools such as *Docker*, container orchestration systems such as *Kubernetes*, and service meshes such as *Istio*.

Since each design approach has its pros and cons, it is important to understand how to use these new concepts before stepping into their path. For this purpose, we examine the concepts of microservices and service meshes, implement a microservice-based Internet of Things system, and analyze how service meshes like Istio can be integrated into this distributed system.

2 Related Work

This section discusses the concepts of microservices and service meshes, providing their definitions, advantages and disadvantages, and comparing them to alternative approaches.

2.1 Microservices

The emergence of microservices architectures has radically changed the software development landscape. They have become a popular way to build applications in recent years.

In this subsection, we will explore what microservices are, how they differ from other approaches to software architecture, what are the pros and cons of using them, and how to deploy them.

2.1.1 Definition of Microservice

Microservices are a relatively new concept in the world of software development, and they are starting to gain popularity as a way to build more scalable and resilient applications.

The definition of *microservices* is a very broad and generic one. To better understand this concept, we shall consider several of them.

In a broad sense, "microservices is an architectural style or an approach for building IT systems as a set of business capabilities that are autonomous, self contained, and loosely coupled" [1].

The most favored definition is given by James Lewis and Martin Fowler:

> *"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies"* [2].

In other sources, microservices are defined as independent yet interrelated software components or services that work together to provide a complete solution [3]. They are usually independent of each other, which means that they can be developed, tested, deployed, and maintained independently of each other [3, 4]. This allows for more flexibility and granularity in scaling individual services, as well as faster deployment cycles [5]. Microservices are typically built around a business domain and provide a set of well-defined interfaces that can be used by other services. Communication between these services is typically done using lightweight protocols, often over messaging or HTTP [2, 3].

Microservice architecture is considered to a good option for developing large applications because it allows the application to be broken down into small,

independent pieces that can be developed and tested in isolation. It also helps to maintain the code base, since each service has its own code base containing all the logic to perform its specific functionality, which makes code maintenance much easier. The advantage of this architecture is that if one service fails, only that particular service will fail and not all the other services [6].

2.1.2 Microservices Architecture

Figure 1 illistrates a microservices architecture.

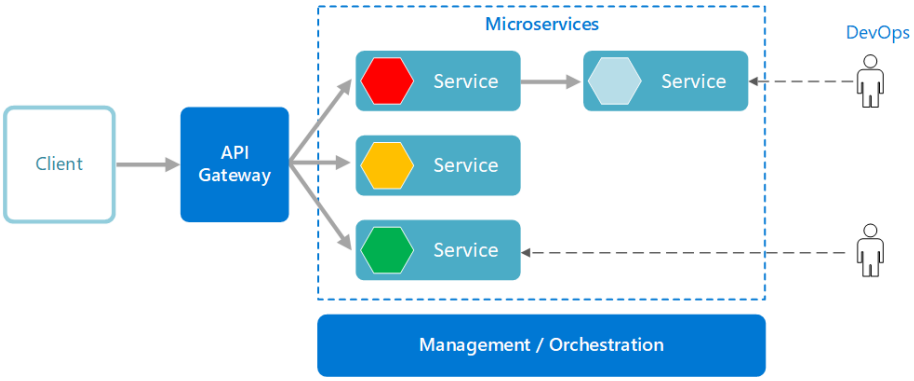

Figure 1: Microservices architecture [6].

2.1.3 Microservices Architecture vs. Monolithic Architecture

When it comes to software architecture, there are two main approaches: microservices and monolithic. Both have their pros and cons. To get a better understanding of microservices architecture, it is helpful to compare it with monolithic architecture.

Monolithic architecture is the traditional way of building software applications. In this approach, everything is built as a single unit where all parts are integrated together in one piece of code [2]. This makes it easy to develop and deploy [7], but it also increases the complexity in terms of testing, maintenance, and scaling. Changes being made to one small part of the application often lead to rebuilding and deploying the entire code. Over time, it's difficult to maintain a good modular structure. This makes changes that affect only one module in the system require resources from other modules in order to scale [2]. Moreover, it can be difficult to adopt a newer technology stack [7].

Microservices architecture is a newer approach that is gaining popularity. It is a type of software architecture in which an application is developed as a collection of services, small, autonomous, well-maintained and testable, loosely coupled, independently deployable, arranged around business capabilities, and belonging to a small team [2, 4, 6, 8]. This can be an advantage because it is

easier to scale and test a microservices application [5]. Furthermore, microservices support polyglot programming and do not need the same technology stack [6]. However, it can also be a disadvantage because it can be more difficult to develop and deploy a microservices application [5].

The main difference between monolithic and microservices architectures lies in how they handle communication between components. In monoliths, there is no clear separation between functions so every component has access to every other component's functionality. In a microservices architecture, each service communicates with others via well-defined interfaces called APIs (Application Programming Interfaces) which make them independent from each other and easier to test individually as well as collectively [2]. In addition, microservices architectures scale better than monolithic architectures due to their smaller size and the ability to deploy to multiple locations simultaneously (e.g., locally or in the cloud) [2, 5, 7]. Finally, microservices architectures simplify testing because each individual service can be tested separately from other services without affecting them (i.e., test isolation) [5].

Microservices architecture builds larger and more complex applications, while monolithic architecture is more suitable for building smaller applications [8].

Figure 2 shows both monolithic architecture and microservices architectures.

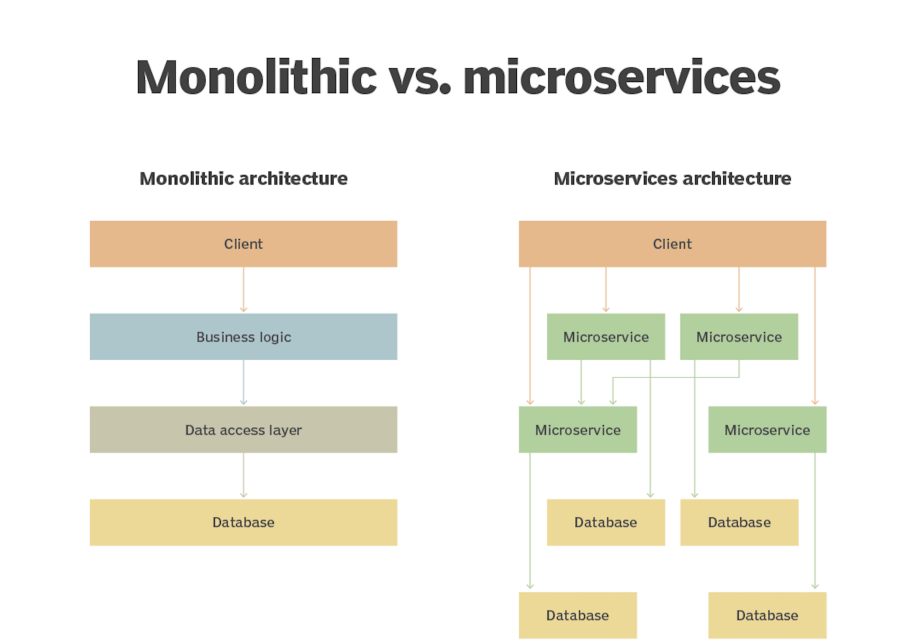

Figure 2: Microservices architecture vs. Monolithic architecture [3].

2.1.4 Benefits and Challenges of Microservices

Microservices offer many advantages, but they are also accompanied by some drawbacks. To understand whether a microservices architecture is the right choice for a particular application, one should consider both the benefits and the challenges of microservices. Table 1 contains the most common ones.

Benefits	Challenges
• Large, complex applications can be continuously delivered and deployed. • Services are small and easy to maintain. • Services are scaled independently of each other. • Services are deployed independently of each other. • Development teams can be autonomous. • New technology can be adopted easily. • Better fault isolation.	• It is difficult to find a suitable set of services. • Complexity of distributed systems makes development, testing and deployment problematic. • Deploying multiple services requires careful coordination. • Deciding when to move to a microservice architecture is a non-trivial decision. • Information barriers may arise when the number of services increases.

Table 1: Benefits and challenges of microservices [9].

2.1.5 Deploying Microservices as Containers

Microservices are often related to containers, which are solutions used to package services and their dependencies [3].

Containers are a lightweight and modern deployment mechanism that packages the code and all its dependencies [9, 10]. They use an operating system-level virtualization mechanism and consist of a single process running in isolation from other containers. Containers behave as if they were running on their own machine, which prevents port conflicts and contains an isolated environment with its own IP address. All processes, for example, can listen on port 8080. Each container has its own root file system [9].

The most popular platform for containers is *Docker* [9]. A Docker image packages a service, while a Docker container defines a service instance [11].

During container creation, it is possible to specify the CPU and memory consumed by a service instance. This is especially important when using Docker

orchestration systems such as Kubernetes, as these limits are taken into account when choosing servers to run containers, thus avoiding overloading them [9].

Kubernetes, also known as K8s, is a container orchestration system for "automating deployment, scaling, and management of containerized applications" [12]. It is an open source project that was originally developed by Google. Kubernetes allows running distributed systems resiliently. It provides features such as load balancing, service discovery, health checks, automated rollouts and rollbacks, automatic bin packing, etc. [13].

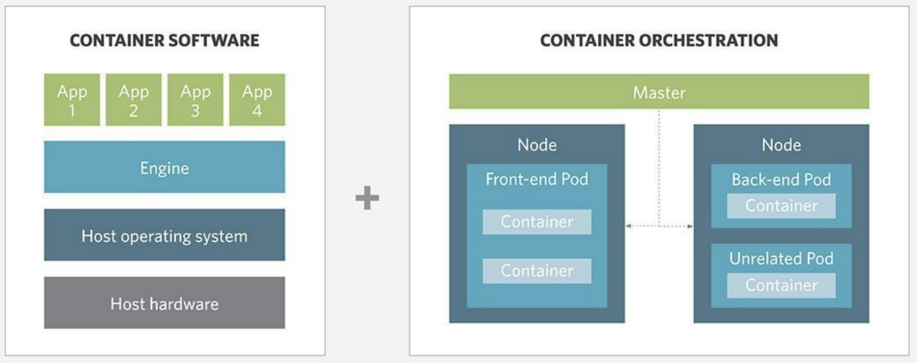

Figure 3: Deploying microservices as containers [14].

2.2 Service Meshes

Service meshes like Istio are becoming increasingly popular as they offer a solution to many microservices-related problems.

In this subsection, we will discuss what a service mesh is and how it can help manage microservices.

2.2.1 Definition of Service Mesh

A microservice has two main functions: *business logic*, which provides business functionalities, and *network logic*, which manages the inter-service communication. Business logic is an essential part of a microservice. Whereas it can be challenging for a microservice to directly perform network functions, because usually microservices are written in multiple languages and use different libraries and frameworks within the same system [15].

This has resulted in the development of *service mesh* (see Table 4), a promising solution to address these challenges by adding a specialized infrastructure layer on top of the microservices without the need to change the service implementation [16]. Furthermore, it is independent of any specific programming language or framework [17].

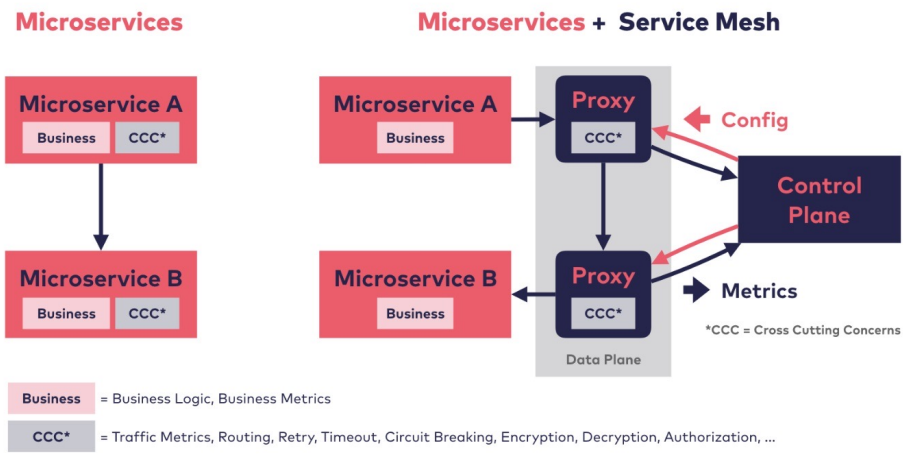

Figure 4: Service mesh [18].

Service mesh is a relatively new term and one that has come a long way [17, 19]. Nevertheless, several definitions have been already proposed to describe this term.

It is believed that the first attempt to define a service mesh was done by William Morgan [16, 19]:

> *"A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware"* [20].

Likewise, a service mesh is defined as a dedicated decentralized application-networking infrastructure layer that manages the delivery of service requests in an application and service-to-service communication over a network, enabling secure, resilient, observable and controllable applications [17, 21].

One the most comprehensive descriptions might be as follows:

> *"A service mesh is a technology that manages all service-to-service traffic within a distributed (potentially microservice-based) software system. It provides both business-focused functional operations, such as routing, and nonfunctional support, for example, enforcing security policies, quality of service, and rate limiting. It is typically (although not exclusively) implemented using sidecar proxies through which all services communicate"* [19].

Alternatively, the service mesh concept is described as an addressable infrastructure layer designed to manage both monolithic and microservice architectures. It provides the desired behavior of the network under constantly changing conditions, such as configuration, new deployments, and workload [22].

In addition, it can be understood as a pattern for managing communications between individual services in a distributed software system, which is applied to decouple the network logic from business logic so that it can be implemented and managed consistently across the whole system [19].

2.2.2 Service Mesh Architecture

Service mesh refers to an architecture composed of a *control plane* for managing the proxies as well as a *data plane* that uses them to handle network traffic for the application [15, 17].

This architecture enables the creation of significant networking capabilities for the application outside of the application and within it [17].

Figures 5 gives an overview of the service mesh architecture.

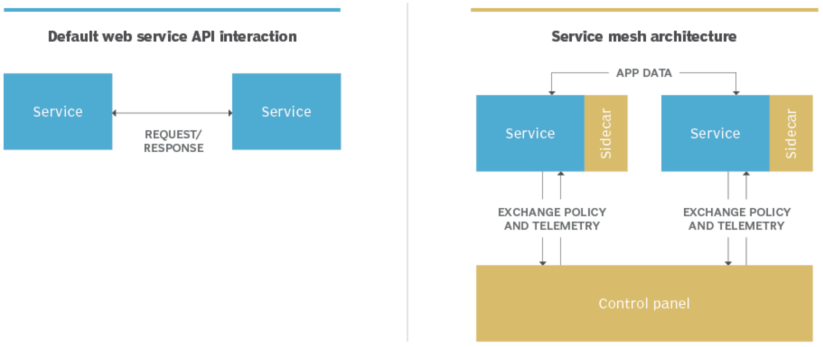

Figure 5: Service mesh architecture [21].

Figures 6 provides a description of the service mesh components. As mentioned above, most of the service meshes have only two planes: a data and control plane. A *management plane* might be used for additional intelligence [22].

2.2.3 Features of Service Meshes

A service mesh offers a set of fundamental features [19] as follows:

- *Connectivity*: traffic control, canary deployment, service discovery, timeouts, retries, etc.
- *Reliability*: fault injection, circuit breaker, etc.
- *Security*: authentication, authorization, encryption, etc.

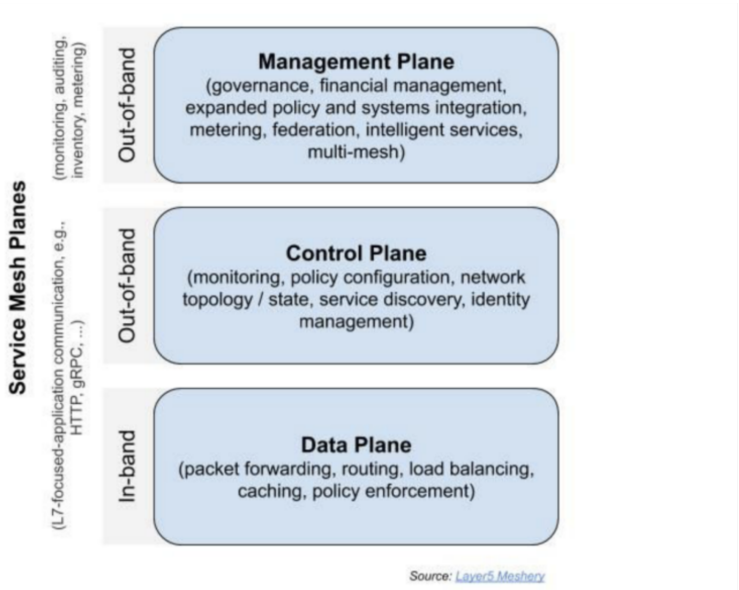

Figure 6: Service mesh components [22].

