University of Stavanger

BJØRN OLAV FRØYTLOG BJØRNSEN AND SINDRE VATNALAND

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Cloud Telemetry of Aquatic Robots

Bachelor's Thesis - Computer Science - May 2024

We, **Bjørn Olav Frøytlog Bjørnsen and Sindre Vatnaland**, declare that this thesis titled, “Cloud Telemetry of Aquatic Robots” and the work presented in it is our own. We confirm that:

- This work was done wholly or mainly while in candidature for a Bachelor’s degree at the University of Stavanger.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.

“It just works”

– Todd Howards

Abstract

Establishing a solid foundation is crucial when developing a scalable solution. This thesis will explore how to build a lightweight and scalable Internet of Things (IoT) infrastructure for moving robots from an existing infrastructure to the cloud.

We begin by analyzing the existing solution, identifying its weaknesses, and determining the necessary steps to integrate it with an Internet of Things (IoT) solution. The solution considers a high level of flexibility and scalability when installing, deploying, and expanding on features and infrastructure.

To build a flexible and scalable solution, we will explore a variety of technologies and designs. We will explore how event-driven design has allowed us to build an easily expandable solution and how IoT helps us build a lightweight and responsive solution.

Acknowledgements

First and foremost, we would like to thank Remora for giving us the opportunity to write such a thesis. They have given us free reign over how we want to accomplish a solution and trusted us to complete it. Giving us this freedom has allowed us to explore cloud architecture, resulting in a product we proudly present.

Secondly, we wish to express our gratitude to Professor Hein Meling, whose insights have been invaluable. His guidance has influenced both the execution of our work and how we present it in this paper. We would also like to thank Eivind Sørbø for his patience and technical expertise, as well as an easy onboarding at Remora that made the initial step into Remora's internal systems a lot easier.

Lastly, we would like to acknowledge our families and friends for their endless patience, understanding, and love. Their constant support and belief in our abilities have been a source of strength and motivation.

This thesis would not have been possible without the collective support and encouragement of all mentioned, and for that, we are eternally grateful.

Contents

OS	Operating System
RTC	Real Time Connection
IP	Internet Protocol
VPN	Virtual Private Network
IPv4	Internet Protocol v4
IoT	Internet of Things
TCP	Transmission Control Protocol
IaC	Infrastructure as Code
DB	Database
SAS	Shared Access Signature
HTTP	Hyper Text Transfer Protocol
DPS	Device Provisioning Service
TTL	Time To Live
GPS	Global Positioning System
MQTT	Message Queuing Telemetry Transport
VCS	Version Control System

GCP	Google Cloud Provider
AWS	Amazon Web Services
GDPR	General Data Protection Regulation
CI/CD	Continuous Integration/Continuous Deployment
AI	Artificial Intelligence
DLQ	Dead Letter Queue
SDK	Software Development Kit
API	Application Programming Interface
BLOB	Binary large object
UI	User Interface

Chapter 1

Introduction

This thesis builds upon the foundation established by Remora's work. Remora is a company that focuses on innovating how we clean fish farms.

1.1 Background

Remora's current solution introduces something similar to a robotic vacuum cleaner. These robots attach to the net and clean the surplus pill feed and fish waste on the net. Previously, the fish farms were cleaned with high-pressure washers. This requires manpower and downtime and can never compete with the frequency with which the robot could clean the net.

The primary source of this surplus waste introduced to the ocean is agriculture on land [1]. In northern Norway, land agriculture is minimal compared to areas such as the Baltic Sea, and we can see how much organic matter the fish farms have introduced to the environment. Since 1990, the nitrogen levels along the coast have doubled, and the amount of phosphorus has quadrupled [1]. As a result, these increased values adversely affect the overall health of fish.

Remora's current approach to this problem is to build robots that are autonomous cleaners with the option to override the control of their movement manually. Remora has installed a Virtual Private Network (VPN) endpoint for each robot to communicate and override the robots, enabling them to connect directly from their office, using the robot's exposed endpoints. Using these endpoints, they can monitor the robot's sensor data and take control of their movement.

1.2 Problem Statement

Remora's current system faces scalability, flexibility, cost, and efficiency challenges. This section outlines these issues and highlights the need for a centralized platform to address them.

1.2.1 Scalability

Remora's current communication protocol operates on a constrained model with limited robots functioning simultaneously. This restriction is imposed by the current VPN setup. As each robot is assigned a specific Internet Protocol (IP)-address in a VPN subnet range, this limitation is quantified by the available Internet Protocol v4 (IPv4) range of 253 addresses. This limits the number of installed robots to 253 at a given time.

The current VPN solution proves to cause issues, as each robot acts as a local server that is only accessible from Remora's office by using an assigned local IP-address. To alter the configuration of the installed robots, Remora has to cycle through each of their assigned IP-addresses, to apply the change. To configure a specific robot, one must first identify its local IP-address before applying the configuration. This also applies when updating the device's firmware.

With only a few active robots, the current VPN solution does not impose

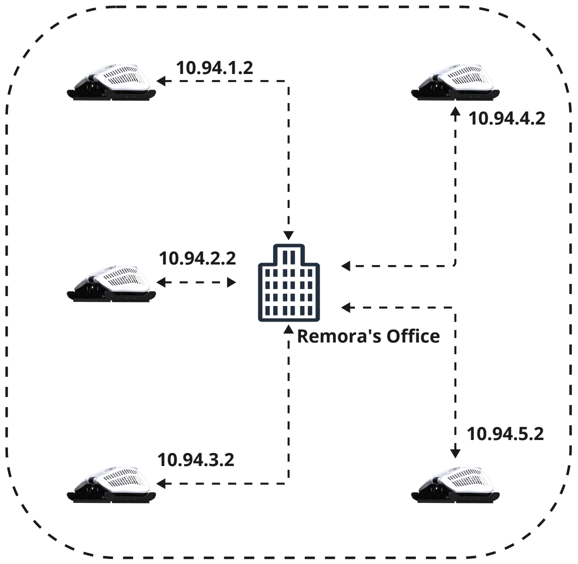


Figure 1.1: Illustration of the current VPN solution.

major problems. However, when scaled up beyond 100 units in a short period of time, this solution has limitations. Remora already plans on doing this in 2024.

1.2.2 Flexibility

The primary structural change Remora has to undertake needs to be done seamlessly. This requires the new solution to integrate well with the existing one and create a seamless and easy transition to the new system.

