Distributed IoT Platform

Author: Joshoua Bigler Advisor: Prof. Dr. Daniel Lenz

May 13, 2025

Abstract

The Internet of Things (IoT) has revolutionized industries by enabling real-time data collection and automation. However, small to medium-sized businesses (SMBs) in the mechanical engineering sector face challenges in adopting IoT solutions due to limited resources, expertise and infrastructure. This project aims to develop a modular IoT platform tailored for SMBs in the mechanical engineering sector, featuring real-time monitoring and integrating modern machine learning techniques. While focusing on a minimum viable product (MVP) architecture derived from the target architecture, this project establishes the foundation for a IoT platform which can be further developed.

Declaration of Independence

"I hereby certify that I have written this thesis independently and have not used any auxiliary materials other than those indicated. The passages of the work, which are taken in the wording or the sense after other works (to it also Internet sources count), were marked under indication of the source".

[illegible]

Joshoua Bigler

May 13, 2025

Date

Contents

1 Introduction

The Internet of Things (IoT) has become an important role inside many different fields like agriculture, transportation, healthcare and manufacturing. By facilitating real-time data collection, analysis and automation, IoT empowers businesses to enhance efficiency, reduce costs and improve decision-making processes. Despite these potential, the widespread adoption of IoT is hindered by significant challenges, particularly for small to medium-sized businesses (SMB). Escpecially SMBs in the mechanical engineering sector frequently lack the resources, expertise and infrastructure to implement robust IoT solutions, making it difficult for them to capitalize on the technology's benefits. Because of these challenges many companies who want to implement a IoT Solution into their businesse are strugling to implement a solution by themselfs.

This project aims to develop a generic IoT solution tailored specifically for small to medium-sized businesses in the mechanical engineering sector. While the platform will share a common core architecture across all customers, it must be modular and highly configurable to accommodate the unique challenges and requirements of individual customers. Striking a balance between creating a generic solution and meeting diverse customer needs represents one of the key challenges in this project. The IoT platform is designed to include a real-time monitoring system and support the integration of modern machine learning techniques. It ensures scalability through an efficient database design, facilitates seamless integration with legacy systems and prioritizes robust data security. A core feature of the platform is the creation of generic machine learning modules designed for applications such as anomaly detection, predictive maintenance and operational optimization. These modules enable the platform to leverage advanced data-driven insights while remaining adaptable to various use cases. Furthermore, the platform incorporates sensor simulation and perturbation mechanisms to facilitate extensive testing. These tools not only ensure the reliability of the system but also enable the effective integration and evaluation of machine learning models by analyzing simulated and perturbed data.

Since to tackle all these challenges would be pushing the limits of this project, this project aims to create the foundation for a IoT platform that can be further developed. This is done by deriving the requirements for the minimum viable product (MVP) architecture, from the target architecture which takes the buisiness requirements into account. This documentation provides a concise overview of the business case and presents the derived software architecture, including its security, monitoring and database design concepts. It also outlines the high-level implementation of the MVP architecture and offers a detailed description of the sensor simulation and perturbation mechanisms.

2 Business Case

The business case in a nutshell presented here are drawn from the author's extensive experience in the field, rather than from formal market analysis. By focusing on real-world challenges faced by machine manufacturers, this document aims to highlight the value and potential impact of the proposed IoT platform, which serves as the basis for the software architecture.

Pain Points

Machine manufacturers, particularly SMBs, face significant challenges that hinder their ability to efficiently manage operations and provide effective customer support. A common issue is the lack of detailed information during machine failures, often caused by product users' inability to identify the root cause of downtime. This knowledge gap can lead to prolonged machine downtimes and higher costs. Additionally, many manufacturers struggle with limited human resources, making it difficult to respond promptly to maintenance requests. Service technicians frequently face frustration when attempting to resolve issues with incorrect or incomplete information, leading to inefficiencies and dissatisfied customers. Moreover, the high cost of dispatching repair workers, particularly for unnecessary travel due to misdiagnosed problems, adds a financial burden that many businesses cannot afford.

Value Creation

The proposed IoT solution addresses these key challenges by providing manufacturers with a comprehensive overview of their machines in the field. Detailed insights into machine failures reduce downtime and frustration during troubleshooting, while enabling minor issues to be resolved per remote, cutting unnecessary technician travel costs. The platform also supports the development of predictive maintenance and anomaly detection systems which reduces the overal machine downtime. Faster failure diagnosis leads to quicker service delivery and increased customer satisfaction while reducing machine downtimes. Regular tracking of machine issues helps address recurring problems proactively, while objective, historical data supports reliable reporting for legal and contractual purposes. Additionally, the platform includes features such as pay-per-use models, proactive notifications and interfaces for integration with external systems (e.g. ERP systems).

3 Software Architecture

To define the software architecture, the requirements are derived from the business case in section 2.

Requirements

1. **User Interface**: The system must provide a user interface to interact with the platform.
2. **Configurable**: The platform hast to be generic and highly configurabe to support a wide range of individual needs.
3. **Scalable**: The platform must be able to scale horizontally to support a large number of devices.
4. **Device Management**: The system shall provide capabilities for provisioning, monitoring, and updating devices.
5. **Data Collection and Processing**: The platform must support mechanisms to gather, process, and analyze data from connected devices.
6. **Security**: The system must be secure and provide mechanisms to protect data and devices.
7. **Monitoring**: The platform must provide monitoring capabilities to track the health of the system and devices.
8. **Alerting**: The system must provide alerting capabilities to notify users of critical events.
9. **User Management**: The system must provide user management capabilities to control access to the platform.
10. **Interface to External Systems**: The system must provide a generic interface to external systems (e.g. ERP-System) to interact with the platform.

3.1 Target Architecture

The software architecture of the the distributed IoT platform must be designed to be modular, scalable, expandable and secure. To achieve this the MVP architecture (Figure 2) is derived from the final target architecture (Figure 1) which is created by taking the Requirements from section 3 into account.