- *Observability*: service graph, traffic monitoring, metrics, etc.

2.2.4 Benefits and Challenges of Service Meshes

A service mesh brings a lot of advantages, but there are still some drawbacks (see Table 2).

Benefits	Challenges
• Simplified communication. • Controlled rollouts of new services. • Better security. • Automatic load balancing. • Faster development, testing and deployment. • Inter-language communication.	• Difficulties in configuration. • Technology's immaturity. • Increased latency. • Increased complexity.

Table 2: Benefits and challenges of service meshes [21].

2.2.5 Service Mesh Comparison

Table 3: Service mesh comparison [18, 23, 24].

	Istio	**Linkerd v2**	**Consul**
Initiated	Google, IBM, Lyft	Buoyant	HashiCorp
Announced	May 2017	September 2018	June 2016
Features	Certificate Management, Authentication, Authorization, Blue/Green Deployments, Circuit Breaking, Fault Injection, Rate Limiting, Testing, Monitoring, Multicluster, Mesh Expansion, Traffic Access Control, Traffic Specs, Traffic Split, Traffic Metrics, Load Balancing, Retry and Timeout, ect.	Certificate Management, Authentication, Authorization, Blue/Green Deployments, Fault Injection, Testing (limited), Monitoring, Traffic Split, Traffic Metrics, Load Balancing, Retry and Timeout, ect.	Certificate Management, Authentication, Authorization, Blue/Green Deployments, Circuit Breaking, Rate Limiting, Monitoring, Multicluster, Mesh Expansion, Traffic Access Control, Load Balancing, Retry and Timeout, ect.
Benefits	No other mesh can be customized and expanded like Istio. Kubernetes and other platforms can use its many functionalities.	Linkerd is created to be non-invasive while prioritizing performance and usability. As a result, adoption takes little time.	Consul does not need a scheduler as it can be used in any Consul environment. It is possible to modify and extend the proxy.
Challenges	The flexibility of Istio may be excessive for teams that do not have the capabilities to use more sophisticated technology. In addition, Istio takes control of the ingress controller.	Since Linkerd is tightly integrated with Kubernetes, it cannot yet handle workloads that are not based on Kubernetes. Also, it does not currently support data plane extensions.	Consul does not rely on Kubernetes for persistent storage; instead, it uses its own internal storage.
Complexity	High	Low	Medium

Table 3 shows that basic needs can be met by any of these service meshes. The choice depends on whether someone needs more than the bare minimum. Of these three service meshes, Istio offers the most features and flexibility; but flexibility also entails a higher level of complexity [24].

2.2.6 Istio

Istio is a service mesh implementation that was originally developed by Google, IBM and Lyft, and now has a thriving, open and diverse community that also includes representatives from Red Hat, VMWare, Solo.io, Aspen Mesh, Salesforce and many others [17].

Istio allows building resilient, secure, cloud-based native systems and solving complex issues such as security, policy management and observability with few or no changing application code [17, 25].

Even though Istio is designed for microservices or service-oriented architectures (SOA), it is not meant exclusively for them. Most businesses actually have significant investments in their current platforms and applications. The most likely scenario is that they will choose to build service architectures based on their current applications, and this is where Istio does excel. Istio allows to solve these application-networking challenges without making changes to current systems [17].

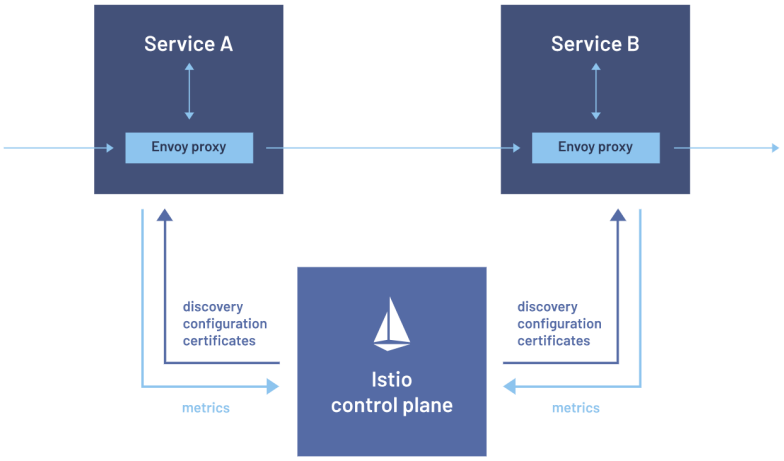

Figure 7: Istio [25].

3 Design and Implementation

This section describes the design and implementation of the Simulated Internet of Things Cloud system developed as part of the research. The functionality of the system components, the design decisions for them and how they are engineered are explained. Besides, a demonstration application showing how the system works is also presented.

3.1 Simulated Internet of Things Cloud

This subsection motivates the scope of the system and gives an overview of the entire system and the interactions between its components. Moreover, it introduces each component of the system and the technologies for its implementation.

3.1.1 Bird's-Eye View of the System

Since the focus of the research is on service meshes that enable service-to-service communication between microservices, the thesis project is based on a set of *microservices* developed using *polyglot programming* "to capture additional functionality and efficiency not available in a single language" [26]. Altogether, the microservices constitute a *Simulated Internet of Things Cloud* system.

The *Internet of things (IoT)* is the term used to describe a system of physical objects ("things") that are able to connect to the cloud and to other devices and systems over the Internet and exchange data without any human-to-human or human-to-computer interaction [27, 28, 29].

An *IoT thing* or *device* is a physical object equipped with sensors, software, and other technologies to collect and send data from the environment to the cloud for analysis. Sometimes these devices are connected to other devices and operate based on the information they receive from them. The devices perform most of their work with no human involvement, but people may interact with the devices, such as setting them up, instructing, monitoring functionality or accessing data [27].

Sensors are pivotal elements of IoT devices. In the broadest definition, an *IoT sensor* is a device that detects changes in its environment and transmits them for further processing. Its inputs may come from different sources, such as temperature, pressure, humidity, light, heat, motion or any other environmental phenomenon [30].

IoT innovations have created the ability to connect everyday objects to the network and collect data using IoT sensors, which has resulted in a wide range of IoT applications to meet many different needs.

An *IoT application* is an application that integrates data received from the IoT devices. It can be used to analyze this data for making informed decisions based on it, detect anomalies for preventing costly downtime, or predict future data for gaining insight into critical trends. Artificial intelligence (AI) and machine learning algorithms are widely used to perform these tasks. [28, 29].

The Internet of Things can be observed in numerous real-world scenarios. For example, smart homes are controlled remotely through connected lighting, heating, and electronic devices. Smart buildings can automatically regulate temperature. In agriculture, IoT sensors help control light, temperature, and humidity. Wearable devices that collect and analyze user data are used to ensure public safety during emergencies by suggesting advantageous routes to a particular location [27].

As the use of a single IoT device provides only a limited view of a situation and therefore limited results, the project deploys multiple IoT devices and demonstrates different IoT applications.

More precisely, the Simulated Internet of Things Cloud system shares sensor data from a number of IoT devices, applies fog computing to aggregate this data, and derives valuable insights from it using anomaly detection, prediction and analysis techniques.

3.1.2 System Architecture

Figure 8 illustrates the conceptual model of the system as realized by its components and how they relate to each other and to the environment.

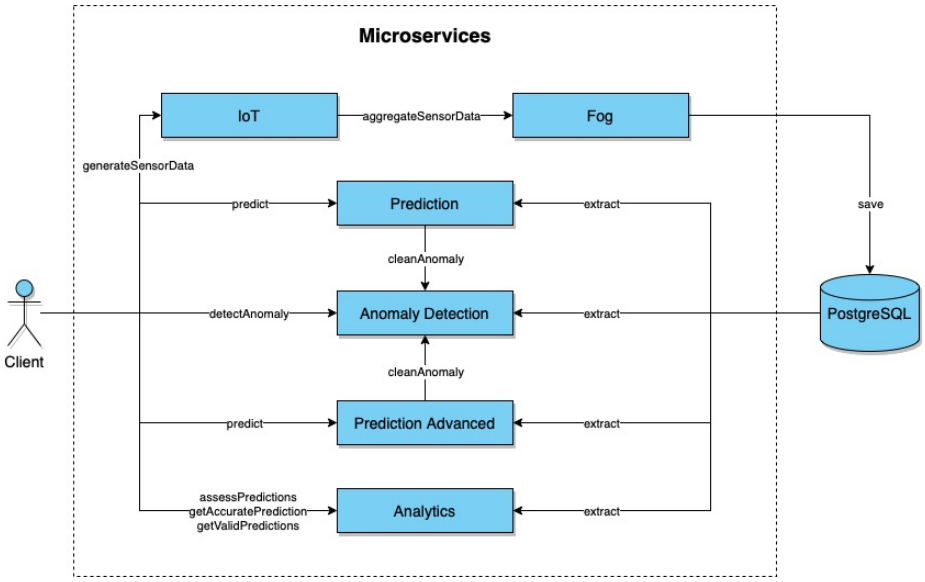

Figure 8: System architecture of Simulated Internet of Things Cloud.

3.1.3 System Components

Table 4 includes the description of the system components and technoligies used to implement them.

Component	Description	Technology Stack
Application Runner	Runs data generation and simulates common client activity.	Python 3.10 [1], Faker [2]
API Gateway	Redirects requests from clients to microservices.	Python 3.10 [1], Flask [3]
Microservice IoT	Simulates IoT devices by generating and publishing sensor data.	Java 11 [4], Apache Maven [5], Spring Boot [6], Project Lombok [7]
Microservice Fog	Aggregates sensor data received from IoT devices and saves them to the database.	Java 11 [4], Apache Maven [5], Spring Boot [6], Project Lombok [7]
Database PostgreSQL	Stores aggregated data.	PostgreSQL [8]
Microservice Anomaly Detection	Detects anomalies of stored data and cleans data outliers.	Python 3.10 [1], Flask [3], Anomaly Detection Toolkit (ADTK) [9]
Microservice Prediction	Make predictions on stored data.	Python 3.10 [1], Flask [3]
Microservice Prediction Advanced	Make predictions on stored data.	Python 3.10 [1], Flask [3], Darts [10], Prophet [11]
Microservice Analytics	Evaluates predictions.	Python 3.10 [1], Flask [3], Scikit-learn [12]

Table 4: System components.

[1] https://www.python.org
[2] https://faker.readthedocs.io/
[3] https://flask.palletsprojects.com/
[4] https://openjdk.org/projects/jdk/11/
[5] https://maven.apache.org
[6] https://spring.io/projects/spring-boot
[7] https://projectlombok.org
[8] https://www.postgresql.org
[9] https://adtk.readthedocs.io/en/stable/
[10] https://unit8co.github.io/darts/
[11] https://facebook.github.io/prophet/
[12] https://scikit-learn.org/stable/

3.1.4 System Sequence Diagram

Figure 9 shows possible interactions between the client of the system and the system, as well as between the components in the system.

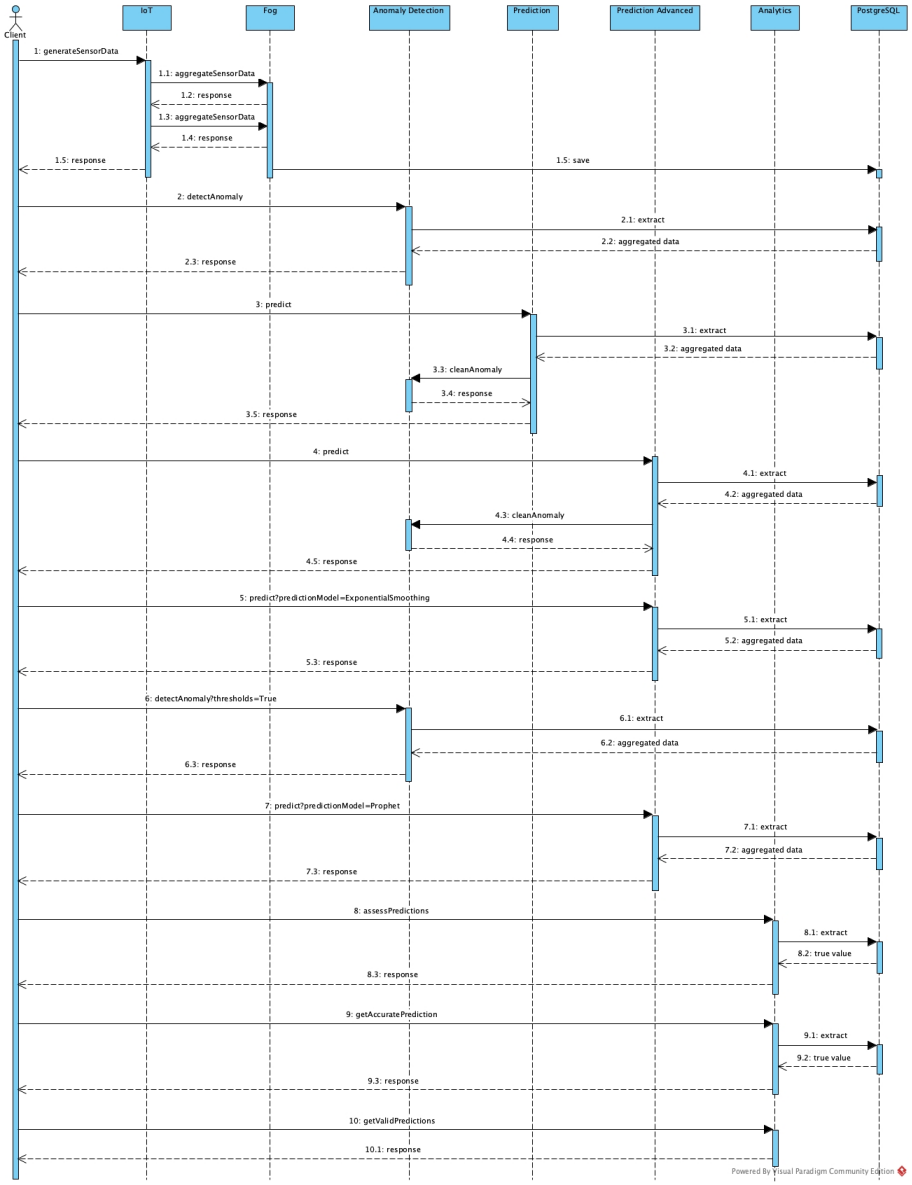

Figure 9: System sequence diagram of Simulated Internet of Things Cloud.

3.2 Application Runner

The Runner is a demonstration application that runs data generation, simulates common client activity and handles request/response communication between microservices while serving the client.

It starts data generation of both types: temperature and pressure within 1800000 milliseconds (30 minutes). And while data is being generated, it performs anomaly detection, prediction and analysis tasks.

To execute the aforementioned tasks concurrently and asynchronously, a thread pool software design pattern is used [31]. It is implemented using the `ThreadPoolExecutor` subclass in the `concurrent.futures` module [32].

Figure 10 provides a rough illustration of the performed tasks.

To stop the execution of the tasks, the application must be stopped.