1.2.3 Cost Considerations

One of Remora’s main concerns is costs. Building a solution where the cost per robot is fairly low is important to build a sustainable business model. The new solution must consider the costs of development, maintenance, security, data storage, and uptime of the potential services at scale.

1.2.4 Efficiency

Automating the process of provisioning new devices and decreasing the amount of configuration reduces time spent on tedious tasks. Remora is a small company, and spending time on menial repetitive tasks and maintenance wastes many resources. For the firm to be cost-efficient, this needs to be avoided.

The issue of updating the robots with the limitation of the VPN solution, as discussed in section 1.2.1, is a tedious and manual process. Remora is in its early production stage with frequent updates, developing a quick and easy way to automate this process will reduce the required resources.

1.2.5 Telemetry

Data telemetry is the process of automatically storing data collected/provided by various sources. These sources can be anything capable of collecting data, such as a simple temperature sensor or a device like a coffee machine [2], often referred to as IoT-devices. By gathering and analyzing telemetry, Remora can measure a robot's performance and notify operators of anomalies.

1.3 System Overview

Remora wants to move all their robots to a centralized system that allows them to build a platform that gives an overview of each unit. The platform should allow them to analyze and measure the robot's performance and provide a way to take control of their movement.

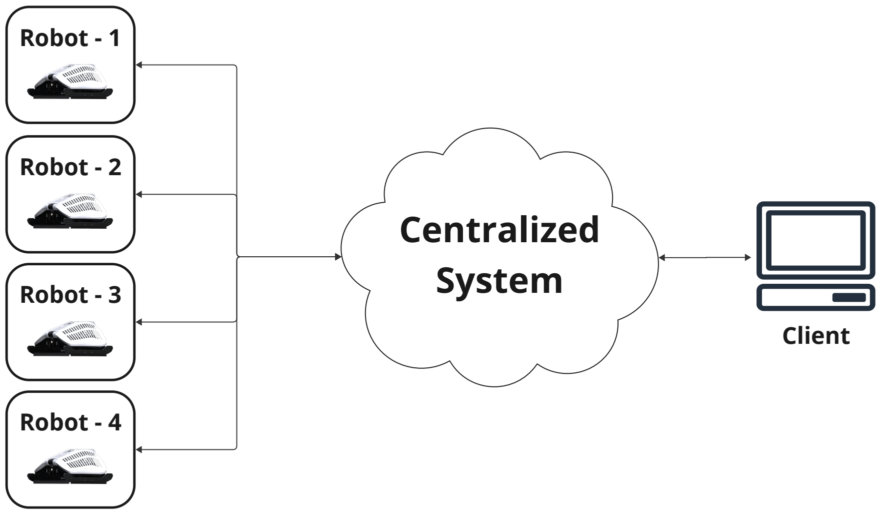


Figure 1.2: Proposed solution.

1.4 Structure

This section explains the structure of the project from start to finish. Each chapter builds upon the last, detailing the development process.

Chapter 2: Foundation

The framework chapter sets the foundation of our work and the reasons for our decisions. We explore different technologies that allow us to build the previously mentioned centralized system and different hosting platforms. We will eventually land on the hardware needed to accompany the robot and how to develop its software.

Chapter 3: Architecture

In the architecture chapter, we look at the bigger picture of the infrastructure. The main challenges of the thesis come from understanding the project's scope and every component we need to use to implement a cloud infrastructure. It is important to go through every service to understand why it is needed and how to use these services to produce an effective infrastructure that can be implemented by Remora. The main focus of the chapter follows the data flow.

Chapter 4: Implementation

This chapter provides an intricate overview of infrastructure deployment, IoT device setup, and cloud services that manage data flow and device updates within Remora's system. This chapter effectively outlines the process of device provisioning, data handling, and firmware updating, all critical for ensuring efficient and secure operations as Remora scales up its fleet of robots.

Chapter 5: Results and Discussions

This chapter discusses the results of our endeavors going through the problem statements and discusses further implications of these findings.

Chapter 6: Future Work & Conclusion

We discuss any potential future works, as this is a solution that builds an infrastructure for a company, there are many features that can be added to the work.

Chapter 2

Foundation

This chapter introduces the fundamentals of Remora's solution and explores various hosting platforms and technologies. The pros and cons of each option will be evaluated.

2.1 Hosting Technologies

As one of the goals for Remora is to store telemetry from the robots and communicate messages back remotely, it is important to develop the solution with technologies that excel in this.

HTTP server

One approach to achieve this is by hosting the solution on a basic Hyper Text Transfer Protocol (HTTP) server. With this setup, all robots can send data to the server's HTTP endpoints, where each endpoint can handle specific logic. Sending messages back to the robots could be achieved by utilizing webhooks [3].

Such a solution offers the necessary features within a compact and easily maintainable architecture, whether hosted locally or in the cloud. However, due to the frequency of messages sent from each robot, using the HTTP protocol increases payload size since each HTTP request includes redundant data in the header [4].

Internet of Things

A desirable option is to convert the robots to IoT devices. As IoT devices usually communicate using the Message Queuing Telemetry Transport (MQTT) WebSocket protocol, excluding the unnecessary header in each message and reducing latency [5].

This solution requires a large infrastructure to route data around. MQTT is lightweight and commonly used to build event-driven architectures, providing an easy and scalable way to connect and expand upon different services.

2.2 Hosting Platforms

Choosing the appropriate hosting solution is important, as it establishes the foundation for how Remora will continue to operate and evolve the platform.

On-Premise

Hosting a service on-premises provides greater control over the system. Hosting the platform in a local, private environment reduces server costs to a one-time payment, excluding power and internet consumption. However, this approach incurs maintenance costs including securing, configuring, handling backups, and ensuring uptime.

Cloud Platform

When looking for a suitable cloud platform that provides the features we need at scale, there are three main companies to consider: Microsoft Azure, Amazon Web Services, and Google Cloud. These platforms provide a multitude of scalable services that integrate well together.

These providers have data centers installed worldwide, where they are responsible for maintaining and securing the servers. This allows Remora's solution to scale worldwide with just small configuration changes.

2.2.1 On-Premise vs Cloud

Since both on-premises and cloud solutions offer valuable options for Remora, selecting the option that aligns with Remora's long-term preference is important.

As shown in Table 2.2.1, there are many reasons to choose either option. Opting for the on-premises solution provides a higher level of privacy and predictable costs, but requires more manual work for configuration and setup, delaying development.