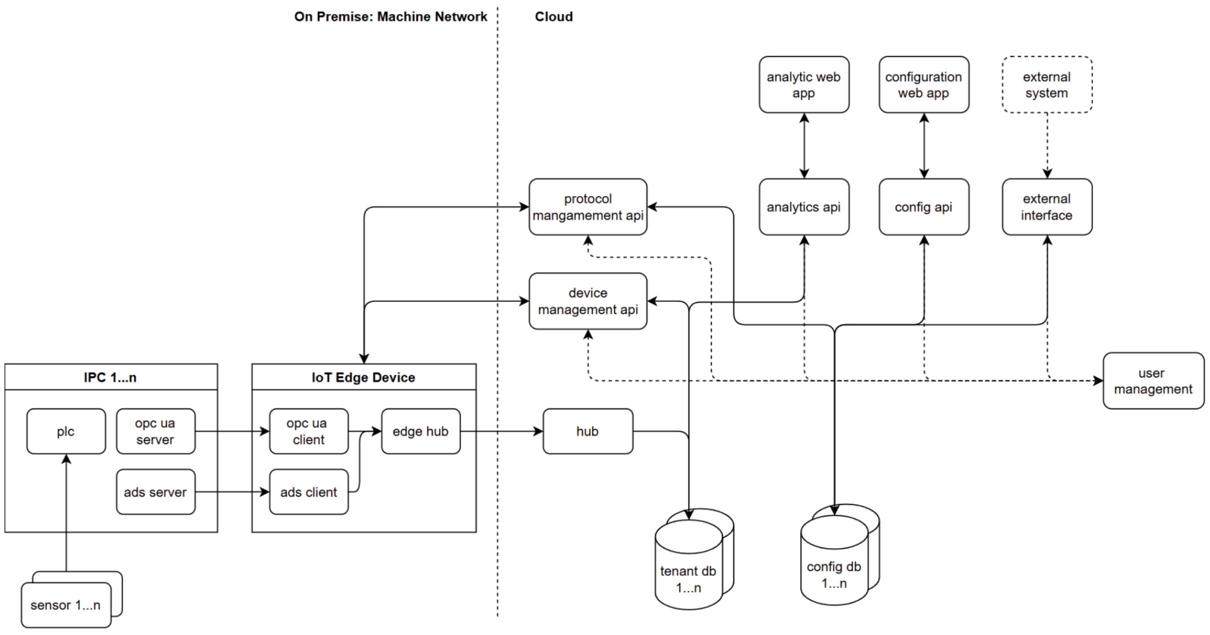

Figure 1: IoT Target Architecture

The final software architecture is split into three main components: the **IoT edge device** (on premise), the **backend** (cloud) and the **frontend** (cloud).

IoT Edge Device
The IoT edge device is responsible for collecting data from various sources, such as a OPC UA or a ADS server, from an industrial PC (IPC) provided by the customer. This data is then transmitted to the cloud hub for further processing. Each communication protocol operates within a dedicated, configurable container managed through the protocol management API. The collected data is first sent to the edge hub via gRPC and then forwarded to the cloud hub, also via gRPC, to ensure efficient and reliable communication.

Backend
The backend is responsible for processing and persisting the collected data from the edge devices. It also provides the different API endpoints for configuration, device management and the user interface. Further more it handles authentication and authorization of users and devices.

Frontend
The frontend is the user interface of the platform. It allows the user to interact with the platform, configure the protocol management API and monitor the platform.

3.2 MVP Architecture

The MVP architecture is a simplified version of the target architecture. It is designed to provide the minimum functionality required to satisfy the base requirements.

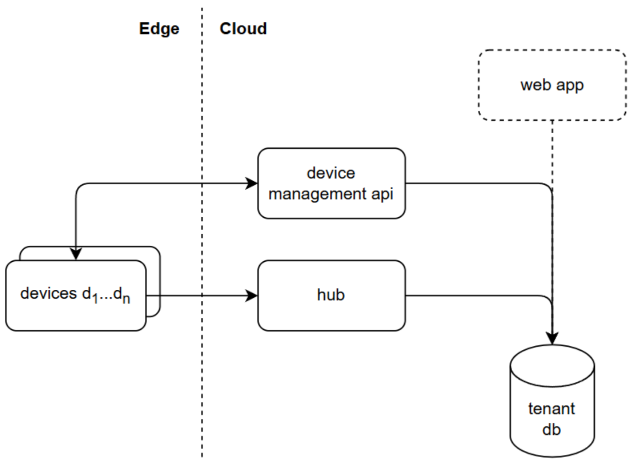

Figure 2: MVP Architecture

In the MVP architecture, the on-premise IoT edge device consists of a single application that simulates a device generating sensor data from various sources, such as temperature and humidity sensors. Initially, this data is simulated rather than originating from actual sensors. The simulated data is transmitted to the cloud hub via gRPC and stored in a PostgreSQL database (*PostgreSQL* 2024). Device configuration, such as creating new temperature sensors or registering new devices, is managed through the device management API. The cloud hub handles receiving data from the devices over gRPC and storing it in the database. For data storage, a separate PostgreSQL database with the Timescale extension (*Timescale* 2024) is maintained for each tenant. Optionally, a web application can be used to visualize the data stored in the database.

4 Monitoring

Monitoring is a vital component of the IoT platform, ensuring that the system operates reliably and efficiently. With multiple interconnected devices, continuous data streams, and supporting infrastructure, an IoT platform requires consistent oversight to detect and address potential issues promptly. Given the complexity and scale of IoT platforms, monitoring provides the necessary insights to ensure smooth operations and support long-term growth. This section will outline the approach that can be used to build an effective monitoring system.

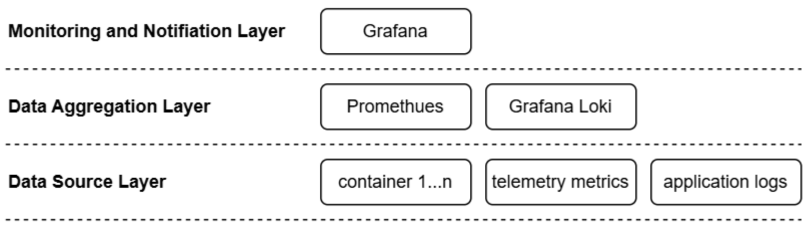

Figure 3: Monitoring Layers

As shown in Figure 3, the monitoring system is divided into three layers: Data Source Layer, Data Aggregation Layer, Monitoring and Notification Layer. In the following sections, each layer will be discussed in detail.

Data Source Layer
The data source layer consists of the application logs, the telemetry metrics from the devices and the container logs. The application logs can be divided into four categories: Debug, Info, Warning and Error. The telemetry metrics are as an example the CPU usage, memory usage, and network traffic of the devices. The container logs refer to the logs generated by Docker (*Docker* 2024) containers, including metrics such as CPU usage, memory usage, uptime and network traffic, for the services they run.

Data Aggregation Layer
On the data aggregation layer the data from the data source layer is collected and aggregated using Prometheus (*Prometheus* 2024) for the container and the telemtry metrics. The data is then stored in a custom prometheus TSDB for further processing. The application logs are collected using Grafana Loki (*Grafana Loki* 2024).

Monitoring and Notification Layer
The monitoring and notification layer is responsible for processing and monitor the system's health and performance from the data aggregation layer. Also it is responsible for sending notifications to the system administrator in case of an error. The monitoring is done using Grafana (*Grafana* 2024) and the notifications are sent using Grafana Alertmanager.

5 Security

Security is a critical consideration in the design and operation of any IoT platform. The IoT ecosystems connect diverse devices, applications, and networks, often transmitting sensitive data and controlling critical operations. This interconnectedness, while enabling advanced capabilities, also introduces significant security challenges, such as potential vulnerabilities in devices, data breaches, and unauthorized access. To address these challenges, a comprehensive security strategy is essential.