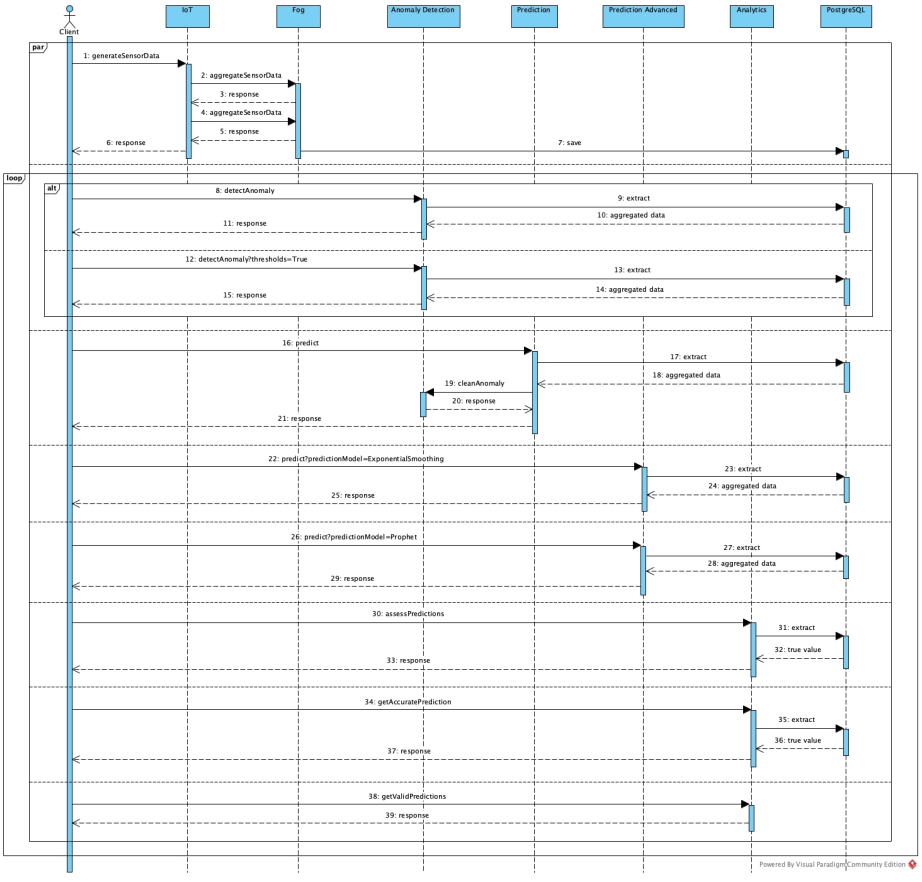

Figure 10: Sequence diagram of Runner application.

3.3 API Gateway

The *API Gateway* is a pattern in microservices architecture that manages communication between clients and microservices. It is a service that acts as an intermediary, forwarding requests from clients to microservices and returning responses from microservices to clients [33].

If no API-gateway is specified, clients must send requests directly to microservices, and this causes the following problems. Clients are connected to internal microservices. When internal microservices are developed and refactored, maintenance is affected, and client applications have to be updated frequently. All microservices must be exposed, which increases the attack surface. It is necessary for every publicly published microservice to address issues such as authorization and SSL [33].

Using an API Gateway in a microservices architecture has many advantages [33, 34]:

- It can help reduce the complexity of the entire system by providing a single entry point into the microservices system. This isolates clients from how the system is divided into microservices.
- It can provide features such as security, load balancing, and caching that would otherwise have to be implemented in each individual microservice. This can help improve system performance and make the system more resilient to failure.
- It can help manage microservices more efficiently by providing tools for monitoring and logging

However, the use of an API Gateway has some potential drawbacks [33, 34]:

- If implemented incorrectly, the API Gateway can become a bottleneck in the system.
- It can introduce additional development cost and maintenance.
- The presence of an API Gateway usually increases the response time as an additional call comes into the network.

Table 5 contains all the endpoints defined in the API Gateway. The descriptions for them are the same as for microservices endpoints.

Endpoints	Description
/api/generateSensorData	See Table 6
/api/detectAnomaly	See Table 8
/api/predict	See Tables 11, 12
/api/assessPredictions	See Table 13
/api/getAccuratePrediction	See Table 13
/api/getValidPredictions	See Table 13

Table 5: API Gateway.

3.4 Microservice IoT

The IoT microservice simulates different IoT devices by generating and publishing sensor data.

3.4.1 Design Decisions

Each IoT device has one sensor that generates a unique type of data — *temperature* or *pressure*. This is achieved by providing a single implementation and multiple instances (SIMI). The differences in the devices are configured inside the `application.properties` files and via API calls.

According to a configured sensor data type, an IoT device simulation strategy is injected to generate and publish a stream of simulated sensor data on an hourly basis. Standard sensor values, deviation from them, anomaly sensor values and their frequency are defined to simulate reasonable sensor data.

The generated data has to be sent to the Fog.

3.4.2 Implementation

After starting a microservice instance, an HTTP request got by `RestController` configures a sensor to be simulated by the *sensor data type* and the *sensor ID* and initiates sensor data generation starting from 1 January 1971 within provided request duration. If the data generation is completed sucessfully, the HTTP 200 OK success status is returned. The instance remembers the date of the last generated sensor data as a generator start day for the future requests.

The `IoTController` class injects the simulation strategy of the IoT device according to the configured sensor data type, generates and publishes a stream of simulated sensor values hourly.

To make the realization of the business logic more efficient and meet the "open-closed principle", the *Strategy* and *Factory* design patterns using dependency injection are applied.

The *Factory* design pattern is a creational pattern that uses factory methods to solve the problem of creating objects without having to specify the exact class of the object being created. This is accomplished by invoking a factory method rather than a constructor when creating objects. Factory methods can either be defined in an interface and implemented by concrete classes, or they can be implemented in a base class and optionally overridden by derived classes [35]. In other words, factory methods eliminate the need to bind application-specific classes into the code. The code only deals with the interface; therefore it can work with any user-defined concrete classes [36].

The *Strategy* design pattern is a behavioral pattern that enables choosing an algorithm at runtime. Code receives run-time instructions on which algorithm to use rather than implementing a particular algorithm directly [37]. This pattern is used when there is a need to configure a class with one of several behaviors [36].

Since the IoT microservice is build with *Spring Boot*, it is more convenient to implement design patterns using its annotations for dependency injection.

Each simulator strategy is extracted to a separate class: `PressureSimulator` (see Listing 1) and `TemperatureSimulator` (see Listing 2).

```java
@Component
public class PressureSimulator implements ISimulator {
    @Override
    public double getRandomAverageValueForOneMonth(Date currentDate
    ) {
        // implementation
    }

    @Override
    public double getRandomAverageValueForOneDay(double
    avgValueForCurrentMonth) {
        // implementation
    }

    @Override
    public SensorData simulate(Sensor sensor, double
    avgValueForOneDay, long currentHourInMs) {
        // implementation
    }

    @Override
    public ESensor getSimulatorName() {
        return ESensor.PRESSURE;
    }
}
```

Listing 1: Pressure simulator strategy class.

```java
@Component
public class TemperatureSimulator implements ISimulator {
    @Override
    public double getRandomAverageValueForOneMonth(Date currentDate
    ) {
        // implementation
    }

    @Override
    public double getRandomAverageValueForOneDay(double
    avgValueForCurrentMonth) {
        // implementation
    }

    @Override
    public SensorData simulate(Sensor sensor, double
    avgValueForOneDay, long currentHourInMs) {
        // implementation
    }

    @Override
    public ESensor getSimulatorName() {
        return ESensor.TEMPERATURE;
    }
}
```

Listing 2: Temperature simulator strategy class.

Each strategy is identified using a simulator name defined by an Enum Type (see Listing 3).

```
public enum ESensor {
    TEMPERATURE,
    PRESSURE
}
```

Listing 3: Enum for simulator names.

Concrete simulator classes `PressureSimulator` and `TemperatureSimulator` implement abstract methods defined by the `ISimulator` interface (see Listing 4).

```
@Service
public interface ISimulator {

    double getRandomAverageValueForOneMonth(Date currentDate);

    double getRandomAverageValueForOneDay(double
    avgValueForCurrentMonth);

    SensorData simulate(Sensor sensor, double avgValueForOneDay,
    long currentHourInMs);

    ESensor getSimulatorName();

}
```

Listing 4: Interface for simulator strategy.

There is no need to select a particular strategy in advance; it is sufficient to work through the mentioned interface and the `SimulatorFactory` class (see Listing 5), which injects all available strategies in the factory.

```
@Component
public class SimulatorFactory {

    private Map<ESensor, ISimulator> simulators;

    @Autowired
    public SimulatorFactory(Set<ISimulator> strategySet) {
        createSimulator(strategySet);
    }

    private void createSimulator(Set<ISimulator> strategySet) {
        simulators = new HashMap<>();
        strategySet.forEach(s -> simulators.put(s.getSimulatorName
    (), s));
    }

    public ISimulator findSimulator(ESensor sensorType) {
        return simulators.get(sensorType);
    }

}
```

Listing 5: Factory for simulators.

This allows to add new simulation strategies or modify existing ones without changing the code of the `IoTController`. The declared methods are used by the `IoTController` to execute a simulation strategy (see Listing 6).

```
@Component
public class IotController {

    @Autowired
    private SimulatorFactory simulatorFactory;

    public long generateData(long stopTime, long startDay) {

        // get the strategy by passing its name
        ISimulator simulator = simulatorFactory.findSimulator(
    sensor.getType());

        // call the methods defined in strategy
        simulator.getRandomAverageValueForOneMonth(currentDate);
        simulator.getRandomAverageValueForOneDay(avgValueForMonth);
        SensorData sensorData = simulator.simulate(sensor,
    randomValueForOneDay, currentHourInMs);

    }

}
```

Listing 6: Controller of IoT.

The `GeneratorParameters` class provides parameters used to generate reasonable sensor data. As average pressure is taken the value 1013.25, deviation from it is 15 for one day and 3 for one hour, anomaly frequency is 50. Average temperature is specified for each month (see Listing 7), deviation from it is 10 for one day and 1 for one hour, anomaly frequency is 30. Anomaly sensor values added to the standard values can be either -100 or 100.

```
public static final Map<Integer, Double>
    MONTH_TO_AVERAGE_TEMPERATURE = new HashMap<>() {{
    put(0, -30.0);
    put(1, -20.0);
    put(2, -10.0);
    put(3, 10.0);
    put(4, 20.0);
    put(5, 25.0);
    put(6, 35.0);
    put(7, 30.0);
    put(8, 15.0);
    put(9, 0.0);
    put(10, -15.0);
    put(11, -20.0);
}};
```

Listing 7: Average temperature specified for each month.

The `IotNetwork` class is responsible for sending the generated data to Fog. The `IotApplicationTests` contains unit tests.

Note: The code in listings is simplified for the better understanding of the described concepts.

3.4.3 Class Diagram

Figure 11 explains the structure of the IoT microservice, showing the defined classes, their attributes, operations and the relationships between objects.

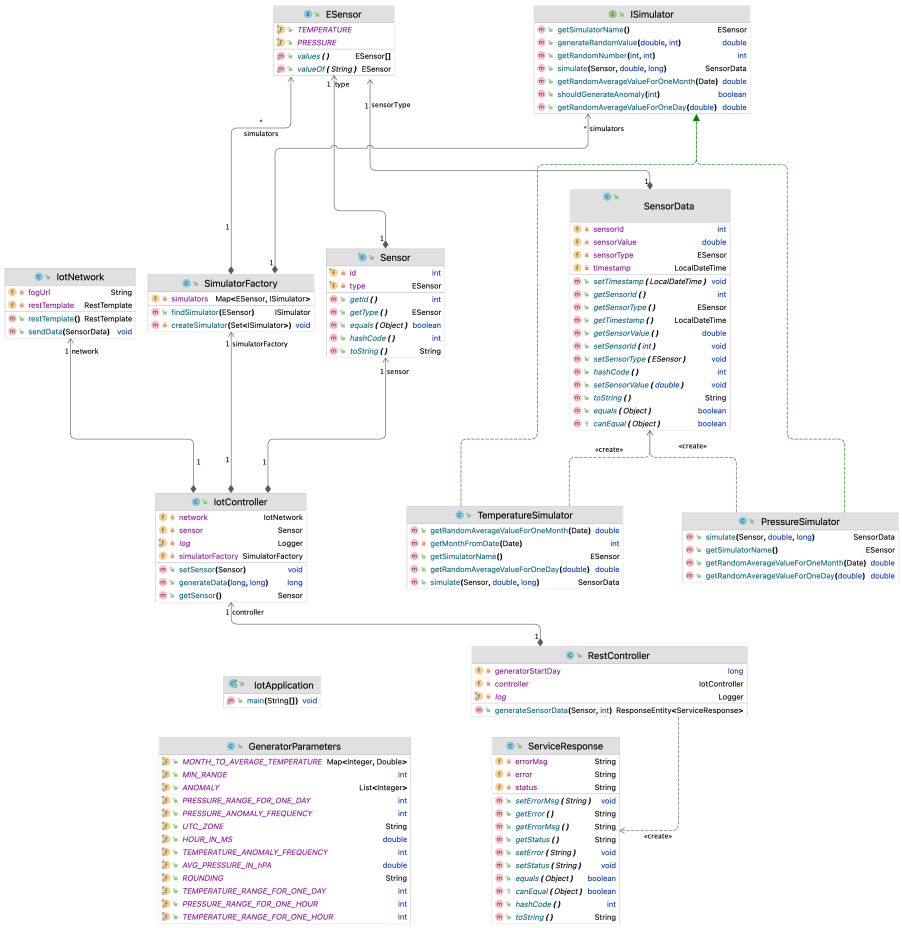

Figure 11: Class diagram of IoT microservice.

3.4.4 API Reference

Table 6: IoT API.

Generate sensor data	
Description	Configures the sensor data type and the sensor ID. Initiates sensor data generation within provided request duration. Remembers the date of the last generated sensor data for the future requests.
Endpoint	/v1/generateSensorData
Method	POST
Parameters	• requestDuration: Duration of data generation. Type: int
Request Body Example	See Listing 8.
Request Body Description	• id: An identifier of the sensor to be simulated. Type: number • type: A data type of the sensor to be simulated. Values: "PRESSURE", "TEMPERATURE". Type: string
Response Body Example	See Listing 9.
Response Body Description	• error: An error code. Type: string • errorMsg: An error description. Type: string • status: An error status. Type: string
Returns	A 200 OK status, if data generation is completed successfully.
Errors	• Connection refused: If connection to Fog is refused, then an error message will be returned. Status: 500 Internal Server Error

```
{
    "id": 1,
    "type": "PRESSURE"
}
```

Listing 8: Request body example of “Generate sensor data”.

```
{
    "error": "",
    "errorMsg": "",
    "status": "OK"
}
```

Listing 9: Response body example of “Generate sensor data”.

3.5 Microservice Fog

The Fog microservice is used to avoid the high network load caused by the deployment of multiple IoT devices by applying fog computing.

3.5.1 Design Decisions

The term “fog computing” was coined by Cisco Systems as a new model for facilitating wireless data transfer over the Internet of Things (IoT). They define *Fog Computing* “as a paradigm that extends cloud computing and services to the edge of the network” [38]. The Fog resides between the data source and the cloud and provides data, compute, storage and networking resources to end users [38, 39].

This approach meets the needs of the IoT — the vast amount of data that IoT devices generate and which would be challenging and time-consuming to send to the cloud for processing. Specifically, large amounts of data put a strain on network bandwidth and creating large data centers for storing and managing this data is expensive. Fog computing reduces the bandwidth required, eliminates bottlenecks, provides the ability to analyze data locally, decides which data should be forwarded to a centralized cloud and improves security [38, 39, 40].

In our research, fog computing performs an aggregation of sensor values received from IoT devices before storing and analysing them further.

Each unique data type, either *temperature* or *pressure*, is handled by a different Fog instance. This is achieved by providing a single implementation and multiple instances (SIMI). The differences between the instances are specified inside the `application.properties` file. The configurable properties are *data type*, *aggregation mode*, and *aggregation interval*.

Once the Fog starts collecting sensor data, the Fog’s aggregation strategy is injected, which is determined by the configured properties. The introduced aggregation modes are the calculation of *average*, *maximum* and *minimum* sensor values. The supported aggregation intervals are *day*, *month* and *year*.

After obtaining sensor values over the entire aggregation interval, sensor data is aggregated and saved to the database.

3.5.2 Implementation

The `RestController` class controls that only sensor data corresponding to the configured data type is processed by the current Fog instance.

The aforementioned design patterns, *Strategy* and *Factory*, are applied here as well. This enables the received data to be handled in different ways.