The cloud, on the other hand, offers several benefits. The provider handles maintenance and security, services scale based on demand, and configurations are quick and easy.

As a startup company, Remora needs an option that facilitates rapid development. With expected growth over the coming years and the potential to scale across countries, adopting a cloud platform using flexible IoT technology is a suitable solution.

	On-Premise	Cloud
Maintenance	Requires full maintenance of security, backups, and uptime.	Handled by provider
Configuration	Manual configuration of new services and deployment.	Option to use Infrastructure as Code (IaC) to deploy infrastructure across environments.
Deployment Time	Requires time to set up potential services before developing the solution.	Can set up services and features in a few clicks, before starting on development
Cost	One-time payment of equipment. Predictable monthly costs.	Pay for demanded resources. Fluctuating prices.
Scalability	Requires additional equipment and configuration to scale. Difficult to expand to new locations.	Scales instant- and automatically by demand across countries and continents.
Privacy	Full control of user data and complies strongly with General Data Protection Regulation (GDPR) [6]	Good control of user data, but is stored on a 3rd party server.

Table 2.1: The considerations done on the debate of on-premise vs. cloud

2.3 Cloud Platforms

When deciding which cloud platform to use, we will look at the big three providers: Azure, Amazon Web Services (AWS), and Google Cloud Provider (GCP). Google announced in 2023 that their IoT Core platform is being retired by the end of the year [7]. This makes GCP less viable and shifts the focus on Azure and AWS.

Azure

Azure is Microsoft's cloud platform. It provides a range of services, that automatically scales and binds them together utilizing event-driven architecture. By providing serverless solutions, the client only needs to pay for used resources, as there is no need to allocate resources. Serverless solutions are stateless and automatically scale horizontally on demand [8].

Amazon Web Services

AWS is a strong alternative to Azure and provides many of the same services as mentioned in 2.3. AWS was launched a few years prior to competing alternatives and has had a head start advantage in the market [9]. As Amazon has been developing a lot of IoT devices in-house, such as their Alexa, they have a competing advantage within the field.

2.3.1 Comparison

After examining both providers, there is little difference in price and performance to justify choosing one over the other. The decision largely hinges on external factors. Azure and AWS offer the ability to build scalable IoT architectures at comparable prices.

Coverage

Regarding global coverage, Azure offers the best solution for building IoT architecture in Norway and is currently the only platform providing this service in the country [10]. AWS's nearest server is located in Stockholm [11], while Azure provides services from Eastern Norway.

Price

In isolation, comparing the platforms makes AWS a more compelling choice when looking at its IoT prices. AWS provides a "pay-as-you-go" model for their IoT service and conserve resources [12]. Azure allocates resources for 400,000 messages daily on its cheapest tier [13], but there is a waste of untapped resources.

Preferred Cloud Provider

In the arguments provided it is compelling to see AWS as the better solution for an IoT architecture. However, Azure provides better coverage in the region where Remora operates and has a higher market share of developers [14]. This means there are more potential talent for new hires in the future. The price of hiring and getting new talent onboard is a bigger cost saving than the marginal saving using the AWS platform. With this in mind, the decision lands on Azure as the Cloud provider for this solution.

2.4 Continuous Integration and Deployment

Setting up Continuous Integration/Continuous Deployment (CI/CD) is crucial for a structured and consistent workflow across multiple environments.

Github Workflows

Using Github as the Version Control System (VCS) platform, a workflow on various triggers is configured. These triggers can execute processes manually or automatically, which can build and deploy a solution to desired destinations.

2.4.1 Infrastructure as Code

When deploying infrastructure, it is important to keep consistency between environments. Introducing IaC allows for building infrastructure using configuration files. Integrating IaC with GitHub allows the configuration files to be reviewed before deployment. It can be integrated with GitHub workflows, allowing the solution to be continuously deployed across multiple environments when merging pull requests. A configuration can be altered between environments, and services can be applied to different tiers.

Platform

There are a variety of different IaC platforms to choose between. Microsoft has its own platform, Bicep [15], which offers great support for Azure. One of the most popular platforms is HashiCorp's Terraform [16]. This platform supports IaC on all popular cloud providers thus making it a more desirable choice.

Pulumi

Pulumi is an IaC Software Development Kit (SDK) built on Terraform. They make building and deploying infrastructure possible using familiar programming languages such as Typescript [17]. This improves the productivity in developing the solution and is why it is being used for this project.

2.5 External Device

An external device that acts as a mediator for the robot is beneficial for several reasons: It keeps robots in an isolated network, reduces their exposure to the public internet, and centralizes access control using the mediator. The mediator can be replaced or upgraded with no additional changes to the robot or the cloud architecture. As explained in 1.1 every robot acts as a server with exposed endpoints over the local network. Installing a device on the same local network as the robot will enable the device to communicate with the robot. This device is the mediator between the robot and the Cloud, turning the robots into something resembling a client rather than a server 2.5. This device is named the IoT-bridge.

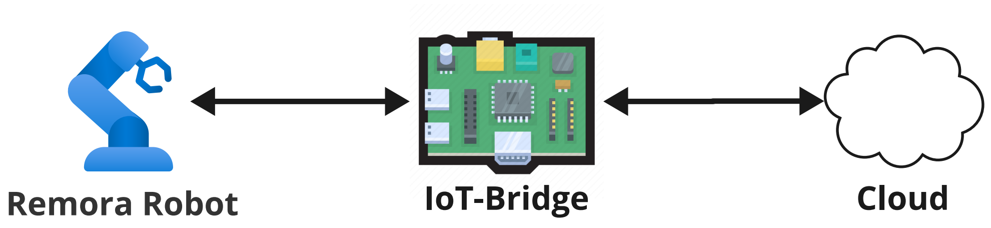


Figure 2.1: The IoT-bridge and the robot are on the same local network; the idea is to decouple the robot from the cloud as much as possible.

2.5.1 Hardware

When deciding on what hardware to choose for the IoT-Bridge, it is important to consider the following: Ease of provisioning, development, and price of the device.

Custom Hardware

Choosing to design and manufacture custom hardware is a tempting choice. The device itself is quite cheap when scaling up to hundreds of thousands of units. The caveat is that you are limited to low-level languages and interfaces, which increases the development time.