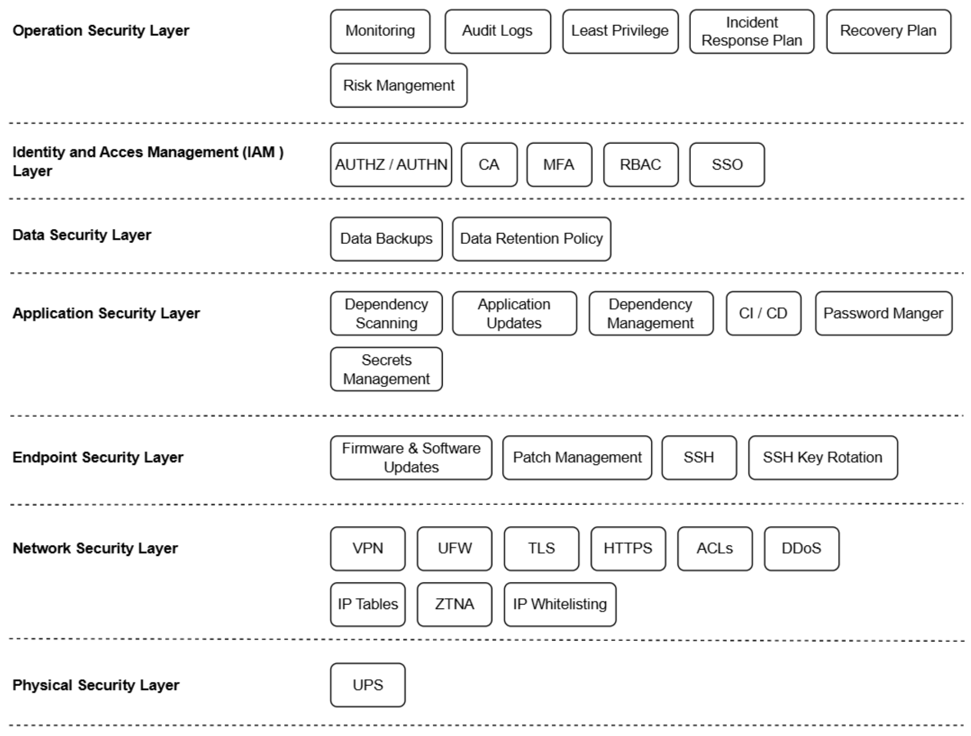

Figure 4: Security Layers

As shown in Figure 4, the security system is divided into seven layers. Altough to address all the layers would go beyond the scope of this project. Here are some of the first steps that should be taken to create a solid security foundation.

Identity and Access Management (IAM)
Keycloak (*Keycloak* 2024) will be used to manage user identities and access control. This includes authentication, authorization and managing roles and permissions for different services and users.

Network Security Layer
Let's Encrypt (n.d.) will provide TLS (Dierks and Rescorla 2008) certificates to ensure secure communication over HTTPS (Fielding, Nottingham, and Reschke 2022b). These certificates will encrypt traffic between clients and servers to protect data in transit. *Certbot* (n.d.), a tool for managing Let's Encrypt certificates, will be set up to automatically renew TLS certificates before they expire. This ensures uninterrupted secure communication without manual intervention. *NGINX* (n.d.) will act as a reverse proxy to manage incoming traffic to the backend services. It will handle TLS termination, ensuring secure HTTPS communication between clients and the proxy. Additionally, NGINX will provide DDoS protection by limiting and filtering malicious traffic and enforce HTTPS redirection, ensuring all traffic is secured with encryption.

6 Sensor Types

To understand the expected data types, formats and volumes, it is essential to first identify the types of sensors that will be used in the IoT platform. Based on this, the data transfer protocol and the database design can be evaluated. In Table 1 are a broad range of expected sensor types listed, that could be used for the IoT platform.

Sensor Type	**Type of Data**
Motor	Control Signals (PWM, Analog), Position Data (Encoders)
Camera	Image Data (JPEG, PNG), Video Data (MP4, AVI)
Temperature Sensor	Continuous Data (Celsius)
Humidity Sensor	Continuous Data (Relative Humidity Percentage)
Pressure Sensor	Continuous Data (Pascals)
Optical Sensor	Light Intensity (Lux)
Audio Sensor	Sound Level (Decibels), Digital Audio Formats (WAV, MP3)
Robot Arm	Position, Angular Position, Torque
Air Quality Sensor	Particulate Matter (PM2.5, PM10), Gas Concentrations (e.g., CO_2, O_3)
Ultrasonic Sensor	Distance Measurements (cm, Meters)
Accelerometers	Acceleration Forces (g-Force)
Vibration Sensors	Frequency Data, Amplitude of Vibrations
Flow Meters	Flow Rate (m^3/s)
GPS Sensors	Latitude, Longitude
RFID Sensors	Object Identification
Solar Energy Monitors	Solar Panel Performance Metrics (W/m^2, Voltage, Current)
Wind Speed Sensors	Wind Speed (m/s), Wind Direction
Solar Energy	UV Ray Intensity, Solar Energy Prediction
Job Names	String representation of the Job Name
Event Logs	Timestamps, Event Types, System States

Table 1: Sensor Types

6.1 Data Format and Type

According to the possible sensor types in Table 1, the expected data types and formats can be identified. As shown in Table 2, the data types can be categorized into numerical data, text data and binary data formats. And within each category there are different data formats the IoT plaform must be able to work with.

Category	**Examples**
Numerical Data	Integer, Float, Array (1D, 2D, 3D, ..., nD)
Text Data	String
Binary Data Formats	Images: JPEG, PNG, BMP, TIFF Audio: MP3, WAV, FLAC Video: MP4, AVI, MKV

Table 2: Data Types and Formats

6.2 Data Volume and Network Traffic

The data colume and network traffic can be calculated using the following formulas:

$$data_{volume\_per\_time} = devices \times sensors \times samples \times data_{size}$$
$$data_{volume\_total} = retention_{time} \times data_{volume\_per\_time}$$
$$network_{traffic} = data_{volume\_per\_time} \times freq$$

$data_{volume\_per\_time}$	Data generated per time unit, measured in GB/min
$data_{volume\_total}$	Total data generated over a given retention time, measured in GB
$devices$	Total number of devices in the system
$sensors$	Average number of sensors per device
$samples$	Number of samples per time unit, measured in $samples/min$
$data_{size}$	Size of each data sample, measured in $bytes$
$network_{traffic}$	Total data transmitted over the network, measured in GB/min
$freq$	Frequency of data transmission, measured in $transmissions/min$

As shown in Figure 5 the data volume and network traffic are visualized for different number of divices, sensors per device, data sizes and frequencies over a timespan of 36 months.