Hence, one of the concrete strategy objects (`MaxAggregator`, `MinAggregator` or `AvgAggregator`), which implement the common interface `IAggregator`, is passed via dependency injection. The concrete strategy classes determine different aggregation modes. The creator class `AggregatorFactory` is responsible for providing a specific aggregator class to the `FogController` class. The aggregator is used then to aggregate data depending on the defined aggregation mode.

The implemented patterns encapsulate the object creation code from the rest of the application code. The object creation code can be extended without touching the client code. To add an additional aggregation mode, it is enough to create a new class and define a strategy in it that operates on the data in the desired configuration. In other words, the code will be "open for extension, but closed for modification".

To minimize the overhead and eliminate the delay of data aggregation, especially for longer aggregation intervals, a fixed thread pool is added to the `FogController` (see Listing 10).

There is always a certain number of running threads in a pool of this type. At any given time, no more than `nThreads` threads will be actively processing tasks. Tasks are submitted to the pool through an internal queue, which stores additional tasks when the number of active tasks exceeds the number of threads. Additional tasks will wait in the queue until a thread becomes available. If any thread is terminated due to a runtime failure before shutting down, a new thread will take its place [41].

A significant advantage of a fixed thread pool is the smooth degradation of the applications using it. If the application handled each HTTP request with a separate thread and the system received more requests than it could handle at one time, the application would suddenly stop responding to all requests. By limiting the number of threads created, the application would not handle HTTP requests as fast as they come in, but it would handle them as fast as the system can handle them [41].

The *Thread Pool* pattern is designed to save resources in a multithreaded application and to keep parallelism within predefined limits [42]. And at the same time it makes the application more responsive.

```
1 @Component
2 @Slf4j
3 public class FogController {
4     @Autowired
5     FogRepository fogRepository;
6
7     @Autowired
8     public DataHandler dataHandler;
```

```
9
10      private Set<SensorData> sensorDataSet;
11      private Set<LocalDate> aggregatedDates;
12      private ThreadPoolExecutor executor;
13
14      @Autowired
15      public FogController() {
16         sensorDataSet = ConcurrentHashMap.newKeySet();
17         aggregatedDates = ConcurrentHashMap.newKeySet();
18         executor = (ThreadPoolExecutor) Executors.newFixedThreadPool
        (10);
19      }
20
21      public void onDataReceived(SensorData sensorData) {
22          executor.execute(() -> aggregateData(sensorData));
23      }
24
25      private void aggregateData(SensorData sensorData) {
26          // if data was already aggregated, do not process it again
27          if (aggregatedDates.contains(sensorData.getTimestamp().
        toLocalDate())) {
28              return;
29          }
30          // find the strategy based on config
31          IAggregator aggregator = dataHandler.findAggregator();
32          this.sensorDataSet.add(sensorData);
33          // get all data for current day/month/year
34          List<SensorData> dataForInterval = dataHandler.
        getDataForCurrentInterval(sensorData, this.sensorDataSet);
35          // get number of sensors, that sent data to Fog
36          long distinctSensors = dataHandler.getDistinctSensorNumber(
        dataForInterval);
37          // get number of hours for current day/month/year
38          long hoursForInterval = dataHandler.
        getHoursForCurrentInterval(sensorData);
39
40          // if sensor data for the whole interval is received,
        aggregate and save it to database
41          if ((distinctSensors * hoursForInterval) == dataForInterval
        .size()) {
42              dataForInterval.sort(Comparator.comparing(SensorData::
        getTimestamp));
43              LocalDate timestamp = dataForInterval.get(
        dataForInterval.size() - 1).getTimestamp().toLocalDate();
44              double aggregatedValue = aggregator.aggregate(
        dataForInterval);
45              AggregatedData aggregatedData = dataHandler.
        getAggregatedData(timestamp, aggregatedValue);
46              aggregatedDates.add(timestamp);
47              dataForInterval.forEach(this.sensorDataSet::remove);
48              fogRepository.save(aggregatedData);
49              log.info("Saved to database: {}", aggregatedData);
50          }
51      }
52  }
```

Listing 10: Controller of Fog.

The fog computing is realized in the `FogController` class (see Listing 10), which calls the methods from the `DataHandler` class, containing the data handling logic according to the aggregation mode and aggregation interval.

The `FogRepository` interface is responsible for saving the aggregated data to the database. It extends the JPA specific Repository interface, *JpaRepository*. This allows *Spring Data* to find this interface, analyse the defined methods and automatically create implementation from the method names [43].

The `FogApplicationTests` contains unit tests.

3.5.3 Class Diagram

Figure 12 explains the structure of the Fog microservice, showing the defined classes, their attributes, operations and the relationships between objects.

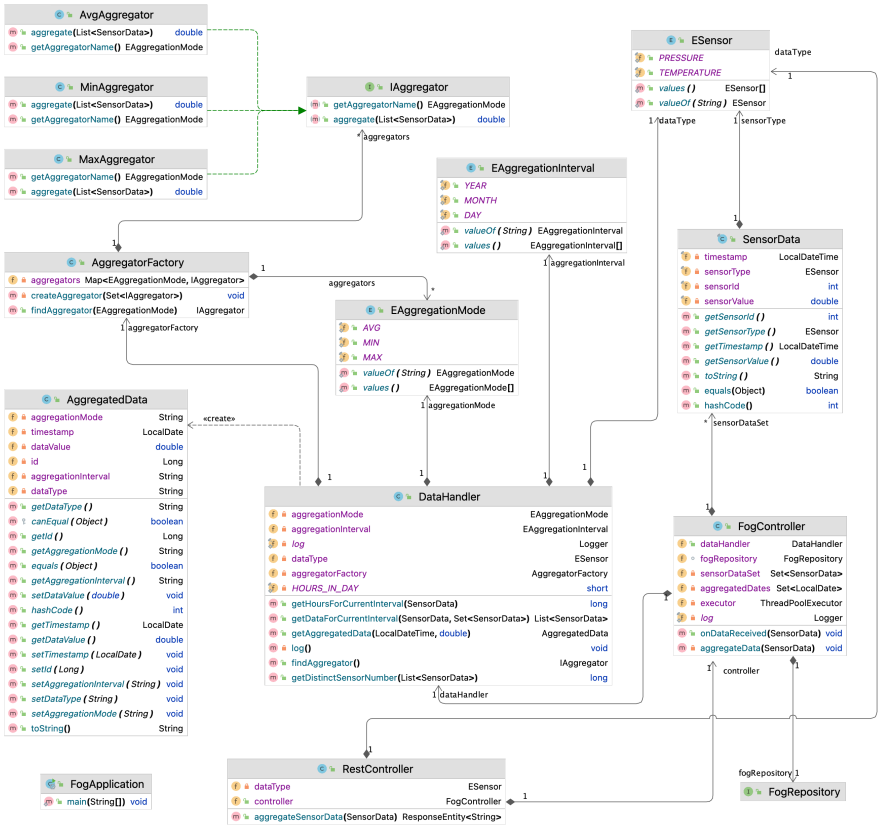

Figure 12: Class diagram of Fog microservice.

3.5.4 API Reference

Table 7: Fog API.

Aggregate sensor data	
Description	Aggregates sensor data from IoT according to a configured data type.
Endpoint	/v1/aggregateSensorData
Method	POST
Request Body Example	See Listing 11.
Request Body Description	• sensorId: An identifier of the sensor that sends data. Type: string • sensorType: A type of data generated by the sensor. Values: "PRESSURE", "TEMPERATURE". Type: string • timestamp: Time when data was generated. Type: string • sensorValue: A value generated by the sensor. Type: string
Returns	A 200 OK status, if sensor data can be aggregated.
Errors	• Incorrect data type: If a sensor data type does not match a data type configured in Fog, then an error message will be returned. Status: 400 Bad Request

```
{
    "sensorId": "1",
    "sensorType": "PRESSURE",
    "timestamp": "1971-01-17T23:00",
    "sensorValue": "1003.0686850159233"
}
```

Listing 11: Request body example of “Aggregate sensor data”.

3.6 Database PostgreSQL

The PostgreSQL database is used to store data aggregated by the Fog.

3.6.1 Entity Relationship Diagram

Figure 13 describes data stored in the PostgreSQL database.

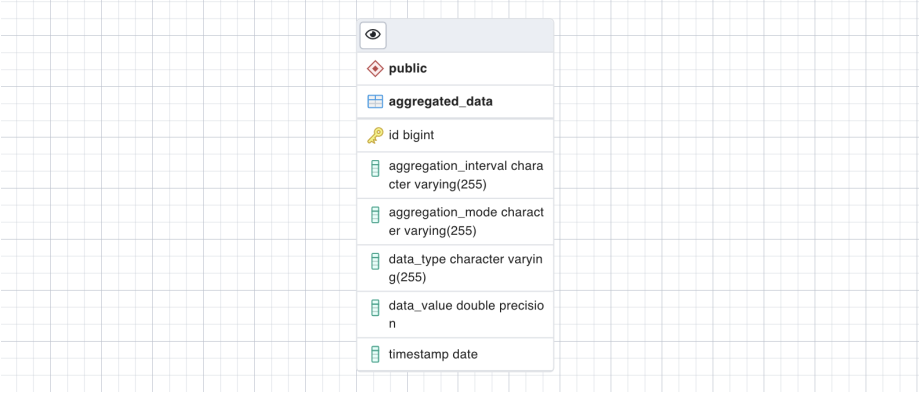

Figure 13: Entity relationship diagram of PostgreSQL database.

3.6.2 Aggregated Data

Description of data aggregated by the Fog and stored in the database:

- `aggregation_interval`: A time interval during which sensor data is aggregated. Type: string
- `aggregation_mode`: An operation chosen to aggregate sensor data. Type: string
- `data_type`: A type of sensor data. Type: string
- `data_value`: A result of aggregation. Type: number
- `timestamp`: A date indicating the end of aggregation interval. Type: string

Listing 12 contains an example of aggregated data.

```
{
    "id": 1,
    "aggregation_interval": "DAY",
    "aggregation_mode": "AVG",
    "data_type": "PRESSURE",
    "data_value": 1006.2072817329005,
    "timestamp": "1971-01-01"
}
```

Listing 12: Aggregated data example.

3.7 Microservice Anomaly Detection

The Anomaly Detection microservice performs an outlier analysis, detecting unexpected behavior, and allows system monitoring to reduce machine downtimes. It can be used to detect anomalies of stored data based on *generalized ESD test* or on *user-given thresholds* or to clean data outliers to reduce the noise.

3.7.1 Design Decisions

Anomaly detection is concerned with discovering patterns in the data that do not conform to the expected behavior. Such patterns are called *anomalies*, *outliers* or *abnormalities*. The importance of anomaly detection stems from the fact that anomalies are turned into valuable information about the anomalous behavior of the data generation process, which can be used across a broad range of applications such as business intelligence and decision making. [44, 45, 46].

The aim of anomaly detection is to locate a region of data that corresponds to normal behavior, and to label as an anomaly any data that does not belong to it. To accomplish this goal, various anomaly detection techniques have been developed. They are based on concepts from numerous scientific fields, including statistics, data mining, machine learning and information theory [44].

Usually the choice of a particular technique is determined by a great number of factors, such as the type of anomaly, the availability of a training data set with labels for normal and anomalous observations, the way anomalies are reported, and the application domain of anomaly detection [44].

First of all, it should be emphasized that the microservice performs the detection of time series anomalies. A *time series* is a series of data points indexed in successive time order [47]. It contains a set of values, which are formed as a result of continuous measurement for some time. Values belonging to consecutive timestamps do not change very dramatically or change smoothly. Consequently, sudden changes in data values are treated as anomalous [45].

Anomalies detected based on user-given thresholds can be considered as *contextual anomalies*. Thus, a specific context represented by thresholds determines whether a data value is anomalous or not [44]. For example, a temperature of 30°C is normal during the summer, but it will be identified as anomalous if it exceeds the threshold values.

Changes in values can also refer to *point anomalies*. In this case, a data value is considered abnormal if it is beyond the normal range of values [44]. This can be achieved by detecting anomalies using generalized ESD test. It is assumed that the values follow an approximately normal distribution [48]. For example, a very high temperature like 115° will be an anomaly.

Since labels indicating whether a time record is normal or anomalous are not provided in this research, *unsupervised anomaly detection* is preferred. Techniques of the unsupervised mode require no training data and are therefore most applicable [44].

As the output of anomaly detection, *binary labels* denoting if a time record is abnormal are sufficient. Binary labeling provides less information than scoring,

but it is the output that is often desired for decision making [44, 45].

The application domain of anomaly detection defines such a use case of this microservice as data cleansing. Specifically, an abnormal sensor reading can indicate a breakdown of one of its components. Such breakdowns can strongly affect predictions made based on historical data. Thus, cleaning anomalies based on generalized ESD test beforehand can help to reduce the noise level and obtain more accurate predictions.

3.7.2 Implementation

For general anomaly detection, only a data type is required. For detecting with thresholds, a data type, start and end dates, low and high values are necessary.

Both aforementioned algorithms have the same workflow. It starts by extracting the stored data according to the parameters passed by the client; an aggregation interval and aggregation mode are smartly derived from the data type provided and the aggregation performed by the Fog. If data of the given type is aggregated over different time intervals or in different modes, the first matching pair is selected.

After extracting from the database, data is preprocessed (see Listing 13). It is achieved by transforming raw data in a required format, removing duplicates and sorting time index.

```
from adtk.data import validate_series
import pandas as pd

def preprocess(df):
    df['timestamp'] = pd.to_datetime(df['timestamp'], format="%Y-%m
    -%d")
    df.set_index('timestamp', inplace=True)
    df = validate_series(df)
    return df
```

Listing 13: Data preprocessing in Anomoly Detection.

Anomaly detection techniques are realized using *Anomaly Detection Toolkit (ADTK)*, a Python package for unsupervised / rule-based time series anomaly detection [49]. For detecting anomalies an appropriate detector should be specified (see Listing 14).

```
from adtk.detector import GeneralizedESDTestAD, ThresholdAD

esd_ad = GeneralizedESDTestAD()
df_esd_ad  = esd_ad.fit_detect(df)
threshold_ad = ThresholdAD(low=low_value, high=high_value)
df_threshold_ad = threshold_ad.detect(df)
```

Listing 14: Anomaly detection.

Subsequently, the detected anomalies are returned with the anomaly detection parameters.

To clean anomalies, data to be cleaned must be received and detected for anomalies in it. Afterwards, the detected anomalies are removed from the original data, and the cleaned data is sent back.

3.7.3 Activity Diagram

Figure 14 describes the dynamic aspects of the Anomaly Detection microservice, demonstrating the logic of the anomaly detection and cleaning algorithms and describing the operations performed. Thresholds for detecting anomalies do not affect the workflow.

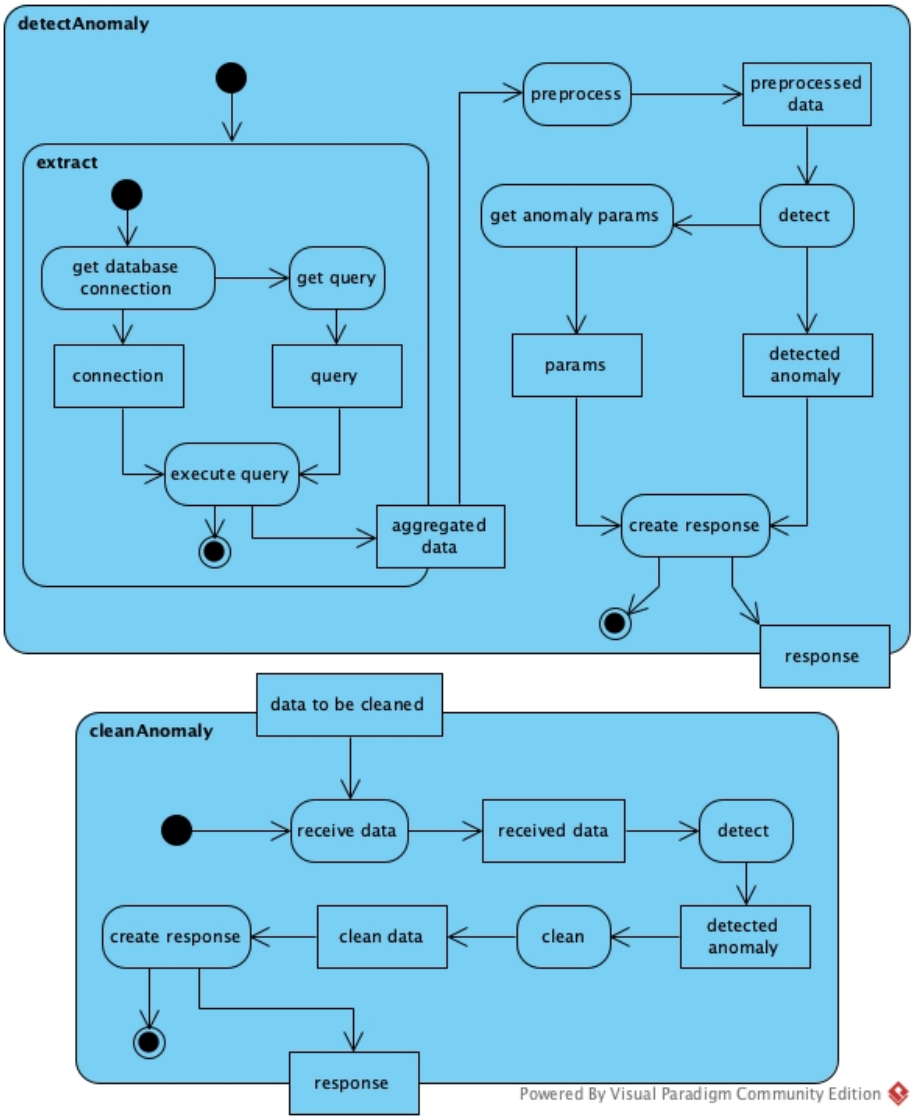

Figure 14: Activity diagram of Anomaly Detection microservice.