Microcontroller Board

There are options such as the Raspberry Pi Pico [18] and Zero [19] to save time on designing and manufacturing custom devices. These are small, cheap, and versatile microcontroller boards and allow the development of familiar languages. Its drawbacks are the lack of physical inputs and computing power, making it harder to install on the network without configurations or ethernet dongles.

Single Board Computer

Single-board computers like the Raspberry Pi 5 [20] are small, powerful computers that can handle fairly demanding tasks. It can run Linux and every language supported on this Operating System (OS). It is a great choice for building a prototype of the solution. It supports a lot of inputs on the board, which allows the device to be installed on a robot by only plugging in power and ethernet. It is quite powerful, making it future-proof for potential additional demanding tasks. Making the Raspberry Pi 5 a perfect solution for this project.

2.5.2 Software and Language

Using a powerful device like the Raspberry Pi 5, the supported languages are limitless. Developing on a familiar platform like Linux and using a well-supported language for Azure increases the efficiency when developing a solution. C#/Dotnet is Microsoft’s flagship programming language and is longtime supported [21]. Using dotnet as the programming language for the IoT-Bridge device ensures good integration against the Azure cloud, future support, and security updates.

Chapter 3

Architecture

Azure cloud platforms provide hundreds of services for building complex applications. A cloud architecture needs to be established to build a flexible and scalable solution to fit Remora's needs for storing and communicating with their robots.

3.1 Service Architecture for Cloud and IoT Devices

Services take care of all the tasks being run on the cloud. It manages resources, such as storage, data processing, monitoring, security, and connection to the cloud. Services have a wide range of applications and use cases.

As shown in Figure 3, the first step of this architecture is to enable each robot to connect to the cloud endpoint individually. This endpoint allows the robots to communicate with the cloud. To understand how this architecture works and why each service is important, we will go through each service and explain their contribution to the solution.

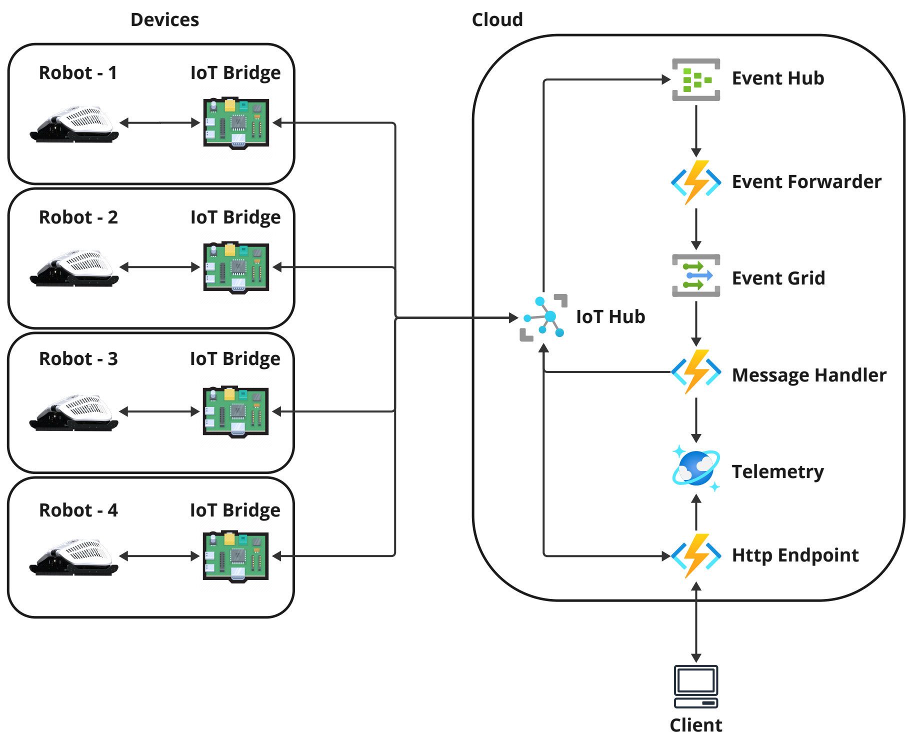


Figure 3.1: Dataflow binding the robot to the cloud and processing telemetry

3.1.1 IoT-Bridge

The figure 3 shows the first link in the architecture is to enable communication between the robot and IoT-Bridge. Using the robot's exposed endpoints, the IoT-Bridge can connect to the robot over a local network and allow them to exchange data. Establishing a connection between the devices allows them to work together and will turn the robot from a local server to a client capable of communicating with the cloud. This configuration makes it appear to the cloud like the robot, and the IoT-bridge is a single device.

3.1.2 IoT Hub

To establish a connection between the IoT-bridge and the cloud, we need an IoT Hub. An IoT hub is a connection point for registered IoT devices. It facilitates communication between a device and the cloud and supports a million active connections per hub [22]. The IoT hub allows the option to send messages back to the device, allowing bi-directional communication. This service allows every IoT-bridge to send messages to the cloud, where the service is configured to route the message to the Event Hub.

Device Twin

The IoT Hub introduces a Device Twin for each device. The Device Twin is a device state stored in the cloud that allows for changing the state regardless of the robot's connectivity. The robot will either receive an updated state as soon as it updates or, next time, regain connection. This property of the IoT hub will be further explored in 4.7

3.1.3 Event Hub

The event hub ensures that no telemetry is lost due to unforeseen circumstances. It receives messages from the IoT Hub and stores all new messages in a buffer capable of holding hundreds of gigabytes with a retention span of 90 days [23]. This buffer enables the architecture to manage bursts of data that would otherwise be lost due to the temporary unavailability of processing resources. Failed messages are placed in a Dead Letter Queue (DLQ), where they are scheduled for retry after a predetermined period.

Realistically, Remora is unlikely to exceed the data capacity required to expand the Event Hub, as data traffic does not typically peak at any point during the day or week. Thus, the Event Hub primarily serves as a precautionary measure to safeguard the data stream and provides a robust foundation for future-proofing solutions.

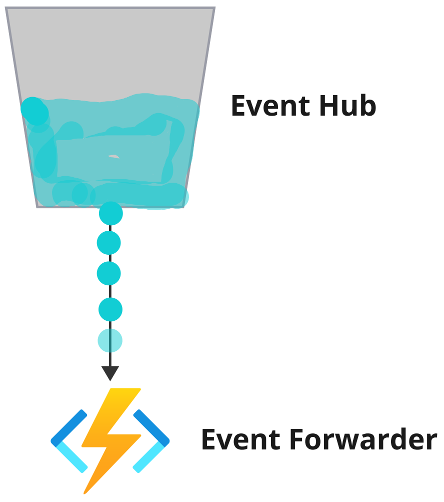


Figure 3.2: The metaphor of picturing a funnel where the Azure Function pulls out the data works well.