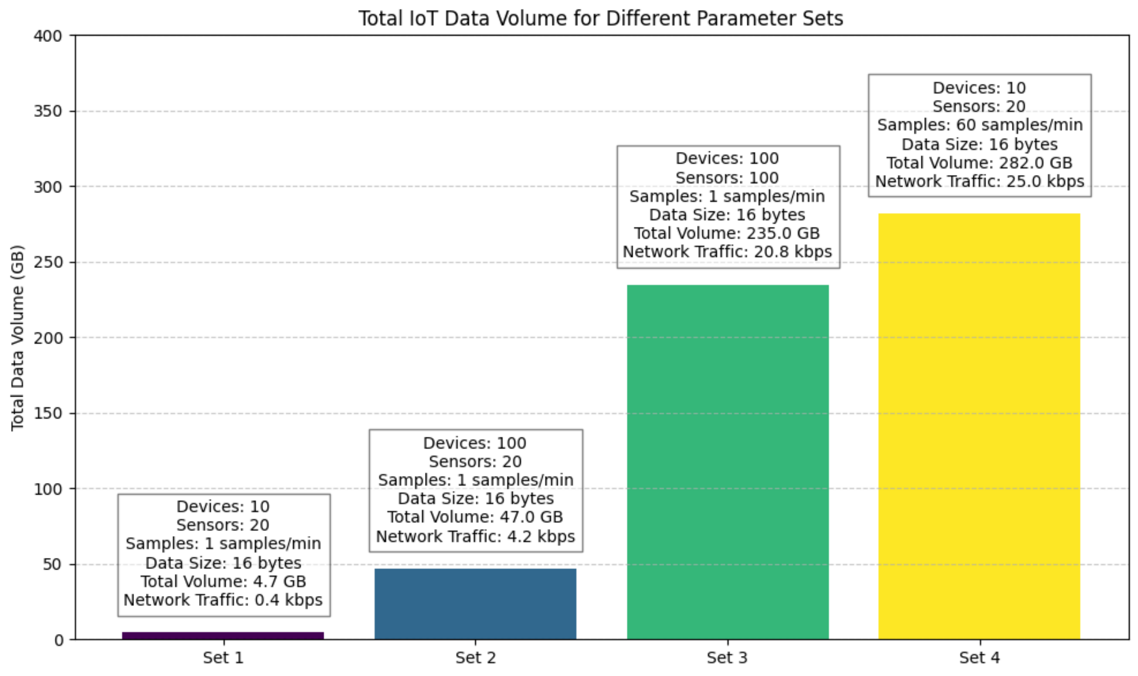

Figure 5: Data Volume and Network Traffic

According to Figure 5 one can see the total data volume over 36 months for 100 devices, 100 sensors per device, 1 $sample/min$ with a data size of 46 $bytes$ per sample, is 235 GB with a network traffic of 20.8 $kbps$. Since the total data volume scales linearly with the number of $samples/min$, even when the number of devices and sensors is much lower (10 devices and 20 sensors per device) the total data volume over 36 months is 282 GB when going from 1 $sample/min$ to 60 $samples/min$ with a network traffic of 25 $kbps$. So the avergae number of $samples/min$ plays a crucial role in the total data volume and network traffic since it can scale up or down based on the given configuration much faster than the number of devices or even number of sensors per device.

Estimation
Every further decision for evaluating a system, based on the data volume and network traffic, is made based on the following assumptions:

- 100 devices
- 100 sensors per device
- 1 *sample/min*
- 16 *bytes* per sample
- 36 months retention time

This results in a total data volume of 235 *GB* and a network traffic of 20.8 *kbps* over 36 months. Where the data volume increases linearly over time as shown in Figure 6.

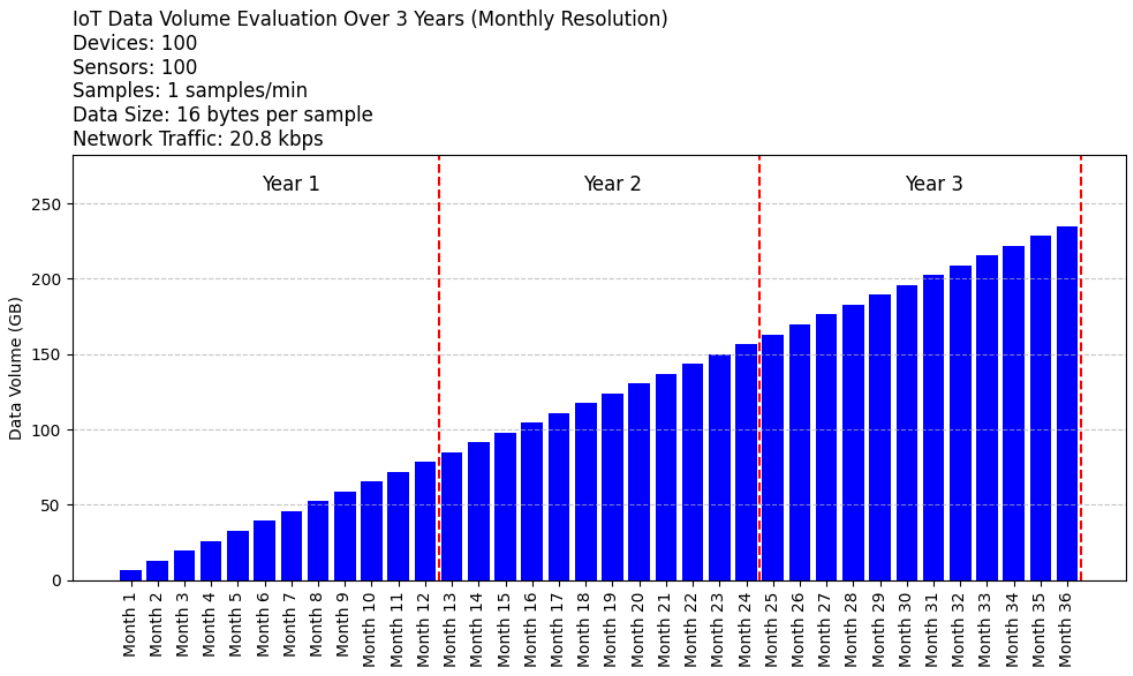

Figure 6: Data Volume over Time

7 Database Design

The database is one of the core components of the IoT platform. It is responsible for storing the data retrieved from the edge devices. The database is a crucial part of the system as it is responsible for storing the data that is used for monitoring, analysis and reporting. Therefore, considering the business case from section 2, the requirements for the database system are as follows:

- **Scalability:** The database should handle the large volume of data generated by IoT devices as the system scales.
- **Time Series Data Management:** The database should be optimized for time series data storage and retrieval.
- **Low Latency for Queries:** The database should provide low latency for queries to support real-time monitoring and analysis.
- **Data Retention Policies:** Efficient handling of historical data with retention policies to manage storage.
- **Security:** Role based access control (RBAC) and encryption of data at rest and in transit.
- **Backup and Recovery:** Regular backups and disaster recovery mechanisms to ensure data integrity.
- **Cost Efficiency:** Cost-effective storage solutions to manage the growing volume of data.