3.7.4 API Reference

Table 8: Anomaly Detection API.

Detect anomaly	
Description	Detect anomalies of historical data based on generalized ESD test.
Endpoint	/v1/detectAnomaly
Method	POST
Request Body Example	See Listing 15.
Request Body Description	• type: A type of data to be inspected for anomalies. Values: "PRESSURE", "TEMPERATURE". Type: string
Response Body Example	See Listing 16.
Response Body Description	• anomalies: Detected anomalous data defined by timestamp and value. Type: string • params: Parameters according to which the anomaly is detected: aggregation interval, aggregation mode and type of data, as well as anomaly count. Type: object
Returns	Detected anomalies with anomaly detection parameters and a 200 OK status if intended action is successful.
Errors	• Absence of request body elements: If one of request body elements is not passed, then an error message will be returned. Status: 400 Bad Request • Database connection failed: An error message related to the database engine or operation will be returned. Status: 400 Bad Request • No data of this type: If an invalid data type is passed, then an error message will be returned. Status: 400 Bad Request

	• Not enough data to detect: If there is no data matching the query in the database, then an error message will be returned. Status: 400 Bad Request

```
{
    "type": "PRESSURE"
}
```

Listing 15: Request body example of "Detect anomaly".

```
{
    "anomalies": "{\"37152000000\":{\"data_value
    \":920.585192002},\"40348800000\":{\"data_value
    \":1099.5327693036}}",
    "params": {
        "aggregation_interval": "DAY",
        "aggregation_mode": "AVG",
        "anomaly_count": 2,
        "data_type": "PRESSURE"
    }
}
```

Listing 16: Response body example of "Detect anomaly".

Detect anomaly with thresholds	
Description	Detect anomalies of historical data based on user-given thresholds.
Endpoint	/v1/detectAnomaly?thresholds=True
Method	POST
Parameters	• thresholds: Activation of threshold anomaly detection. Type: string
Request Body Example	See Listing 17.
Request Body Description	• type: A type of data to be inspected for anomalies. Values: "PRESSURE", "TEMPERATURE". Type: string • start_date: A left bound specifying on which date to start inspecting for anomalies. Type: string • end_date: A right bound specifying on which date to end inspecting for anomalies. Type: string

	• low_value: A threshold below which a value is regarded anomaly. Type: number • high_value: A threshold above which a value is regarded anomaly. Type: number
Response Body Example	See Listing 18.
Response Body Description	• anomalies: Detected anomalous data defined by timestamp and value. Type: string • params: Parameters according to which the anomaly is detected: aggregation interval, aggregation mode and type of data, start and end dates, low and high values, as well as anomaly count. Type: object
Returns	Detected anomalies with anomaly detection parameters and a 200 OK status if intended action is successful.
Errors	See Errors for "Detect anomaly".

```
{
    "type": "PRESSURE",
    "start_date": "1971-01-01",
    "end_date": "1971-01-31",
    "low_value": 1000,
    "high_value": 1030
}
```

Listing 17: Request body example of “Detect anomaly with thresholds”.

```
{
    "anomalies": "{\"31536000000\":{\"data_value
    \":928.829496902}}",
    "params": {
        "aggregation_interval": "DAY",
        "aggregation_mode": "AVG",
        "anomaly_count": 1,
        "data_type": "PRESSURE",
        "end_date": "1971-01-31",
        "high_value": 1030,
        "low_value": 1000,
        "start_date": "1971-01-01"
    }
}
```

Listing 18: Response body example of “Detect anomaly with thresholds”.

Clean anomaly	
Description	Clean outliers of historical data based on generalized ESD test to reduce the noise.
Endpoint	/v1/cleanAnomaly
Method	DELETE
Request Body Example	See Listing 19.
Request Body Description	• DataFrame to be cleaned and converted to a JSON format as a string. Type: string
Response Body Example	See Listing 20.
Response Body Description	• DataFrame cleaned and converted to a JSON format as a string. Type: string
Returns	Cleaned data and a 200 OK status if intended action is successful.
Errors	• Wrong type of time series object: Must be pandas DataFrame. Status: 500 Internal Server Error • Wrong type of time index object: Must be pandas DatetimeIndex. Status: 500 Internal Server Error

```
"{\"31536000000\": {\"data_value\": 928.829496902},\"31622400000\":
    {\"data_value\": 1008.7187482827},\"31708800000\": {\"
   data_value\": 1007.9199375627},\"31795200000\": {\"data_value
   \": 1014.933905023},\"31881600000\": {\"data_value\":
   1016.5428145354},\"31968000000\": {\"data_value\":
   1010.5516496529}}"
```

Listing 19: Request body example of "Clean anomaly".

```
"{\"31622400000\":{\"data_value
   \":1008.7187482827},\"31708800000\":{\"data_value
   \":1007.9199375627},\"31795200000\":{\"data_value
   \":1014.933905023},\"31881600000\":{\"data_value
   \":1016.5428145354},\"31968000000\":{\"data_value
   \":1010.5516496529}}"
```

Listing 20: Response body example of "Clean anomaly".

3.8 Microservice Prediction

The Prediction microservice performs time series forecasting on stored data.

3.8.1 Design Decisions

The importance of forecasting has never diminished. In order to provide scientific methods for forecasting, theories of probability and statistics were developed in the first place. Today, forecasting is crucial in all application domains [50]. This research concentrates on the time series domain.

Time series forecasting means to predict upcoming behavior based on time series, i.e., historical timestamped data, and, possibly, other related factors. [47, 51]. The goal in forecasting time series data is to estimate how the series of observations will continue over time [52].

There are lots of forecasting methods, often developed with specific purposes in mind. When choosing a method, it is important to consider each method's specific properties, accuracies, requirements and limitations [52].

In this version of the microservice, predictions are made using *Freedman Diaconis Estimator*, an outlier-resilient estimator that takes into account the variability of data and its size [53]. This is achieved by developing a simple algorithm that creates a histogram of historical data, slices data into bins and counts the number of observations that fall into each bin. The highest bin is the most likely one and it is used as the prediction. The bin width is determined with the Freedman Diaconis Estimator.

Many practical issues arise when using the forecasting methods, such as how to handle missing values [52], especially when a prediction must be made for some distant time in the future for which not all data are available at the time of the prediction. One way to deal with missing values is to fill them or, better yet, to predict missing values. Then based on historical and predicted values, a target prediction can be made.

Another crucial point is the availability of data, since the sufficient amount of historical data can lead to more accurate predictions. The more data available, the better. If there is not enough historical data to make the prediction, an error should be returned [54, 55].

3.8.2 Implementation

In order to make a prediction, a data type, a prediction accuracy, and a date on which the prediction is to be made are required.

The predictive algorithm starts by extracting the stored data according to the parameters passed by the client and additional parameters that are derived from the provided ones. Thus, an aggregation interval and aggregation mode are smartly obtained from the data type provided and the aggregation performed by the Fog. If data of the given type is aggregated over different time intervals or in different modes, the first matching pair is selected.

Then, based on the aggregation interval, date, and accuracy, a date range is computed for extracting historical data from the database. For data aggregated

annually, the upper bound of the range will be the last day of the previous year. For data aggregated monthly, the upper bound will be the last day of the month of the previous year. And for data aggregated daily, the upper bound will be the previous day. The lower bound of the range is calculated depending on the accuracy and the upper bound. For low accuracy, the lower bound is two years earlier than the upper bound. And for high accuracy, it is 1 January 1971.

After extracting from the database, data is preprocessed (see Listing 21). It is achieved by transforming raw data in a required format, removing duplicates and sorting time index.

```
import pandas as pd

def preprocess(df):
    df['timestamp'] = pd.to_datetime(df['timestamp'], format="%Y-%m
    -%d")
    df.drop_duplicates(subset=['timestamp'], inplace=True)
    df.set_index('timestamp', inplace=True)
    df.sort_index(inplace=True)
    return df
```

Listing 21: Data preprocessing in Prediction.

After that, data is sent to the Anomaly Detection to detect and clean outliers, which helps to make a more accurate prediction.

As discussed earlier, if there is a time lag between the historical data and the date on which the prediction is to be made, the missing values are predicted first, and then the target value can be predicted. This approach ensures that there are no prediction failures.

The future dates for missing and target values are computed depending on the aggregation interval and the date corresponding to the last available value. For data aggregated annually, the next date will be the last day of the next year. For data aggregated monthly, the next date will be the last day of the month of the next year. And for data aggregated daily, the next date will be the next day.

Listing 22 contains a code snippet that shows how a prediction is made using the Freedman Diaconis Estimator and a histogram. The `numpy.histogram` function [56] is used for this. Moreover, it can be noticed that the average value of the edges of the highest bin is used as the prediction.

```
import numpy as np
import pandas as pd

arr = df['data_value'].to_numpy()
hist, bins = np.histogram(arr, bins='fd')
predicted_value = (bins[hist.argmax()] + bins[hist.argmax()+1]) / 2
```

Listing 22: Prediction with Freedman Diaconis Estimator.

Finally, the prediction is formulated and returned with all prediction parameters. However, the date returned may differ from the given date; it depends on the aggregation interval and corresponds to the data that will be aggregated and stored in the future.

3.8.3 Activity Diagram

Figure 15 describes the dynamic aspects of the Prediction microservice, demonstrating the logic of the predictive algorithm and describing the operations performed.

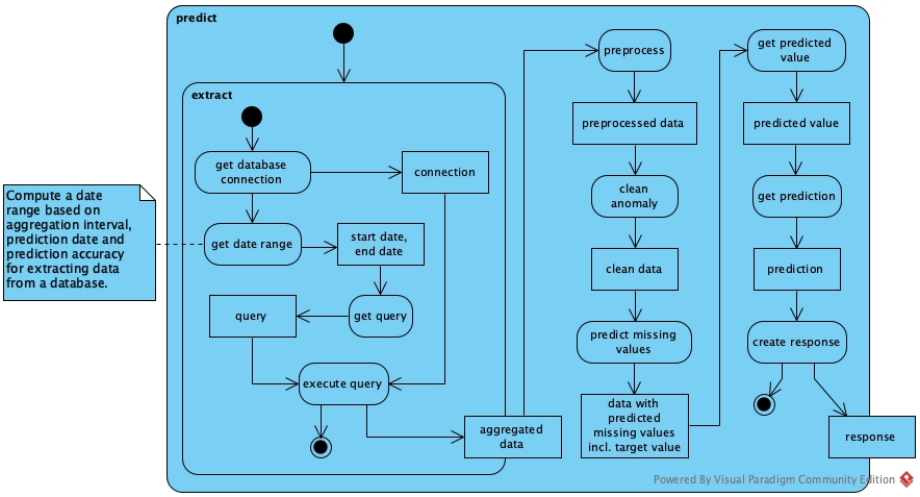

Figure 15: Activity diagram of Prediction microservice.

3.8.4 API Reference

Table 11: Prediction API.

Predict	
Description	Predict with Freedman Diaconis Estimator.
Endpoint	/v1/predict
Method	POST
Request Body Example	See Listing 23.
Request Body Description	• type: A type of data to be predicted. Values: "PRESSURE", "TEMPERATURE". Type: string • date: A date on which the prediction is made. Type: string • accuracy: Prediction accuracy. Values: "LOW", "HIGH". Type: string

Response Body Example	See Listing 24.
Response Body Description	• aggregation_interval: A time interval during which historical data is aggregated. Type: string • aggregation_mode: An operation chosen to aggregate historical data. Type: string • data_type: A type of predicted data. Type: string • date: A date on which the prediction is made. Type: string • predicted_value: A result of prediction. Type: number
Returns	A prediction with prediction parameters and a 200 OK status if intended action is successful.
Errors	• Absence of request body elements: If one of request body elements is not passed, then an error message will be returned. Status: 400 Bad Request • Database connection failed: An error message related to the database engine or operation will be returned. Status: 400 Bad Request • No data of this type: If an invalid data type is passed, then an error message will be returned. Status: 400 Bad Request • Not enough data to predict: If there is no data matching the query in the database, then an error message will be returned. Status: 400 Bad Request • HTTP error: An HTTP error occurred while cleaning anomaly. Status: 400 Bad Request

```
{
    "type": "PRESSURE",
    "date": "1971-01-07",
    "accuracy": "HIGH"
}
```

Listing 23: Request body example of “Predict”.

```
{
    "aggregation_interval": "DAY",
    "aggregation_mode": "AVG",
    "data_type": "PRESSURE",
    "date": "1971-01-07",
    "predicted_value": 1007.3210722543249
}
```

Listing 24: Response body example of “Predict”.

3.9 Microservice Prediction Advanced

The Prediction Advanced microservice is a new version of the Prediction microservice. It provides advanced algorithms to make reasonable and automated predictions on stored data.

3.9.1 Design Decisions

In this advanced version, as it is customary to compare several forecasting methods, two predictive models are added to give clients the opportunity to choose the most appropriate forecasting method, use trending or seasonal data forecasting and evaluate the methods later in the Analytics microservice.

Thus, predictions can be made with the already introduced method, which uses Freedman Diaconis Estimator and histograms, and such time series forecasting models as Exponential Smoothing or Prophet.

Exponential smoothing is a time series smoothing method. Predictions made using exponential smoothing are “weighted averages of past observations, with the weights decaying exponentially as the observations get older” [52]. To put it another way, the more recent the observation, the higher the corresponding weight. One of the advantages of this method is that it quickly provides reliable predictions, which is of great importance for application domains [52].

Prophet is a new time series forecasting model developed by Facebook’s Core Data Science team through many iterations of forecasting various data on Facebook. It is fast and fully automatic, robust to outliers and missing data [57].

Each model is itself an artificial construct based on a number of assumptions and commonly includes one or more parameters that must be estimated with the help of historical data. The choice of the optimal model depends on the availability of historical data and factors affecting its behavior, such as trend and seasonality [52]. For instance, if data does not have at least two full seasonal cycles or a well-defined frequency, it would be better not to use the these predictive models, but to stick to the simple algorithm using histograms.

3.9.2 Implementation

The predictive algorithm does not have changed significantly.

However, while using predictive models there is no need to address to the Anomaly Detection for cleaning outliers.

Furthermore, data preprocessing is extended: raw data can be transformed to other formats (see Listing 25).

```
import pandas as pd
from darts import TimeSeries

def preprocess(df, transform=None):
    df['timestamp'] = pd.to_datetime(df['timestamp'], format="%Y-%m
    -%d")
    df.drop_duplicates(subset=['timestamp'], inplace=True)
    df.sort_values(by=['timestamp'], inplace=True)
    if transform == 'TimeSeries':
        s = TimeSeries.from_dataframe(df,'timestamp','data_value')
    elif transform == 'ProphetDataFrame':
        s = df.rename(columns={'timestamp':'ds', 'data_value':'y'})
    else:
        s = df.set_index('timestamp')
    return s
```

Listing 25: Data preprocessing in Prediction Advanced.