3.1.4 Function Apps

Function Apps are the architecture's core as they enable the event-driven design [24]. It is a serverless service, meaning there is no initial server hosting the application, but rather small partitions that get spun up as requests are made. Function apps scale horizontally and automatically based on demand, and their functions are configurable to trigger a variety of Azure services. This means it can listen to changes in storage accounts, databases, event hubs, and many different services and receive events. This event triggers a function and can be handled like any other function property.

3.1.5 Event Forwarder

As seen in the figure 3.1.3, there is set up a Function App called an Event-Forwarder. It includes a singular function responsible for listening to new messages sent to the Event Hub 3.1.3. When a message is pushed to the Event Hub, a trigger will execute the Event Forwarder, pulling and processing the message, before forwarding it to the Event Grid 3.1.6. The nature of the buffer allows the function to pull messages whenever resources are available.

3.1.6 Event Grid

Event Grid makes the solution highly expandable. It is a Pub Sub message distribution service [25] and means a client can push a message to a given Topic, where every client subscribing to this Topic will receive the message. Subscriptions can be configured to filter the incoming events and only trigger functions that comply with the configurations, as seen in 4.7.2.

3.1.7 MessageHandler

Multiple functions are responsible for handling messages pushed to a topic. By utilizing a filter, the functions are only executed when explicitly requested. This structure allows for easily expandable functionality, as newly introduced features just need to subscribe to the topic. One of the many tasks the Messagehandler functions have to do is to listen for telemetry events and store them in a database.

3.1.8 HTTP Endpoint

The HTTP endpoint concerns the frontend. The functions in this function app trigger requests over a HTTP endpoint and are responsible for querying data

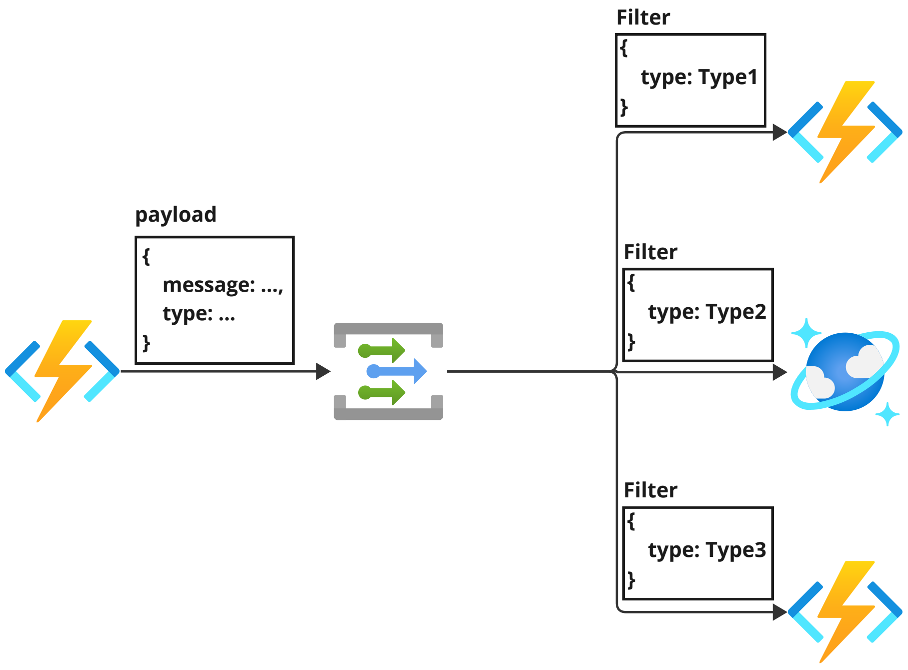


Figure 3.3: A client sends a message to an Event Grid Topic, which broadcasts the message to subscribing clients. The clients have applied filters on the event type.

from the database and returning it to the frontend. These endpoints include fetching IoT-bridge's and its associated telemetry.

3.1.9 Database

The telemetry database needs to be a time-series database as it supports a large amount of data, fast insertion times, and quick and efficient retrieval. The database also needs to support multiple indexing to query not only timestamps but also a preferred device ID. Indexing in the database would allow for faster lookups, saving valuable time, resources, and money. To select a certain Azure infrastructure database, we must recognize and compare our requirements to available databases.

Cosmos

Cosmos Database (DB) is the choice for several reasons. It is a DB that offers a schema-agnostic solution [26]. In the starting phases for Remora, this DB offers the most flexibility without migrating the database between changes. It also provides a schema where one can easily query the structure. One of Cosmos DB's biggest strengths is its ability to handle large amounts of data quickly. This means it can capture detailed information from devices without missing anything or slowing down. This is one of Cosmos DB 's greatest strengths.

3.2 Overview

As each of the services has now been described let us follow the path of the telemetry

1. Robots send telemetry data to the IoT-bridge device using the exposed protocols over the local network.

2. The IoT-bridge, forwards this data to the Azure IoT Hub.

3. IoT Hub sends the data to the Event Hub, where it is temporarily stored and partitioned. This is where an Azure function (Event Forwarder) retrieves data from the Event Hub and pushes it further into the event grid.

4. Processed data triggers events in the Event Grid, which then routes these events to specific Azure functions for further actions or logging, based on predefined topics.

5. Among other things, these functions store the data in the DB

3.3 File Storage

In addition to a DB, we needed to introduce some file storage. Storing files in Azure uses a Storage Account with a Blob Container, a directory to store Binary large object (BLOB) files [27].

Function Apps

The first container is responsible for storing Function Apps. Function Apps can be deployed by referencing a compiled zip file in a container. Updating the function app is as simple as replacing the associated zip file in this container.

Firmware

The second container is the Firmware container. It is responsible for storing firmware for the IoT devices and is how the IoT-bridge devices receive updates.

Chapter 4

Implementation

This solution is a candidate for Remora's infrastructure, which means the project needs proper implementation from the first commit to its follow-through. As in the previous chapter, we will review the technologies and services used and explain why and how they work.

4.1 Infrastructure as Code

Deploying infrastructure through configuration files has been essential in enabling consistent architecture between environments.