Given these requirements and the need to store timeseries data, PostreSQL with Timescale extension was chosen as the database system for the IoT platform due to the following reasons. PostgreSQL is a powerful and widely used open-source relational database management system that provides robust features for data storage and retrieval. With the Timescale extension on top, PostgreSQL can efficiently handle time series data with optimized storage and query performance.

7.1 Database Schema

To store the retrieved data from the edge devices, a database schema was designed as shown in Figure 7 with following assumtions:

- Each tenant has its own isolated database with an unique tenant identifier.
- A tenant can have multiple devices.
- Each device has a unique device identifier.
- Each device can have multiple sensors.

In first place the schema only takes numerical and text data format into account as defined in Table 2. However, the database implementation currently only supports numerical scalar data types, as this is the only type of data used in this first project version. The schema and implementation can be easily extended to support other data types, such as binary formats.

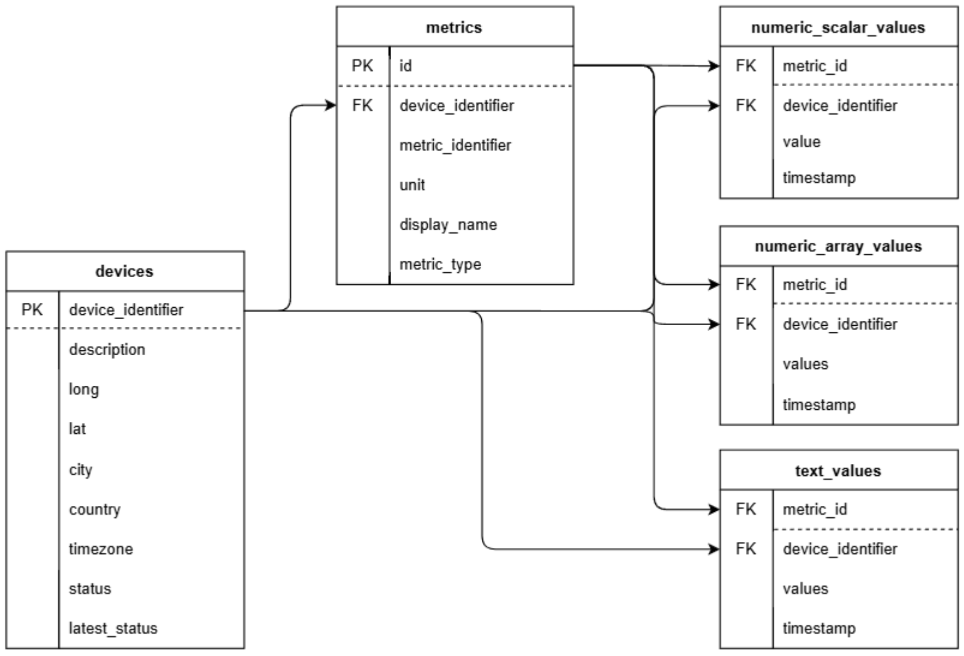

Figure 7: Database Schema

The database schema for a tenant consists of the following tables:

- **Devices:** Contains information about the devices owned by the tenant, such as the device identifier, name and status.
- **Metrics:** Contains information about the metrics, such as the metric identifier, unit, and metric type (numeric_scalar, text).
- **Numeric Scalar Values** Contains the time series data collected from the devices, such as the value and timestamp.

8 System Design and Implementation

In this chapter the components from the MVP architecture in Figure 2 are explained in more detail. For the software implementation Python is used as the only programming language due to it's simplicity and the vast amount of libraries in the field of data science and machine learning.

8.1 Device Management

The Device Management API is responsible for managing the IoT devices and configure the simulated sensor types. The API provides endpoints to register, update and remove devices, as well as to add and remove sensors with the creation of anomalies based on the basis of a probability model. The API is implemented using the *FastAPI* (n.d.) Python framework with Uvicorn *Uvicorn* (n.d.) serving as it's ASGI web server.The device management API consists of two communication protocols: the RESTful API using HTTP/1.1 (Fielding, Nottingham, and Reschke 2022a) for standard request-response communication and the WebSocket protocol (Fette and Melnikov 2011), which uses an HTTP/1.1 handshake to establish the initial connection and then switches to the WebSocket protocol for bidirectional communication. The RESTful API is used for the device registation and configuration. The WebSocket protocol is used to invoke event driven actions on the devices once the connection is establised, like creating or removing different sensor simulators.

Example API Request

The following is an example of how to use the `/tenants/devices/sensors/add` endpoint for creating a humidity sensor simulator with white noise disturbances and a drift for the tenant identifier 100000 and device identifier 000001. The sensor simulator will send data every 10 seconds with a 0.5 offset, a white noise disturbance with a mean of 0, a standard deviation of 0.5, a drift disturbance with a rate of 0.01 and a start value of 0.

```
POST http://127.0.0.1:8000/api/v1/tenants/devices/sensors/add
Headers:
  Content-Type: application/json

Body: {
    "device_identifier": "000001",
    "tenant_identifier": "100000",
    "sensor_identifier": "100003",
    "sensor_type": "humidity",
    "sampling_interval": 10,
    "data": {
      "offset": 0.5,
      "disturbances": [
        {
          "disturbance_type": "white_noise",
          "parameters": {
            "mean": 0,
            "std": 0.5
          }
        },
        {
          "disturbance_type": "drift",
          "parameters": {
            "rate": 0.01,
            "start": 0
          }
        }
      ]
    }
}
```

Listing 1: POST Request Example to Add a Sensor

8.2 Hub

The hub is responsible for ingesting data from different IoT edge devices and routing it to the appropriate data storage. The API is designed to handle different types of data formats as specified within the Table 2 and provide a secure way to ingest data. Although in this project work the hub is only cappable of handling numeric scalar data and device status messages (see Appendix A). As already mentioned in section 3 the message communicatoin is implemented using the *gRPC* (2024) framework from Google. This protocol was chosen based on the expected data formats and types outlied in section 6, due to its high performance, the ability to define the message types in a protobuf schema and gRPC's support for multiple programming languages, as well as its capability to facilitate real-time data streaming. Additional advantages include efficient binary serialization with Protocol Buffers, built-in security through TLS, bidirectional streaming and scalability for distributed systems.

Messages
The hub can either process numeric scalar values or device status messages. It will process the data and store it in the assosiated database table (Figure 7). Further more the hub has an automated mechanism that updates the device status in the devices table based on its received state. For example, it updates the device status to 'offline' if no device status has been received within a specified time, such as 20 seconds otherwise the device state remains 'online'.