`TimeSeries` is the main class in the `darts` module required to make predictions with Exponential Smoothing. It guarantees to have a monotically increasing time index with a well defined frequency, be non-empty, etc. It can easily be tranformed to or from other formats [58] (see Listing 26).

```
import pandas as pd
from darts import TimeSeries

ts = TimeSeries.from_dataframe(df, 'timestamp', 'data_value')
df = ts.pd_dataframe()
```

Listing 26: TimeSeries.

To make predictions with Prophet, it is required to have a dataframe with two columns: `ds` and `y`. The `ds` is a column for timestamps. The `y` column contains values of historical data. Later, the predicted values will be saved in the column `yhat` [59].

Listing 27 contains a code snippet that shows how a prediction is made using Holt-Winters' Exponential Smoothing. The forecasting model in the `darts` module is used for this [60].

```
from darts import TimeSeries
from darts.models import ExponentialSmoothing

m = ExponentialSmoothing(damped=True)
m.fit(ts)
ts = m.predict(num_steps)
predicted_value = ts.last_value()
```

Listing 27: Prediction with Exponential Smoothing.

Listing 28 contains a code snippet that shows how a prediction is made using Prophet. The `prophet` module [59] is used for this.

```
import pandas as pd
from prophet import Prophet

m = Prophet()
m.fit(df)
future = m.make_future_dataframe(periods=periods)
df = m.predict(future)
predicted_value = df['yhat'].iat[-1]
```

Listing 28: Prediction with Prophet.

Note: The code in listings is simplified for the better understanding of the described concepts.

3.9.3 Activity Diagram

Figure 16 describes the dynamic aspects of the Prediction Advanced microservice, demonstrating the logic of the predictive algorithms and describing the operations performed.

In particular, it shows the operations performed while a prediction model is being specified. For both predictive models, Exponential Smoothing and Prophet, the workflow remains the same.

See Figure 15 to see the workflow for the case where the mentioned parameter is not passed and a prediction is made using the Freedman Diaconis Estimator.

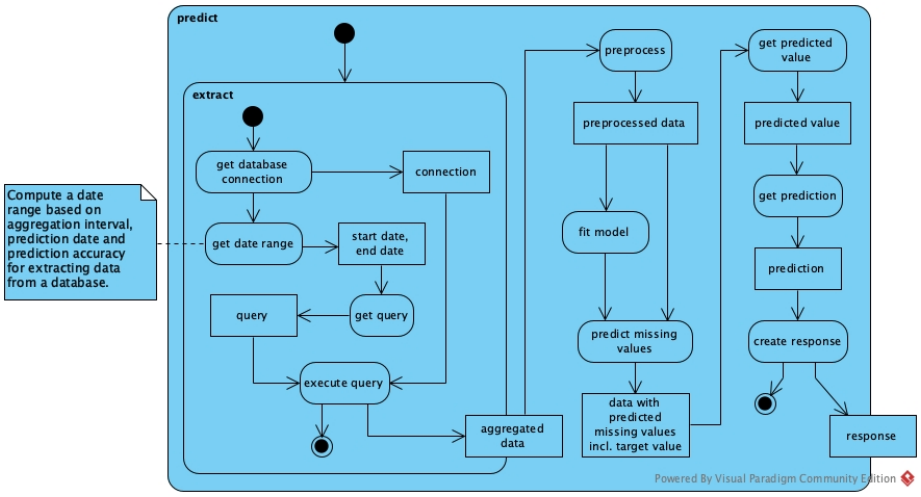

Figure 16: Activity diagram of Prediction Advanced microservice.

3.9.4 API Reference

Table 12: Prediction Advanced API.

Predict	
Description	Predict with Freedman Diaconis Estimator, Exponential Smoothing or Prophet.
Endpoint	/v1/predict
Method	POST
Parameters	• predictionModel: An optional parameter for specifying a predictive model that enables automatic predictions to be made on historical data. If the parameter is not passed or its value is incorrect, then the prediction is made using Freedman Diaconis Estimator. Values: "ExponentialSmoothing", "Prophet". Type: string
Request Body Example	See Listing 23.
Request Body Description	See Table 11.
Response Body Example	See Listing 24.
Response Body Description	See Table 11.
Returns	See Table 11.
Errors	• Model fitting error: Data set is too small or has less than two full seasonal cycles. Status: 500 Internal Server Error • Frequency error: The time index of the provided data set is missing a well-defined frequency, and the frequency could not be directly inferred. Status: 500 Internal Server Error • See Table 11 for other types of errors.

3.10 Microservice Analytics

The Analytics microservice provides an error analysis [61] of the predictions by evaluating prediction models implemented in the Prediction Advanced microservice to see how they perform compared to the default estimator presented in the Prediction microservice. It also assesses predictions made with different prediction accuracy in both prediction microservices to confirm the size of the required data set. In addition, it helps to eliminate incorrect predictions.

3.10.1 Design Decisions

Model evaluation is the process of using various evaluation metrics to understand the effectiveness of a machine learning model, along with its strengths and limitations. Model evaluation is essential for evaluating model performance in the early stages of a study, and also plays an important role in monitoring the model [62]. This evaluation is often a relative assessment of which of several alternative models is best for a particular application. There are lots of metrics by which one can assess a model [63].

Since the goal of the models used in the research is prediction, then the appropriate metric to test their prediction accuracy is the mean squared error of prediction. The *mean squared error (MSE)* is a model evaluation metric used to assess the quality of a model by measuring the mean of the squares of errors. It is the difference between the true value and the predicted value. When there is no error in the model, the MSE is zero. As model error increases, the MSE value becomes greater as well [64, 65].

3.10.2 Implementation

As the microservice receives predictions as dictionaries, first it needs to convert them into class objects and ensure that parameters of predictions match for further analysis. Next steps are performed depending on the type of analysis. Analysis is organized into three types: assessing predictions, reporting (getting) the most accurate prediction and obtaining (getting) valid predictions.

Thus, for assessing predictions and reporting the most accurate prediction, the *true value* aggregated by the Fog is required and needs to be extracted from the database. After this, predictive models can be evaluated and their errors can be estimated. The mean squared error is realized by the `mean_squared_error` function of the `sklearn.metrics` module [66]. The difference between these two types of analysis is their return value. Assessing predictions yields the estimated errors of all the submitted predictions. Whereas when reporting the most accurate prediction, all errors are also calculated, but only the prediction with the predicted value closest to the true value is returned. The calculated error is only used to determine that prediction and should not be returned.

However, for obtaining valid predictions, no true value is required. Predicted values are examined against the expected *thresholds*. Each predicted value is compared with expected thresholds, and it is identified as invalid when it is beyond the thresholds. Only valid predictions are returned.

3.10.3 Activity Diagram

Figure 17 describes the dynamic aspects of the Analytics microservice, demonstrating the logic of the analysis techniques and describing the operations performed.

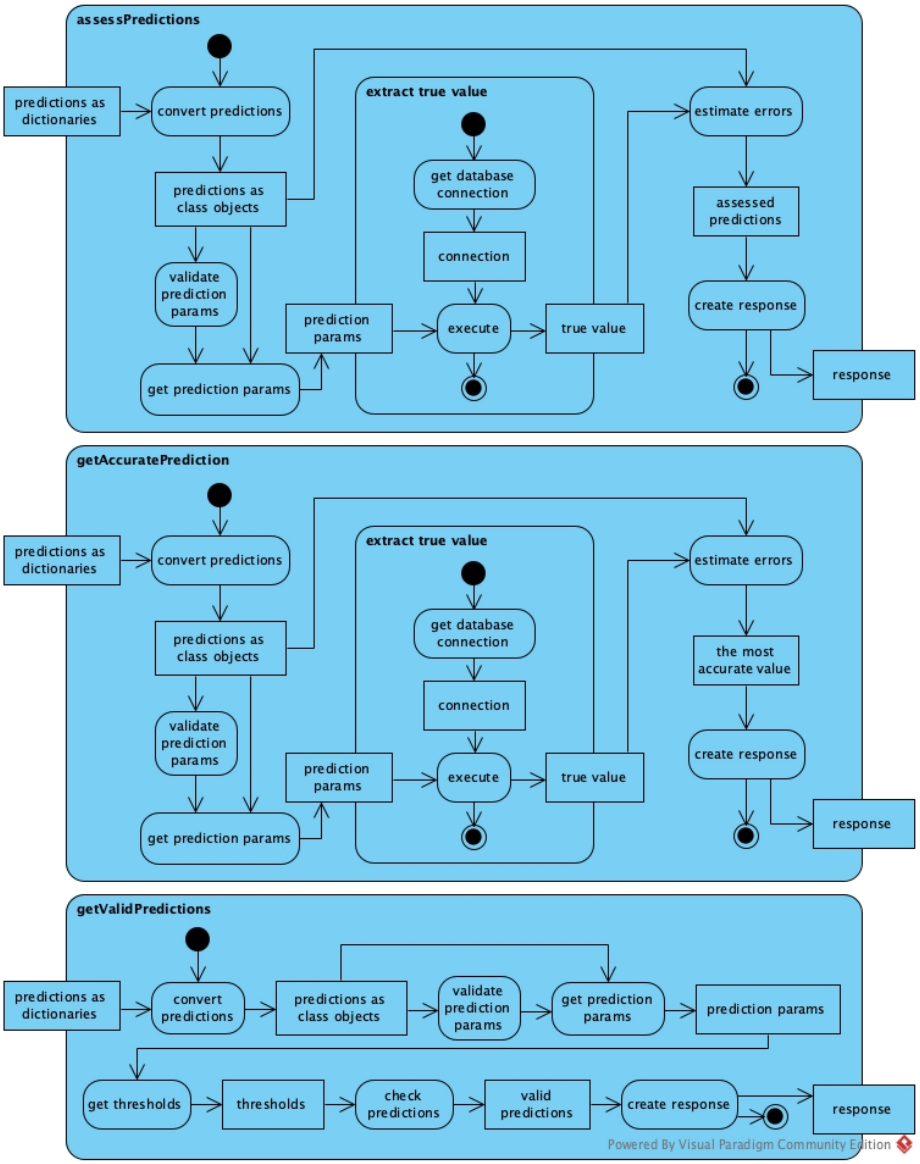

Figure 17: Activity diagram of Analytics microservice.

3.10.4 API Reference

Table 13: Analytics API.

Assess predictions	
Description	Accept predictions as dictionaries with the same parameters. Measure the mean squared error (MSE) of the predicted values. Applicable only if the true value is known.
Endpoint	/v1/assessPredictions
Method	POST
Request Body Example	See Listing 29.
Request Body Description	• predictions: Predictions to be assessed. Type: array
Response Body Example	See Listing 30.
Response Body Description	• predictions: Assessed predictions. Type: array
Returns	Assessed predictions and a 200 OK status if intended action is successful.
Errors	• Absence of request body elements: If one of request body elements is not passed, then an error message will be returned. Status: 400 Bad Request • Predictions conversion failed: An error message related to conversion predictions from dictionaries to class objects will be returned. Status: 400 Bad Request • Predictions are empty: If converted predictions are empty, then an error message will be returned. Status: 400 Bad Request • Parameters of predictions do not match: Parameters should match for further analysis. Status: 400 Bad Request

	• Database connection failed: An error message related to the database engine or operation will be returned. Status: 400 Bad Request • True value is not available: If a true value is not stored in the database, then an error message will be returned. Status: 400 Bad Request • Only integer or float is allowed: To calculate the mean squared error (MSE), a true value and predicted values should be of integer or float type. Status: 400 Bad Request


```
{
    "predictions": [
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "predicted_value": 1007.3210722543249
        },
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "predicted_value": 1015.3134745555662
        },
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "predicted_value": 1030.4862869542915
        }
    ]
}
```


Listing 29: Request body example of “Assess predictions”.

```
{
    "predictions": [
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "error": 281.14697488802955,
            "predicted_value": 1007.3210722543249
        },
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "error": 77.00125036533619,
            "predicted_value": 1015.3134745555662
        },
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "error": 40.931547540469005,
            "predicted_value": 1030.4862869542915
        }
    ]
}
```


Listing 30: Response body example of "Assess predictions".

Get accurate prediction	
Description	Accept predictions as dictionaries with the same parameters. Measure the mean squared error (MSE) of the predicted values. Applicable only if the true value is known.
Endpoint	/v1/getAccuratePrediction
Method	POST
Request Body Example	See Listing 29.
Request Body Description	• predictions: Predictions to be assessed. Type: array
Response Body Example	See Listing 31.
Response Body Description	• aggregation_interval: An aggregation time interval of the most accurate prediction. Type: string

	• aggregation_mode: An agg. mode of the most accurate prediction. Type: string • data_type: A type of the most accurate prediction. Type: string • date: A date on which the prediction is made. Type: string • predicted_value: A result of the most accurate prediction. Type: number
Returns	The most accurate prediction and a 200 OK response if intended action is successful.
Errors	See Errors for "Assess predictions".

```
{
    "aggregation_interval": "DAY",
    "aggregation_mode": "AVG",
    "data_type": "PRESSURE",
    "date": "1971-01-07",
    "predicted_value": 1030.4862869542915
}
```

Listing 31: Response body example of "Get accurate prediction".

Get valid predictions	
Description	Accept predictions as dictionaries with the same parameters. Examine predicted values against the expected thresholds. No true value is required.
Endpoint	/v1/getValidPredictions
Method	POST
Request Body Example	See Listing 29.
Request Body Description	• predictions: Predictions to be validated. Type: array
Response Body Example	See Listing 32.
Response Body Description	• predictions: Valid predictions. Type: array
Returns	Valid predictions and a 200 OK response if intended action is successful.

Errors	• Absence of request body elements: If one of request body elements is not passed, then an error message will be returned. Status: 400 Bad Request • Predictions conversion failed: An error message related to conversion predictions from dictionaries to class objects will be returned. Status: 400 Bad Request • Predictions are empty: If converted predictions are empty, then an error message will be returned. Status: 400 Bad Request • Parameters of predictions do not match: Parameters should match for further analysis. Status: 400 Bad Request • Unknown data type: If a data type is not recognized while retrieving expected thresholds, then an error message will be returned. Status: 400 Bad Request • Unknown aggregation interval: If an aggregation interval is not recognized while retrieving expected thresholds, then an error message will be returned. Status: 400 Bad Request


```
{
    "predictions": [
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "predicted_value": 1007.3210722543249
        },
        {
            "aggregation_interval": "DAY",
            "aggregation_mode": "AVG",
            "data_type": "PRESSURE",
            "date": "1971-01-07",
            "predicted_value": 1015.3134745555662
        }
    ]
}
```


Listing 32: Response body example of “Get valid predictions”.

4 Deployment

4.1 Technology Stack

Technologies used in deploying microservices: Apache Maven [5], Docker 20.10.17 [13], Kubernetes v1.24.3 [14], minikube v1.26.1 [15], kubectl v1.25.0 [16], Istio 1.15.0 [17], Prometheus [18], Kiali [19], Grafana [20]. All of them should be installed to perform the next steps.