Pulumi

Pulumi uses environment configuration files, tailoring the services in each environment, such as pricing tiers and features. This allows for consistent architecture deployment across environments while applying necessary resources for the use case. When using Pulumi to configure infrastructure, the architecture can be set up in Typescript. Initializing classes associated with different services, Pulumi can deploy them when the program runs.

As seen in the code snipped below 4.1 there are included a couple of variables for the main production branch. Each environment may include its own configuration file that can override the properties of the main file.

```
IotCloud:iotHubServiceName: IotHub
IotCloud:iotEventHubName: IotEventHub
IotCloud:iotEventHubNamespaceName: IotEventHubNamespace
IotCloud:iotRouteName: IotEvenHubRoute

IotCloud:iotDpsName: IotHubProvisioningService
IotCloud:iotDpsTier: S1
IotCloud:iotDpsCapacity: 1
```

Listing 4.1: Pulumi Env Config File

When the Pulumi code runs, it will start by executing its main file, as seen in the snippet below 4.1. It loads the configurations automatically selected based on the environment to deploy to. Pulumi continues by creating a resource group and its following services, using the values from the config file.

```
const config = new pulumi.Config();
const resource = config.require("resource");
const cloudLocation = config.require("cloudLocation");

export const resourceGroup = new azureNative.resources.ResourceGroup(
    resource,
    {
        location: cloudLocation,
    }
);

const iotEventHubNamespaceName = config.require("iotEventHubNamespaceName");
const eventHubNamespacePlan = config.require("iotEventHubNamespacePlan");
const eventHubNamespace = createEventHubNamespace(
    iotEventHubNamespaceName,
    eventHubNamespacePlan
);

const iotEventHubName = config.require("iotEventHubName");
const eventHub = createEventHub(iotEventHubName,
    eventHubNamespace
);

const iotHubServiceName = config.require("iotHubServiceName");
const iotPlan = config.require("iotPlan");
const iotHub = createIotHub(
    iotHubServiceName,
    iotPlanIotHubSku,
    1,
    eventHub
);

const iotDpsName = config.require("iotDpsName");
const iotDpsTier = config.require("iotDpsTier");
const iotDpsCapacity = config.require("iotDpsCapacity");
const iotDps = createIotDps(
    iotDpsName,
    iotHub,
    iotDpsTier,
    iotDpsCapacity
);
```

Listing 4.2: Main Pulumi file, creating Resource Group, Event Hub, IoT Hub, and DPS

When the main file is being executed it runs functions to create the various services from different files. This is to ensure reusability and maintain structure. How Pulumi is creating services can be seen in the snippet below 4.1

```
export const createIotDps = (iotDpsName: string, iotHub: IotHub, skuName:
    IotDpsSku, capacity: number): IotDpsResource => {
```

```
  return new azureNative.devices.IotDpsResource(iotDpsName, {
    location: resourceGroup.location,
    resourceGroupName: resourceGroup.name,
    sku: {
      name: skuName,
      capacity: capacity,
    },
    properties: {
      iotHubs: [
        {
          connectionString: pulumi.interpolate‘\${iotHub.connectionString}‘,
          location: resourceGroup.location,
          allocationWeight: 10,
          applyAllocationPolicy: true,
        },
      ],
    },
  })
}
```

Listing 4.3: Pulumi DPS Service Creation

When a service class is initialized like this, Pulumi will pick up on it and put it in a dependency tree. This tree is responsible for building the services according to their dependencies.

4.2 Github

GitHub has been the core tool for developing this solution. It has improved collaboration, documentation, continuous integrations, and deployment.

4.2.1 Separation of Concern

The project is built with multiple pieces that do not directly interact with each other. The IoT-bridge repository contains the source code for the firmware on the IoT-bridge, while the Cloud repository includes everything in Azure. It has been important to separate these applications into different repositories, as it helps by getting a better overview of the applications. The separation in repositories improves the structure, where the commit history and pull requests are easier to follow.

4.2.2 Documentation

GitHub is a VCS and allows tracking changes in the codebase over time, as code pieces are being committed. Writing descriptive messages when committing code has provided a clear, documented history of every feature and bug fix. This will make it easier for developers to trace back changes and better understand the reasoning of the code.

4.2.3 Workflows

Configuring GitHub workflows has provided many benefits for this project by automating tedious and repetitive tasks.

Consistent Testing

This project's workflow is configured to automatically build and test the solution when committing and merging code and has helped catch potential issues early in development.

4.2.4 Continuous Integration and Deployment

Integrating IaC with the workflow has provided a consistent deployment between environments.

Architecture

When completing a merge to the development or production branches in the Cloud repository, a workflow is configured to trigger and automatically deploy the infrastructure in the respective environment, using Pulumi4.1.

However, before deploying the infrastructure, the workflow will build and test the application before compiling, compressing, and zipping the individual function apps.

Pulumi is configured to include these compressed applications in the infrastructure and deploys them along with the infrastructure.

IoT-bridge

Like with the Cloud repository, a workflow is configured to build and test the code in the IoT-bridge repository. If the test is successful, it continues by compressing the compiled application in a zip file and uploading it with a given version number into a BLOB container responsible for storing the device's firmware. This is part of the update solution that will be explored 4.7.

Security

As a workflow allows for setting up secret variables, we can use these secret variables and give Pulumi access to deploy our infrastructure on Azure. By solely providing Pulumi the permission to deploy infrastructure and, by nature, being able to configure secrets and connection strings with no human interaction, the sensitive information is more secure and less prone to leakage.

4.3 IoT-Bridge

The IoT-bridge's main responsibility is establishing a bi-directional connection to a robot. Using this connection, it can forward data from the robot to the cloud and send commands back to the robot.

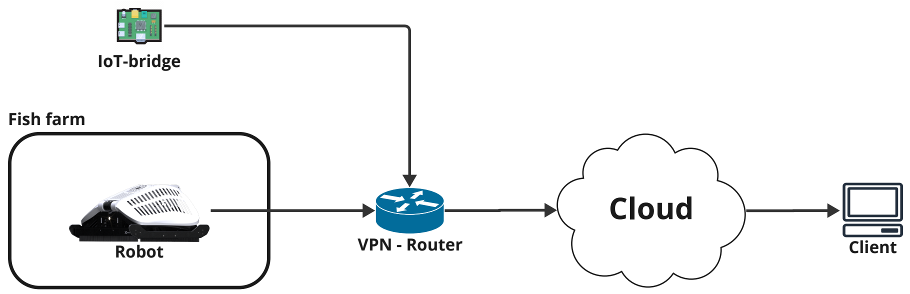


Figure 4.1: IoT-bridge device connected to the robot using the existing architecture

4.3.1 Establishing Connections: IoT-bridge and Robot

When the IoT-bridge is activated, its initial task is to establish a connection with the robot and retrieve its ID. This setup process is straightforward since the robot and the IoT-bridge are on the same local network on a fish farm. Each robot is configured with a static local IP address and has exposed Transmission Control Protocol (TCP) ports. The IoT-bridge device utilizes these ports to establish a connection.