Message Queue
The hub uses a queue to store the incoming messages in memory before the messages are processed and stored in the database. This ensures that the database is not overwhelmed by high-frequency data and allows for efficient processing by decoupling message reception from database operations.

8.3 Device

The device is responsible for simulating the sensor data and sending it to the hub. If a device is started it will connect to the hub using gRPC. The device will also register itself with the device management `/tenants/devices/register` endpoint. If the device does not exist already it will be registered and the device is ready to send data to the hub. Furthermore the device will create a WebSocket connection to the device management API to retrieve commands from the device management, like creating or removing a sensor simulator. The device will also send the device status to the hub every given interval e.g. every 10 seconds.

8.4 Shared Library

A Python package named `iot-libs` was developed to facilitate the sharing of common code across various components. This package includes the Protobuf schema, a thread-safe PostgreSQL Engine Manager and a basic Logger class.

9 Sensor Simulation

To simulate the IoT platform as close as possible to a real world environment, sensors will be simulated to generate data. The sensors also can be configured to simulate anomalies and disturbances.

9.1 Humidity Sensor

The humidity sensor generates simulated data based on a real data set from an indoor humidity sensor in an industrial print shop located in London. Authorized members can access the datasets at the GitLab repository (IoT/src_sensor_models/data). To simulate the given sensor data, an ARMA (autoregressive moving average) model was used. ARMA combines a autoregressive (AR) process, with a moving average (MA) process. An autoregessive model is based on the idea that the current value of a time series is a linear combination of past values together with an additive random error (see Appendix B.1). Since the AR model ignores the correlated noise structure in the time series, the moving average (MA) process addresses this limitation by accounting for dependencies in the noise (see Appendix B.2). In order to model the humidity data the time series has to be weakly stationary. This means that the mean sequence is constant and the sample autocovariance sequence (see Appendix B.3) $\gamma_X(i, j)$ depends on i and j only through their difference $|i - j|$. To address the hypothesis that the humidity data is weakly stationary, the Augmented Dickey-Fuller could be used. But in this case, a solely visual inspection is sufficient.

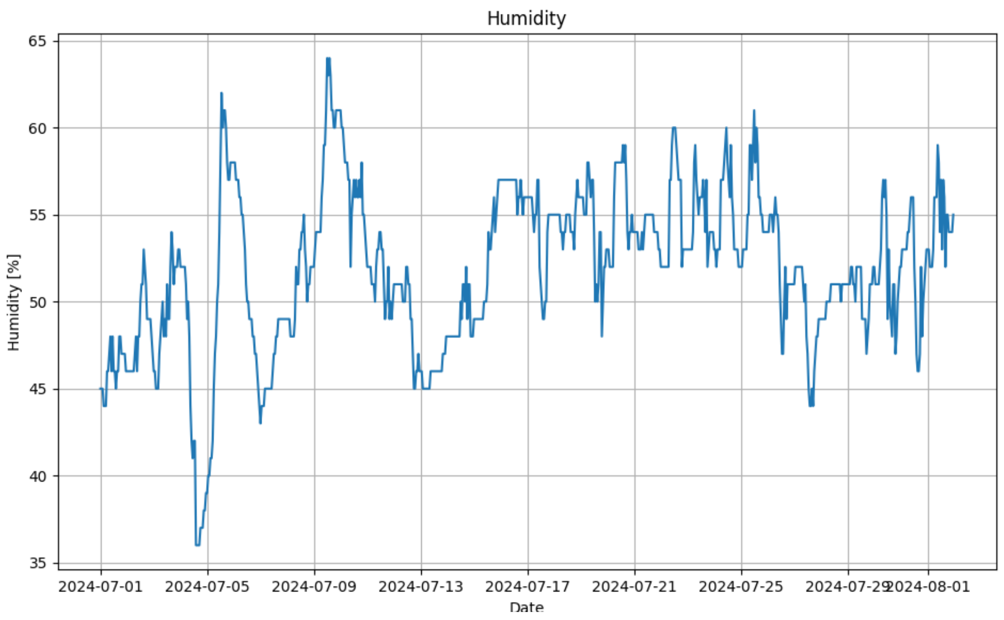

Figure 8: Humidity Data Plot

As seen in Figure 8, the humidity data seems to be weakly stationary since there seems to be no trend in the mean or variance and no seasonality. To further analyze the stationary properties and dependencies in the humidity data, the autocorrelation function (ACF) and partial autocorrelation function (PACF) (see Appendix B.4) plots were generated. These plots provide insight into the structure of the time series.

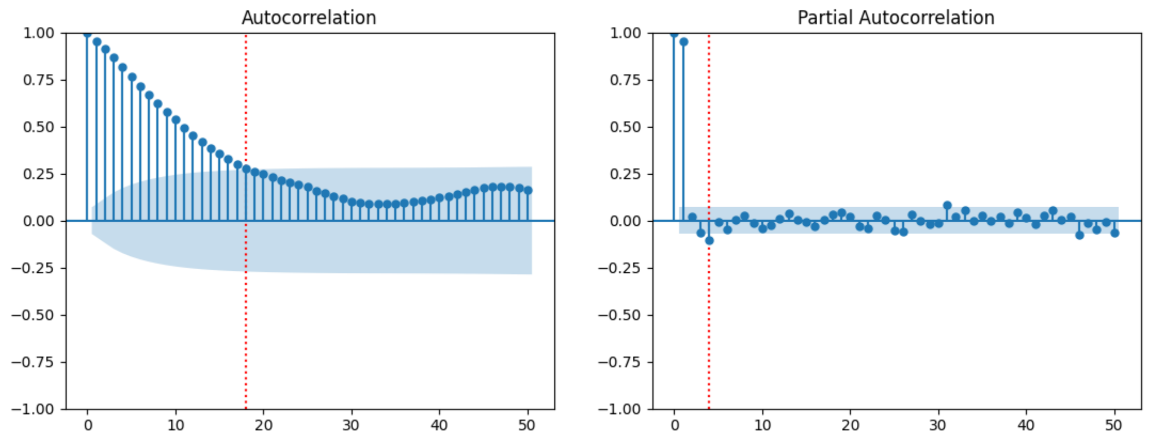

Figure 9: ACF and PACF Plots for the Humidity Data

Figure 9 shows the ACF and PACF plots for the simulated humidity data.

- **Autocorrelation Function (ACF)**: The ACF plot reveals how each observation in the time series is correlated with past observations at various lags. In this case, the ACF decays gradually with a kind of exponential decay, indicating that the time series has autocorrelations that persist over several lags. This suggests the presence of an autoregressive component in the data.
- **Partial Autocorrelation Function (PACF)**: The PACF plot shows the correlation between observations with the effect of intermediate lags removed. Here, the PACF cuts off after lag $p = 4$, indicating that the humidity data can be well-modeled by an ARMA process.