4.2 How to Run

The deployment steps for macOS are as follows:

- Clone the project from GitLab:

  ```
  https://git01lab.cs.univie.ac.at/swa-bachelor-thesis/service-meshes-like-istio.git
  ```

- Change directory:

  ```
  cd service-meshes-like-istio
  ```

- Start minikube with sufficient resources for Istio:

  ```
  minikube start --memory=16384 --cpus=6
  ```

- Move to the Istio package directory:

  ```
  cd istio-1.15.0
  ```

- Add the `istioctl` client to the path:

  ```
  export PATH=$PWD/bin:$PATH
  ```

- Add a namespace label:

  ```
  kubectl label namespace default istio-injection=enabled
  ```

- Point the terminal to use the docker daemon inside minikube to build docker images:

  ```
  eval $(minikube docker-env)
  ```

- Change directory:

  ```
  cd docker
  ```

- Move to the directory of each microservice and build docker images:

  ```
  docker build -t bachelor/api-gateway:v1 .
  docker build -t bachelor/ms-iot:v1 .
  docker build -t bachelor/ms-fog:v1 .
  docker build -t bachelor/ms-anomaly-detection:v1 .
  ```

```
docker build -t bachelor/ms-prediction:v1 .
docker build -t bachelor/ms-prediction-advanced:v1 .
docker build -t bachelor/ms-analytics:v1 .
```

- Pull a database image:

```
docker pull postgres
```

- Check images:

```
docker images
```

- Change directory:

```
cd kubernetes
```

- Deploy all applications:

```
kubectl apply -f services/
kubectl apply -f deployments-v1/
kubectl apply -f deployments-v2/
```

- Be sure that everything is up and running:

```
kubectl get all -A
```

- Delete all applications (required for the further evaluation):

```
kubectl delete -f services/
kubectl delete -f deployments-v1/
kubectl delete -f deployments-v2/
```

- Change directory:

```
cd service-meshes-like-istio
```

- Start an external load balancer for Istio in a different terminal:

```
minikube tunnel
```

- Open Kiali in a different terminal:

```
istioctl dashboard kiali
```

- Open Grafana:

```
istioctl dashboard grafana
```

5 Evaluation and Discussion

5.1 Ingress

Istio offers a configuration model called *Gateway*, which can be used to expose a service outside of the service mesh. An ingress gateway can be configured using an Istio default gateway implementation on port 80 for HTTP traffic.

- Deploy all applications with `v1` version:

  ```
  sh apply_kubernetes_v1.sh
  ```

- Apply Istio configuration for ingress gateway and normal traffic routing to API Gateway:

  ```
  sh ingress.sh
  ```

Figure 18 shows the visualization of the service mesh in Kiali.

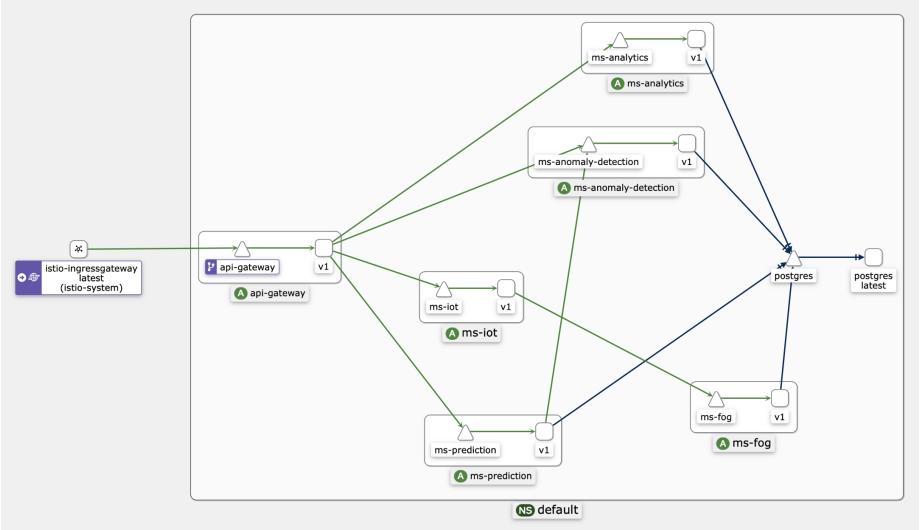

Figure 18: Istio's ingress traffic visualization in Kiali.

- Delete all Istio configurations:

  ```
  sh delete_istio.sh
  ```

5.2 Request Routing

As Figures 8, 9, 10 show, requests can be also sent directly to the microservices. Using Istio will make request routing much easier. Otherwise, every time the

microservice endpoints change, the API Gateway will need to be changed as well. In contrast, Istio allows doing this dynamically.

In addition, the API Gateway represents the only entry point into the system and can become a bottleneck. Nevertheless, it is still possible to use both the API Gateway and Istio, as shown earlier, especially if the API Gateway has more than just a request routing function.

- Apply Istio configuration for request routing:

```
sh request-routing.sh
```

Figure 19 shows dynamic request routing in Kiali.

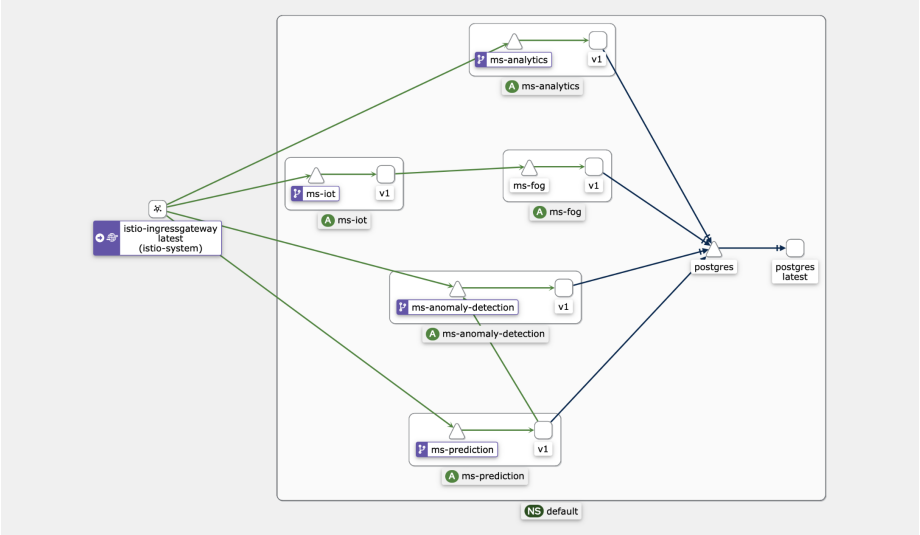

Figure 19: Istio's request routing visualization in Kiali.

- Delete all Istio configurations:

```
sh delete_istio.sh
```

5.3 Fault Injection Abort

Istio can be used to test microservice resiliency using an HTTP abort fault. Although complex systems frequently fail, the ultimate objective is to avoid a catastrophic system breakdown. Istio simplifies fault injection. A testing technique called fault injection enables the introduction of HTTP errors into a system to see if it can withstand and recover from failures. When the system experiences random failures, this straightforward idea enables analysis of the behavior of the system as a whole [67].

- Apply Istio configuration for fault injection abort:

```
sh fault-injection-abort.sh
```

An HTTP abort is introduced for the Anomaly Detection microservice. Thus, the microservice responds 500 with a 10 percent probability. But this only applies to cleaning anomalies, not detecting them.

Figure 20 shows fault injection abort in Kiali.

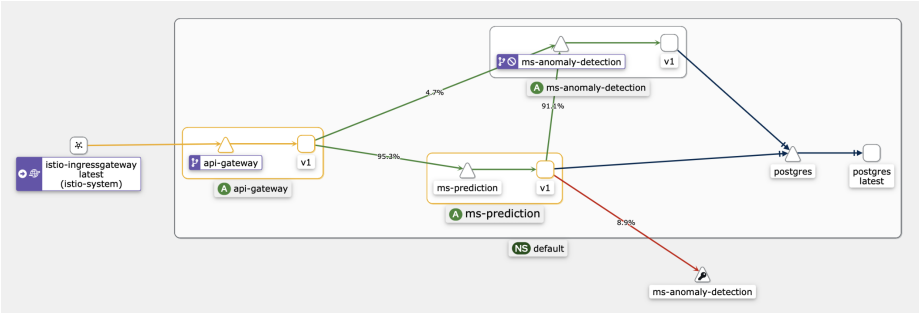

Figure 20: Istio's abort fault injection visualization in Kiali.

Figure 21 shows fault injection abort in Grafana.

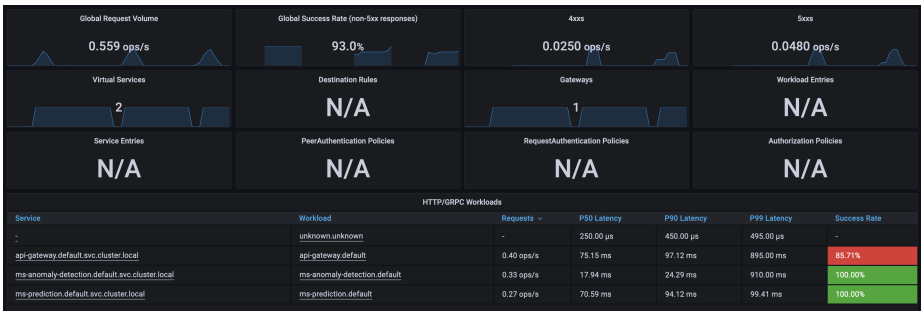

Figure 21: Istio's abort fault injection in Grafana.

- Delete all Istio configurations:

```
sh delete_istio.sh
```

5.4 Retry

Istio's retry feature allows for a few more tries before the error must be handled properly. It helps to improve service availability and application performance.

To demonstrate this feature, a fault injection abort can be used. Thus, API Gateway attempts 10 times a 2s timeout for each retry to get a prediction, if a prediction process was aborted because of the error in the Anomaly Detection which must be addressed by the Prediction.

- Apply Istio configuration for retry:

```
sh retry.sh
```

Figure 22 shows retry in Kiali.

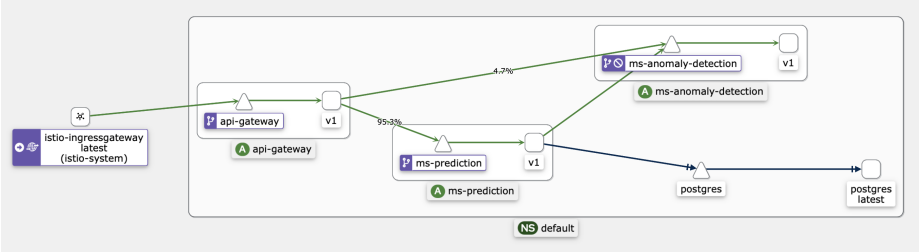

Figure 22: Istio's retry visualization in Kiali.

- Delete all Istio configurations:

```
sh delete_istio.sh
```

5.5 Load Balancing

Istio supports load balancing, if there are multiple deployments of the same service. It operates on a round-robin load balancing policy by default, which means that each service instance responds to requests one at a time.

- Deploy all applications with `v2` version:

```
sh apply_kubernetes_v2.sh
```

- Apply Istio configuration for load balancing:

```
sh load-balancing.sh
```

Figure 23 shows load balancing in Kiali.

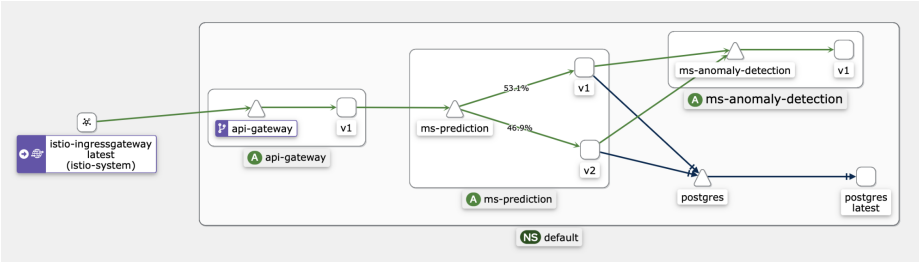

Figure 23: Istio's load balancing visualization in Kiali.

- Delete all Istio configurations:

```
sh delete_istio.sh
```

5.6 Traffic Shifting

Traffic can be shifted from one version of a microservice to another using Istio. The routing rules that route a percentage of traffic from one destination to another can be configured to achieve this. This feature can be used to provide a version rollout or canary deployment.

- Apply Istio configuration for traffic shifting:

  ```
  sh traffic-shifting-90-10.sh
  ```

  Figure 24 shows traffic shifting in Kiali.

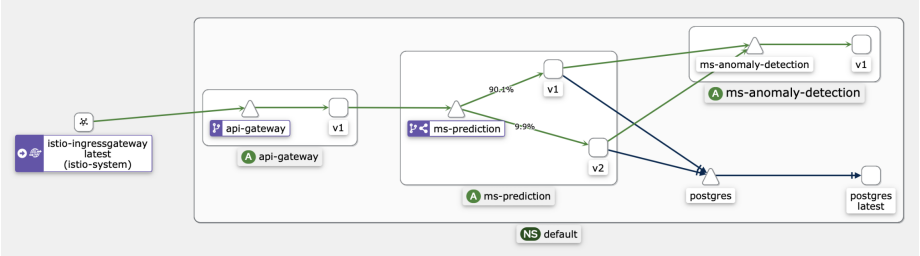

Figure 24: Istio's traffic shifting visualization in Kiali.

- Delete all Istio configurations:

  ```
  sh delete_istio.sh
  ```

5.7 Mirroring

Traffic mirroring, also known as shadowing, is a useful approach that allows to make changes to production with the least amount of risk. A mirrored service obtains a copy of all active traffic. The requests are mirrored as "fire and forget", their responses will not be returned.

- Apply Istio configuration for mirroring:

  ```
  sh mirroring.sh
  ```

  Figure 25 shows mirroring in Kiali.

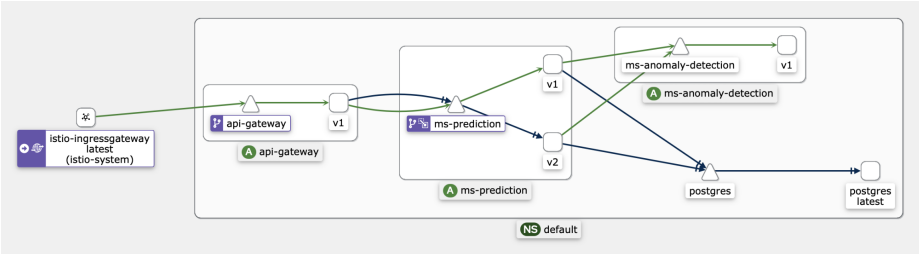

Figure 25: Istio's mirroring visualization in Kiali.

Figure 26 shows mirroring in Grafana. There is no success rate for the Prediction Advanced.

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
.	unknown.unknown	-	NaN	NaN	NaN	-
ms-anomaly-detection.default.svc.cluster.local	ms-anomaly-detection.default	0.89 ops/s	37.50 ms	48.03 ms	80.00 ms	100.00%
api-gateway.default.svc.cluster.local	api-gateway.default	0.45 ops/s	134.62 ms	226.92 ms	247.69 ms	100.00%
ms-prediction.default.svc.cluster.local	ms-prediction.default	0.44 ops/s	79.41 ms	150.00 ms	240.00 ms	100.00%
ms-fog.default.svc.cluster.local	ms-fog.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-prediction.default.svc.cluster.local	ms-prediction-advanced.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-iot.default.svc.cluster.local	ms-iot.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-analytics.default.svc.cluster.local	ms-analytics.default	0.00 ops/s	NaN	NaN	NaN	NaN

Figure 26: Istio's mirroring in Grafana.

- Delete all Istio configurations:

```
sh delete_istio.sh
```

5.8 Fault Injection Delay

With Istio, it is possible to inject delays to test the system resiliency.

Thus, the Prediction Advanced microservice has a 10s delay while predicting with Prophet. All other requests are routed to the Prediction microservice without any failures.

To measure the normal response time, traffic shifting can be applied again to route all the trafic to the Prediction Advanced.

- Apply Istio configuration for traffic shifting:

```
sh traffic-shifting-0-100.sh
```

Figure 27 shows traffic shifting in Kiali.

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
-	unknown.unknown	-	NaN	NaN	NaN	-
api-gateway.default.svc.cluster.local	api-gateway.default	0.02 ops/s	NaN	NaN	NaN	100.00%
ms-prediction.default.svc.cluster.local	ms-prediction-advanced.default	0.02 ops/s	NaN	NaN	NaN	100.00%
ms-fog.default.svc.cluster.local	ms-fog.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-prediction.default.svc.cluster.local	ms-prediction.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-iot.default.svc.cluster.local	ms-iot.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-anomaly-detection.default.svc.cluster.local	ms-anomaly-detection.default	0.00 ops/s	NaN	NaN	NaN	NaN
ms-analytics.default.svc.cluster.local	ms-analytics.default	0.00 ops/s	NaN	NaN	NaN	NaN

Figure 27: Istio's traffic shifting visualization in Kiali.

Figure 28 shows in Postman the normal response time of predicting with Prophet.