4.3.2 Establishing Connections: IoT-bridge and Cloud

When a connection between the robot and IoT-bridge has been established, it fetches the unique robot ID over a HTTP request. It continues by initializing the IoT client using this ID and connecting to the cloud. This way of connecting to the cloud allows the IoT-bridge to be entirely independent of the robot it connects to as the ID is being fetched from the connected robot. It makes it easy to provision, install, and replace an IoT-bridge as the firmware can be duplicated to multiple devices and connected to a new robot over ethernet.

Device Provisioning Service

While an IoT Hub efficiently handles device connectivity and communication, the initial setup and registration of devices in an IoT system can present challenges, especially when scaling at the pace of Remora. The Device Provisioning Service (DPS) solves this issue by automating the process.

To establish a connection with the IoT Hub, the IoT-bridge must first be authorized. This is achieved through Azure DPS. By providing two symmetric keys stored on the IoT device, These keys combined with the specified endpoint enable the device to be authorized with the DPS in the respective environment.

When a device has been authenticated through the DPS, the device will be registered to a suitable IoT Hub. The IoT Hub returns credentials back to the IoT-bridge, through the DPS and allows it to establish a connection. The code for this logic can be seen here A.

Step by step DPS

1. When the IoT-Bridge first powers on, it connects to the robot, receives its ID, and initializes the IoT device client.

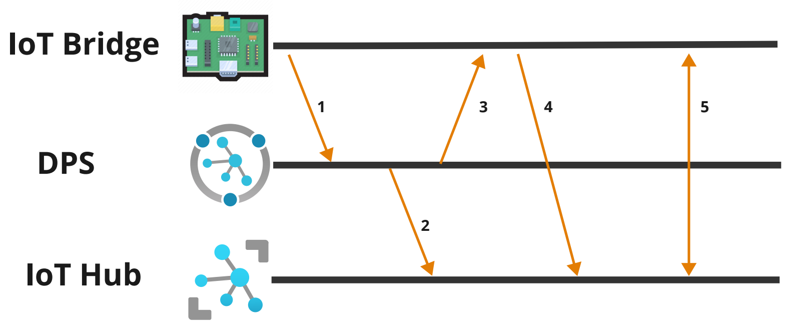


Figure 4.2: As explained per Microsoft‘s documentation [28].

2. The client makes an authorized request to the DPS endpoint with the derived authorization keys.

3. The DPS instance checks the identity of the IoT-bridge against its enrollment list. Once the IoT-Bridge identity is verified, the DPS assigns the device to an IoT-hub and registers it.

4. The DPS instance receives the IoT-Bridge ID and registration information from the assigned hub and passes that information back to the device.

5. The IoT-bridge uses its registration information to connect directly to its assigned IoT Hub and maintains a connection using an MQTT websocket connection.

6. Once authenticated, the IoT-bridge and IoT Hub communicates directly. The DPS instance has no further role as an intermediary. Only if the connection between the 2 units is deleted.

When the request to the DPS succeeds the IoT-bridge will receive credentials that allow the device to connect to the IoT hub. When this connection has been established, a listener is initialized for changes in connection states. It handles reconnect attempts on disconnected events and initializes a listener for incoming cloud messages on connected events.

4.4 Key Vault

Azure Key Vault is a service that provides secure storage of secrets and keys [29]. Azure Functions requires access to sensitive information such as connection strings, API keys, and credentials to access other services. Enabling the service identity setting in the Key Vault configuration authenticates the services within the same resource group to access the Key Vault. This eliminates the need to provide sensitive authentication information and allows services to access Key Vault securely.

4.5 Azure Function Apps

Function Apps are used to expand on the architecture. When an event gets published to the Event Grid 3.1.6 topic, it will broadcast to every subscribed function.

4.5.1 Access

Many of the Function Apps must be authorized to access other services, such as Cosmos, Event Hub, or Event Grid. To grant this access they are all configured with the Key Vault 4.4 endpoint. This endpoint allows the functions to access sensitive information internally and is automatically set up when the function app is initialized. This process can be done dynamically during the function's runtime, ensuring that the latest version of the secret(s) is always applied.

Every function has similar building blocks for its constructor;

1. A Key Vault endpoint is loaded from the Azure configurations, applied by Pulumi
2. Key Vault secrets are accessed where keys and values are being written to the app configurations

Functions

There are 5 Function Apps dispersed over the architecture. These are:

- HttpEndpoint is the function to bind the architecture to a frontend solution using HTTP triggers.
- EventForwarder, which has been explored in 3.1.5.
- TimerTrigger, which continuously refreshes the Blob Container Shared Access Signature (SAS)-tokens for the IoT Device Twins.
- BlobTrigger, which will be further explored in 4.7
- MessageHandler, which is being explained below 4.34

4.5.2 Message Handler

The Message Handler App has 5 functions in its context. This is the biggest context and deals with IoT-bridge messages, such as device states and telemetry.

New Device Handler

This function automates updating the IoT-bridge firmware version info upon a new device registration event as seen in the figure below 4.4. It ensures that

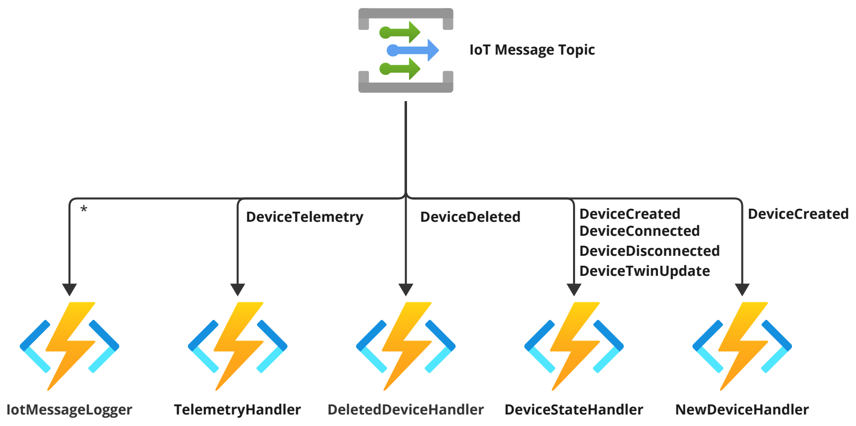


Figure 4.3: All the functions subscribed to one topic: MessageHandler

each new device registered with the cloud is immediately configured with the latest firmware[4.7] version and has the necessary secrets to access the Firmware Blob Container 3.3 securely. This automation is vital for maintaining the integrity and security of the IoT-bridge in a scalable manner.