The ACF and PACF patterns confirm that the data exhibits characteristics consistent with an ARMA process. To determine the order of the ARMA model, the Akaike Information Criterion (AIC) (see Appendix B.5), a metric that balances model complexity and goodness-of-fit, can be used with a grid search. Lower AIC values indicate better models.

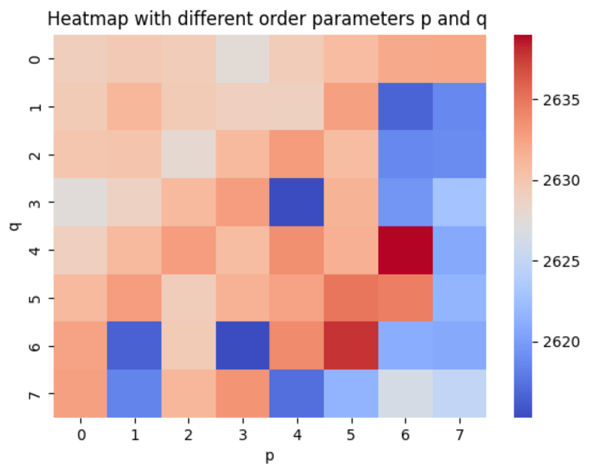

Figure 10: AIC Plot for the Humidity Data

Figure 10 shows the AIC plot, indicating that the optimal ARMA model for the humidity data has parameters $p = 4$ and $q = 3$. Where p is the order of the autoregressive part and q is the order of the moving average part.

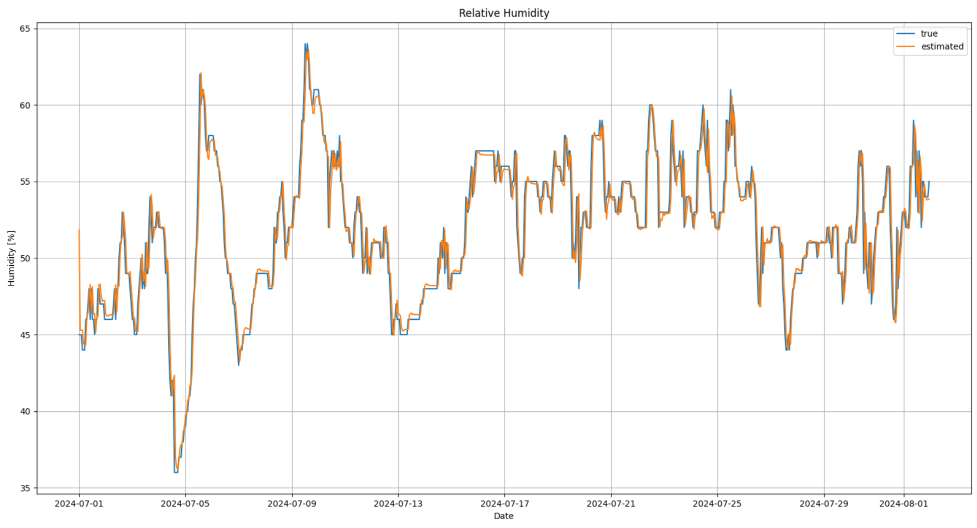

Figure 11: Estimated vs. Actual Humidity

In Figure 11, the estimated humidity data is plotted against the actual humidity data. Visually one can see that the ARMA model closely follows the actual data without further analysing the residuals. As a result, the ARMA model is a good fit for the humidity data and can be used to simulate the humidity sensor.

9.2 Temperature Sensor

The temperature sensor generates simulated data based on a real data set from an indoor temperature sensor in an industrial print shop located in London. Since it is similar to the modelling of the humidity sensor, the intermediate steps are not shown.

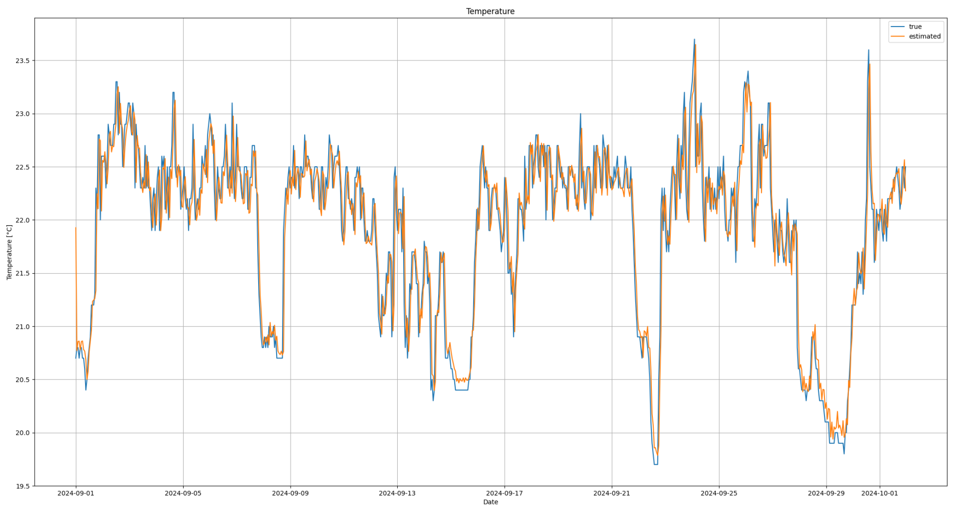

Figure 12: Estimated vs. Actual Temperature

The Figure 12 shows the estimated temperature data plotted against the actual temperature data. Visually one can see that the ARMA model closely follows the actual data. As a result, the ARMA model is a good fit for the temperature data and can be used to simulate the temperature sensor.

9.3 Perturbation

To simulate anomalies and disturbances in the simulated data, the simulated data can be perturbed by adding noise or other types of disturbances. The following mathematical models are used to simulate perturbations:

- **White Noise**: Random noise generated from a normal distribution with specified mean and standard deviation.
- **Random Spike**: Sudden, irregular deviations in the data, occurring with a specified probability and within a defined range.
- **Drift**: Gradual, linear changes in the data over time, simulating long-term trends or shifts.

Model Representation:
Given the original time series X_t generated by an ARMA model, the perturbed series $\tilde{X}_t$ can be expressed as:

$$\tilde{X}_t = X_t + \epsilon_t + S_t + D_t$$

where:

- ϵ_t: White noise component, simulating random fluctuations.
- S_t: Random spike component, simulating sudden anomalies.
- D_t: Drift component, simulating long-term linear trends or shifts.