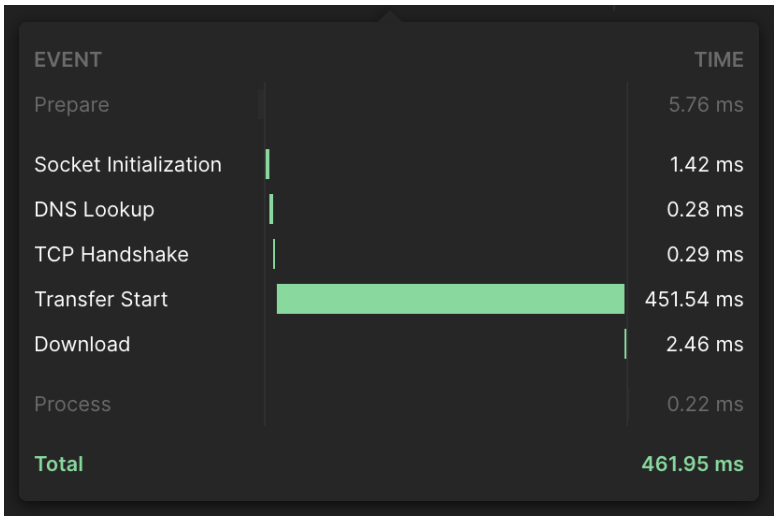

Figure 28: Response time of Prediction Advanced without delay in Postman.

- Delete all Istio configurations:

  ```
  sh delete_istio.sh
  ```

- Apply Istio configuration for fault injection delay:

  ```
  sh fault-injection-delay.sh
  ```

Figure 29 shows in Postman the response time of predicting with Prophet after introducing delay.

Figure 29: Response time of Prediction Advanced with delay in Postman.

Figure 30 shows in Postman the response time of predicting in the Prediction microservice after introducing delay in the Prediction Advanced microservice.

Figure 30: Response time of Prediction without delay in Postman.

- Delete all Istio configurations:

  ```
  sh delete_istio.sh
  ```

5.9 Timeout

Timeout determines how long to wait for a response from a particular service, ensuring that calls are successful or unsuccessful within a predictable period of time and that services are not delayed indefinitely waiting for responses. To demonstrate this feature, a fault injection delay can be used.

- Apply Istio configuration for timeout:

```
sh timeout.sh
```

Figure 31 shows that API Gateway has a timeout after 5s.

```
akbotik@Akbotas-MacBook-Pro service-meshes-like-istio % sh timeout.sh
gateway.networking.istio.io/simulated-iot-cloud-gateway created
virtualservice.networking.istio.io/timeout created
virtualservice.networking.istio.io/fault-injection-delay created
destinationrule.networking.istio.io/ms-prediction created
upstream request timeout%
akbotik@Akbotas-MacBook-Pro service-meshes-like-istio %
```

Figure 31: Istio's timeout.

Figure 32 shows timeout in Kiali.

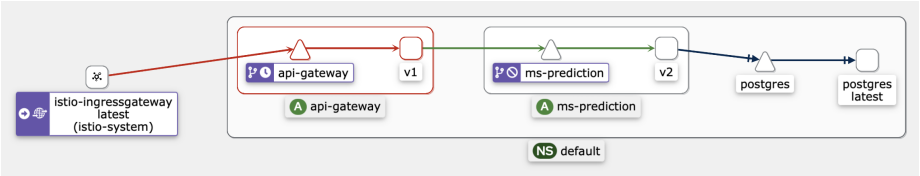

Figure 32: Istio's timeout visualization in Kiali.

- Delete all Istio configurations:

```
sh delete_istio.sh
```

6 Conclusion and Future Work

This bachelor's thesis accomplished several goals. We discussed and examined the concepts of microservices and service meshes. In addition, a microservice-based Simulated Internet of Things Cloud system was implemented. According to our observations in comparing different service meshes, the Istio service mesh was chosen because it offers the greatest number of features and flexibility. Therefore, we based our integration of the service mesh into the implemented distributed system on Istio. To achieve this, we used a number of technologies for deployment such as Docker, Kubernetes, minikube, etc.

To conclude, we may say that we have successfully achieved all of our goals and confirmed that service mesh, like Istio, provides many features that help decouple network logic from business logic. The features tested as part of this research were: ingress traffic, request routing, fault injection abort, retry, load balancing, traffic shifting, mirroring, fault injection delay, and timeout.

The Istio configuration was not as complex as we expected, but the visualization tools could be improved to eliminate errors and avoid constant reloading of dashboards. The most challenging part was deploying the microservices, and the most time-consuming part was implementing them.

References

[1] Rajesh R V. *Spring 5.0 Microservices - Second Edition*. 2nd. Packt Publishing, 2017. ISBN: 978-1-78712-768-5.

[2] Martin Fowler and James Lewis. *Microservices*. URL: https://www.martinfowler.com/articles/microservices.html (visited on 09/22/2022).

[3] Stephen J. Bigelow. *An A-to-Z guide to a microservices architecture transition*. URL: https://www.techtarget.com/searchapparchitecture/An-A-to-Z-guide-to-a-microservices-architecture-transition (visited on 09/22/2022).

[4] Google Cloud. *What is Microservices Architecture?* URL: https://cloud.google.com/learn/what-is-microservices-architecture (visited on 09/22/2022).

[5] Chris Richardson. *Pattern: Microservice Architecture*. URL: https://microservices.io/patterns/microservices.html (visited on 09/22/2022).

[6] Microsoft Corporation. *Microservice architecture style*. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (visited on 09/22/2022).

[7] Chris Richardson. *Pattern: Monolithic Architecture*. URL: https://microservices.io/patterns/monolithic.html (visited on 09/22/2022).

[8] Chris Richardson. *What are microservices?* URL: https://microservices.io (visited on 09/22/2022).

[9] Chris Richardson. *Microservices patterns: With examples in Java*. Manning Publications, 2018. ISBN: 9781617294549. URL: https://www.manning.com/books/microservices-patterns (visited on 09/22/2022).

[10] *Docker*. URL: https://www.docker.com/resources/what-container/ (visited on 09/24/2022).

[11] Chris Richardson. *Pattern: Service instance per container*. URL: https://microservices.io/patterns/deployment/service-per-container.html (visited on 09/22/2022).

[12] *Kubernetes*. URL: https://kubernetes.io (visited on 09/22/2022).

[13] *Kubernetes*. URL: https://kubernetes.io/docs/concepts/overview/ (visited on 09/22/2022).

[14] Tom Nolle. *Explore container deployment benefits and core components*. URL: https://www.techtarget.com/searchitoperations/feature/Container-deployment-and-the-container-management-system (visited on 09/24/2022).

[15] Ramaswamy Chandramouli and Zack Butcher. *Building Secure Microservices-based Applications Using Service-Mesh Architecture*. 2020-05-27 2020. DOI: https://doi.org/10.6028/NIST.SP.800-204A.

[16] Wubin Li et al. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 122–1225. DOI: 10.1109/SOSE.2019.00026.

[17] Christian E. Posta and Rinor Maloku. *Istio in Action*. Manning Publications, 2022. ISBN: 9781617295829. URL: https://www.manning.com/books/istio-in-action (visited on 09/24/2022).

[18] *Service Mesh Comparison*. URL: https://servicemesh.es (visited on 09/25/2022).

[19] Srini Penchikala. *Service Mesh Ultimate Guide 2021 - Second Edition: Next Generation Microservices Development*. URL: https://www.infoq.com/articles/service-mesh-ultimate-guide-2021/ (visited on 09/24/2022).

[20] William Morgan. *What's a service mesh? And why do I need one?* URL: https://linkerd.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/ (visited on 09/24/2022).

[21] Alexander S. Gillis. *What is a Service Mesh and How Does it Work?* URL: https://www.techtarget.com/searchitoperations/definition/service-mesh (visited on 09/24/2022).

[22] Lee Calcote and Zack Butcher. *Istio: Up and running using a service mesh to connect, secure, control, and observe*. O'Reilly, 2020.

[23] *The Service Mesh Landscape*. URL: https://layer5.io/service-mesh-landscape (visited on 09/25/2022).

[24] *Kubernetes Service Mesh: A Comparison of Istio, Linkerd, and Consul*. URL: https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/ (visited on 09/25/2022).

[25] *Istio*. URL: https://istio.io/latest/about/service-mesh/ (visited on 09/25/2022).

[26] Cameron McKenzie. *What is polyglot programming?* URL: https://www.techtarget.com/searchsoftwarequality/definition/polyglot-programming (visited on 09/17/2022).

[27] Alexander S. Gillis. *What is the internet of things (IoT)?* URL: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT (visited on 08/29/2022).

[28] Oracle Corporation. *What is IoT?* URL: https://www.oracle.com/internet-of-things/what-is-iot/ (visited on 08/29/2022).

[29] Amazon Web Services. *What is the Internet of Things (IoT)?* URL: https://aws.amazon.com/what-is/iot/ (visited on 09/17/2022).

[30] Robert Sheldon. *What Are Sensors and How Do They Work?* URL: https://www.techtarget.com/whatis/definition/sensor (visited on 08/29/2022).

[31] Wikipedia contributors. *Thread pool — Wikipedia, The Free Encyclopedia*. 2021. URL: https://en.wikipedia.org/w/index.php?title=Thread_pool&oldid=1061635285 (visited on 09/16/2022).

[32] Python. *concurrent.futures — Launching parallel tasks*. URL: https://docs.python.org/3/library/concurrent.futures.html#module-concurrent.futures (visited on 09/16/2022).

[33] Microsoft Corporation. *The API gateway pattern versus the Direct client-to-microservice communication*. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern (visited on 09/21/2022).

[34] Chris Richardson. *Pattern: API Gateway / Backends for Frontends*. URL: https://microservices.io/patterns/apigateway.html (visited on 09/21/2022).

[35] Wikipedia contributors. *Factory method pattern — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/w/index.php?title=Factory_method_pattern&oldid=1104702038 (visited on 08/29/2022).

[36] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Westford, MA: Addison Wesley, Mar. 1994.

[37] Wikipedia contributors. *Strategy pattern — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/w/index.php?title=Strategy_pattern&oldid=1106787350 (visited on 08/29/2022).

[38] Maher Abdelshkour. *IoT, from Cloud to Fog Computing*. URL: https://blogs.cisco.com/perspectives/iot-from-cloud-to-fog-computing (visited on 08/30/2022).

[39] Brien Posey, Sharon Shea, and Ivy Wigmore. *What is fog computing?* URL: https://www.techtarget.com/iotagenda/definition/fog-computing-fogging (visited on 08/30/2022).

[40] Eljona Proko, Dezdemona Gjylapi, and Alketa Hyso. "From Cloud Computing to Fog Computing and IoT Development". In: *RTA-CSIT*. 2021. URL: https://api.semanticscholar.org/CorpusID:235363780 (visited on 08/30/2022).

[41] Oracle Corporation. *Thread Pools - Essential Java Classes*. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html (visited on 08/30/2022).

[42] Eugen Paraschiv. *Thread Pools - Essential Java Classes*. URL: https://www.baeldung.com/thread-pool-java-and-guava (visited on 08/30/2022).

[43] Eugen Paraschiv. *Introduction to Spring Data JPA*. URL: https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa (visited on 08/30/2022).

[44] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Comput. Surv.* 41.3 (July 2009). ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: https://doi.org/10.1145/1541880.1541882.

[45] Charu C. Aggarwal. "An Introduction to Outlier Analysis". In: *Outlier Analysis*. Cham: Springer International Publishing, 2017, pp. 1–34. ISBN: 978-3-319-47578-3. DOI: 10.1007/978-3-319-47578-3_1. URL: https://doi.org/10.1007/978-3-319-47578-3_1.

[46] Xiaodan Xu, Huawen Liu, and Minghai Yao. "Recent Progress of Anomaly Detection". In: *Complexity* 2019, Article ID 2686378 (Jan. 2019). ISSN: 1076-2787. DOI: 10.1155/2019/2686378. URL: https://doi.org/10.1155/2019/2686378.

[47] Wikipedia contributors. *Time series — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/w/index.php?title=Time_series&oldid=1099453522 (visited on 09/10/2022).

[48] Bernard Rosner. "Percentage Points for a Generalized ESD Many-Outlier Procedure". In: *Technometrics* 25.2 (1983), pp. 165–172. DOI: 10.1080/00401706.1983.10487848. URL: https://www.tandfonline.com/doi/abs/10.1080/00401706.1983.10487848.

[49] *Anomaly Detection Toolkit (ADTK)*. URL: https://adtk.readthedocs.io/en/stable/ (visited on 09/10/2022).

[50] Katta G. Murty. *Histogram, an Ancient Tool and the Art of Forecasting*. 2002. URL: http://www-personal.umich.edu/~murty/histogram.pdf (visited on 09/13/2022).

[51] Jonathan D. Cryer and Kung-Sik Chan. "Introduction". In: *Time Series Analysis: With Applications in R*. New York, NY: Springer New York, 2008, pp. 1–10. ISBN: 978-0-387-75959-3. DOI: 10.1007/978-0-387-75959-3_1. URL: https://doi.org/10.1007/978-0-387-75959-3_1.

[52] Robin John Hyndman and George Athanasopoulos. *Forecasting: Principles and Practice*. 3rd. Melbourne, Australia: OTexts, 2021. URL: https://otexts.com/fpp3/ (visited on 09/13/2022).

[53] NumPy v1.23 Manual. *numpy.histogram_bin_edges*. URL: https://numpy.org/doc/stable/reference/generated/numpy.histogram_bin_edges.html (visited on 09/13/2022).

[54] Ajiboye Abdulraheem, Ruzaini Abdullah Arshah, and Hongwu Qin. "Evaluating the Effect of Dataset Size on Predictive Model Using Supervised Learning Technique". In: *International Journal of Software Engineering & Computer Sciences (IJSECS)* 1 (Feb. 2015), pp. 75–84. DOI: 10.15282/ijsecs.1.2015.6.0006.

[55] Oracle Corporation. *Historical Data and Prediction Accuracy*. URL: https://docs.oracle.com/cd/E57185_01/CBPPU/historical_data_and_prediction_accuracy.htm (visited on 09/12/2022).

[56] NumPy v1.23 Manual. *numpy.histogram*. URL: https://numpy.org/doc/stable/reference/generated/numpy.histogram.html (visited on 09/14/2022).

[57] Sean J Taylor and Benjamin Letham. "Forecasting at scale". In: *PeerJ Preprints* 5 (Sept. 2017). ISSN: 2167-9843. DOI: 10.7287/peerj.preprints.3190v2. URL: https://doi.org/10.7287/peerj.preprints.3190v2.

[58] Darts: Time Series Made Easy in Python. *Timeseries*. URL: https://unit8co.github.io/darts/generated_api/darts.timeseries.html (visited on 09/15/2022).

[59] Prophet — Forecasting at scale. *Python API*. URL: https://facebook.github.io/prophet/docs/quick_start.html#python-api (visited on 09/14/2022).

[60] Darts: Time Series Made Easy in Python. *Exponential Smoothing*. URL: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.exponential_smoothing.html#darts.models.forecasting.exponential_smoothing.ExponentialSmoothing (visited on 09/14/2022).

[61] Dhruvil Karani. *Deep Dive Into Error Analysis and Model Debugging in Machine Learning (and Deep Learning)*. URL: https://neptune.ai/blog/deep-dive-into-error-analysis-and-model-debugging-in-machine-learning-and-deep-learning (visited on 09/07/2022).

[62] *What is Model Evaluation? — Domino Data Science Dictionary*. URL: https://www.dominodatalab.com/data-science-dictionary/model-evaluation (visited on 09/06/2022).

[63] "Model Evaluation". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 683–683. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_550. URL: https://doi.org/10.1007/978-0-387-30164-8_550.

[64] "Mean Squared Error". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 653–653. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_528. URL: https://doi.org/10.1007/978-0-387-30164-8_528.

[65] Jim Frost. *Mean Squared Error (MSE)*. URL: https://statisticsbyjim.com/regression/mean-squared-error-mse/ (visited on 09/06/2022).

[66] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[67] Christian Posta and Burr Sutter. *Introducing Istio Service Mesh for Microservices*. O'Reilly Media, 2018. ISBN: 978-1-491-98874-9.