Components:
Each component is mathematically defined as follows:

1. **White Noise** (ϵ_t):
$$\epsilon_t \sim \mathcal{N}(\mu, \sigma^2)$$
where μ is the mean and σ^2 is the variance of the Gaussian distribution.
2. **Random Spike** (S_t):
$$S_t = \begin{cases} U(a, b) & \text{with probability } p \\ 0 & \text{with probability } 1 - p \end{cases}$$
where $U(a, b)$ is a uniform random variable within the range $[a, b]$, and p is the probability of a spike occurring.
3. **Drift** (D_t):
$$D_t = D_0 + t \cdot r$$
where D_0 is the initial drift value, r is the rate of change, and t is the time step.

10 Conclusion

The development of the MVP provides a solid foundation for the IoT platform, but achieving the final target architecture remains a long-term goal. To ensure the platform aligns with market needs, further analysis of the business case is essential, focusing on customer pain points and their willingness to pay, validated through surveys and interviews.

A key strength of the platform is its modular design, which allows for high configurability while satisfying individual customer requirements. This flexibility will be critical in scaling the platform to meet diverse use cases. The base architecture established by the MVP, including gRPC for real-time message communication and WebSockets for device configuration, is a robust starting point for future enhancements. However, several critical aspects are pending implementation. Strong security measures and comprehensive platform monitoring are necessary to ensure the platform's reliability and resilience. Additionally, creating more sensor simulators will expand the platform's capabilities, enabling the integration of machine learning algorithms for advanced functionalities such as predictive maintenance and anomaly detection.

In conclusion, while the MVP establishes a strong architectural foundation, significant work remains to realize the final vision. This includes implementing robust security measures, comprehensive monitoring, and support for additional communication protocols such as OPC UA or ADS. Furthermore, extending the platform with a customizable user interface will empower users to tailor it to their specific needs. Developing advanced analytics will also be crucial to unlocking the platform's full potential and creating a compelling business case.

List of Figures

List of Tables

Listings

References

Birbaumer, Mirko et al. (n.d.). *Predictive Modeling*. Hochschule Luzern – Technik und Architektur.

Certbot (n.d.). Accessed: 2024-12-31. URL: `https://certbot.eff.org`.

Dierks, T. and E. Rescorla (2008). *The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246*. Tech. rep. RFC 5246. Accessed: 2024-12-31. Internet Engineering Task Force. URL: `https://datatracker.ietf.org/doc/html/rfc5246`.

Docker (2024). Accessed: 2025-01-02. URL: `https://www.docker.com`.

FastAPI (n.d.). Accessed: 2024-12-27. URL: `https://fastapi.tiangolo.com`.

Fette, I. and A. Melnikov (2011). *The WebSocket Protocol, RFC 6455*. Tech. rep. RFC 6455. Accessed: 2024-12-27. Internet Engineering Task Force. URL: `https://datatracker.ietf.org/doc/html/rfc6455`.

Fielding, R., M. Nottingham, and J. Reschke (2022a). *Hypertext Transfer Protocol (HTTP/1.1), RFC 9112*. Tech. rep. RFC 9112. Accessed: 2024-12-27. Internet Engineering Task Force. URL: `https://datatracker.ietf.org/doc/html/rfc9112`.

— (2022b). *Hypertext Transfer Protocol, RFC 9110*. Tech. rep. RFC 9110. Accessed: 2024-12-27. Internet Engineering Task Force. URL: `https://datatracker.ietf.org/doc/html/rfc9110`.

Grafana (2024). Accessed: 2024-12-23. URL: `https://grafana.com`.

Grafana Loki (2024). Accessed: 2024-12-23. URL: `https://grafana.com/oss/loki`.

gRPC (2024). Accessed: 2024-12-27. URL: `https://grpc.io`.

Keycloak (2024). Accessed: 2024-12-27. URL: `https://www.keycloak.org`.

Let's Encrypt (n.d.). Accessed: 2024-12-27. URL: `https://letsencrypt.org`.

NGINX (n.d.). Accessed: 2024-12-27. URL: `https://nginx.org`.

PostgreSQL (2024). Accessed: 2024-12-23. URL: `https://www.postgresql.org`.

Prometheus (2024). Accessed: 2024-12-23. URL: `https://prometheus.io`.

Timescale (2024). Accessed: 2024-12-23. URL: `https://timescale.com`.

Uvicorn (n.d.). Accessed: 2024-12-27. URL: `https://www.uvicorn.org`.

A Hub Protobuf Schema

```
service Hub {
  rpc SendNumericScalarValues(stream NumericScalarValues) returns (google.
      protobuf.Empty) {}
  rpc SendDeviceStatus(DeviceStatus) returns (google.protobuf.Empty) {}
}

enum Status {
  SUCCESS = 0;
  FAILED = 1;
  PENDING = 2;
}

message DeviceStatus{
  string tenant_identifier = 1;
  string device_identifier = 2;
  int32  status = 3;
  google.protobuf.Timestamp timestamp = 4;
}

message NumericScalarValues {
  string tenant_identifier = 1;
  string device_identifier = 2;
  string metric_identifier = 3;
  double value = 4;
  string unit = 5;
  string display_name = 6;
  google.protobuf.Timestamp timestamp = 7;
}

message Response {
  Status status = 1;
  string message = 2;
}
```

Listing 2: Hub Protobuf Schema

B Predictive Modeling

The content in this section is based on Birbaumer et al. n.d.

B.1 Autoregressive Model

The autoregressive model of order p is a discrete stochastic process that satisfies:

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_p X_{n-p} + W_n$$

where $a_1, a_2, \ldots, a_p$ are the model parameters, and W_n is a white noise process with variance σ^2

B.2 Moving Average Model

The moving average model of order q is a discrete stochastic process that satisfies:

$$X_n = W_n + b_1 W_{n-1} + b_2 W_{n-2} + \cdots + b_q W_{n-q}$$

where $b_1, b_2, \ldots, b_q$ are the model parameters, and $W_n, W_{n-1}, \ldots$ is a white noise process with variance σ^2.

B.3 Sample Autocovariance

The sample autocovariance is defined by:

$$\hat{\gamma}(h) = \frac{1}{n} \sum_{i=1}^{n-h} (x_{i+h} - \bar{x})(x_i - \bar{x})$$

with:

$$\hat{\gamma}(-h) = \hat{\gamma}(h), \quad \text{for } h = 0, 1, \ldots, n-1.$$

B.4 Partial Autocorrelation

For a weakly stationary stochastic process $\{X_1, X_2, \ldots\}$, the partial autocorrelation is defined as:

$$\pi(h) = \mathrm{Cor}[X_k, X_{k+h} \mid X_{k+1}, \ldots, X_{k+h-1}]^a$$

where the quantity $\mathrm{Cor}[X, Y \mid Z]$ denotes the conditional correlation of X and Y given the value of Z.

B.5 Akaike Information Criterion

The Akaike Information Criterion (AIC) is defined as:

$$\mathrm{AIC} = -2\log(L) + 2q$$

where L denotes the value of the likelihood function for a particular model and q is the number of variables in this model.