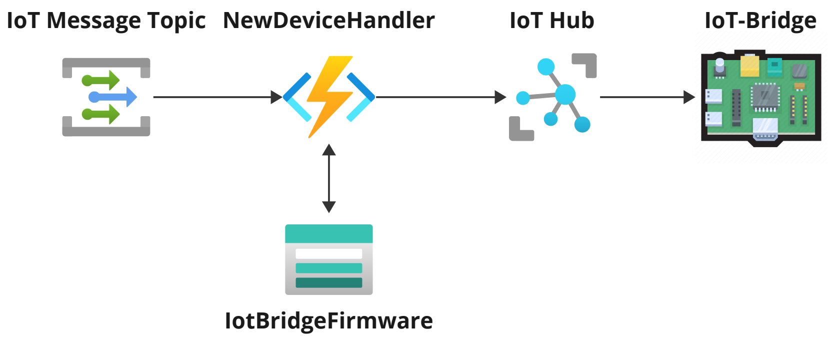


Figure 4.4: All the functions subscribed to one topic: MessageHandler

Device State Handler

This function receives device state events from IoT Hub and forwards them to the Cosmos DB. This is to maintain the current state of each device in a common database as multiple IoT Hubs may occur in a single environment.

Device Telemetry Handler

The device telemetry handler receives telemetry data via the Event Grid. It parses the data, retrieves the telemetry array, and stores it in the Cosmos DB.

Deleted Device Handler

The DeletedDeviceHandler function automates the cleanup of device-related data in a Cosmos DB upon receiving device deletion events originating from the IoT Hub. It ensures that data remains consistent and up-to-date across multiple IoT Hubs. Cleanup of old data is important to help maintain performance and accuracy in the database.

IoT Message Log

This is a logger. It logs everything that passes through the event grid. Its main functionality is to help developers debug, and it does not play a crucial role in the solution's execution.

4.6 Telemetry

Telemetry has been the first feature to be implemented, demonstrating how data flows throughout the infrastructure.

4.6.1 Packet Optimization

Several considerations have been made when building the data flow from the IoT-bridge.

- Hundreds of TCP packets are received every second.
- The TCP packet contains a lot of unnecessary data, for instance, BATTERY_LIFE. Our robot is connected to a power source; it does not use a battery.
- The size of the packets sent to the IoT Hub has a max message size of 4Kb [30].
- Identifying the packets when they have arrived at the cloud.

When the connection has been established between the IoT-bridge and robot, the IoT-bridge will begin receiving messages from the robot over a TCP connection. The messages are filtered based on the desired payload types, temporarily stored in an array, and sent to the cloud at intervals. To maintain the order and time data that is being stored, every data point is constructed with a timestamp of its recording. When redundant data is delivered, it is detected and will override the previous message of the given type. This ensures the payload sent to the cloud is as small as possible without losing important information.

To conserve the most amount of messages and not spend up the IoT Hub message quota, messages maximize the amount of data of 4KB [30]. Sending a message past 4KB will count as two messages.

4.7 Update

Developing the software further for the IoT-bridge devices is desirable for introducing new features and optimizations.

4.7.1 Persistent Software

There are 2 programs running on the IoT-bridge device; the StartupHandler and the IoT software. The StartupHandler is responsible for running the IoT software, downloads and installs updates. StartupHandler is built only to do these simple tasks. It allows the software to run on a sturdy foundation that handles device updates independently.

4.7.2 Publishing new firmware

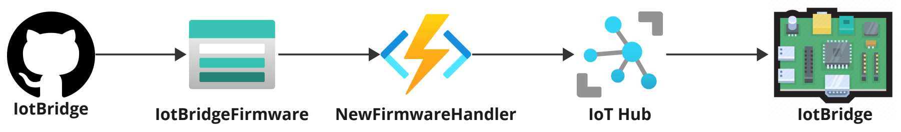


Figure 4.5: Github builds and releases a new version to the firmware blob container where it gets picked up by the NewFirmwareHandler and notifies the IoT-bridge devices

When pushing an update of the device firmware in the IoT-bridge repository to a production branch, a workflow is triggered, which compiles and publishes the new firmware to the firmware blob container.

NewFirmwareHandler

A function app is created to trigger when new entries in the firmware container are added. The event following the function includes the entry's name and the firmware version. Using regex, it verifies the version number and then updates every IoT-bridge Device Twin with this desired version.

Startup Handler

The StartupHandler continuously checks for new updates, using the local Device Twin 3.1.2. Suppose the desired version is newer than the current one, the new version will be downloaded. Upon successful download, the IoT-bridge software will be shut down, and the update will be extracted and replace the existing IoT-bridge program. When this is completed, the program will start up again.

Recovery Mechnism

Suppose the IoT software were to fail. In that case, the local Device Twin will no longer be updated and will break the device. A fallback mechanism is implemented that checks for an update directly in the Blob Container. This happens every 15 minutes and ensures that a device can still be updated remotely, even if the IoT-software were to fail.

4.8 Cloud Monitoring

When scaling up a project, it is important to overview the cloud resources. It helps analyze each service and measure its performance, catch anomalies in data traffic, and optimize the infrastructure.

4.8.1 Grafana

Grafana is a powerful open-source analytics and monitoring solution widely used for visualizing data sources. Connecting Grafana to Azure allows for building custom dashboards with various platform metrics, providing clarity over system performance.

Service Overview

As seen in 4.6, an overview of all the necessary services for the solution is built. It is a simple overview of how many robots are online, how many messages they send, and how each of the mentioned function apps are performing.

Currently, the solution only includes a singular robot that produces a higher load by introducing short intervals between messages.

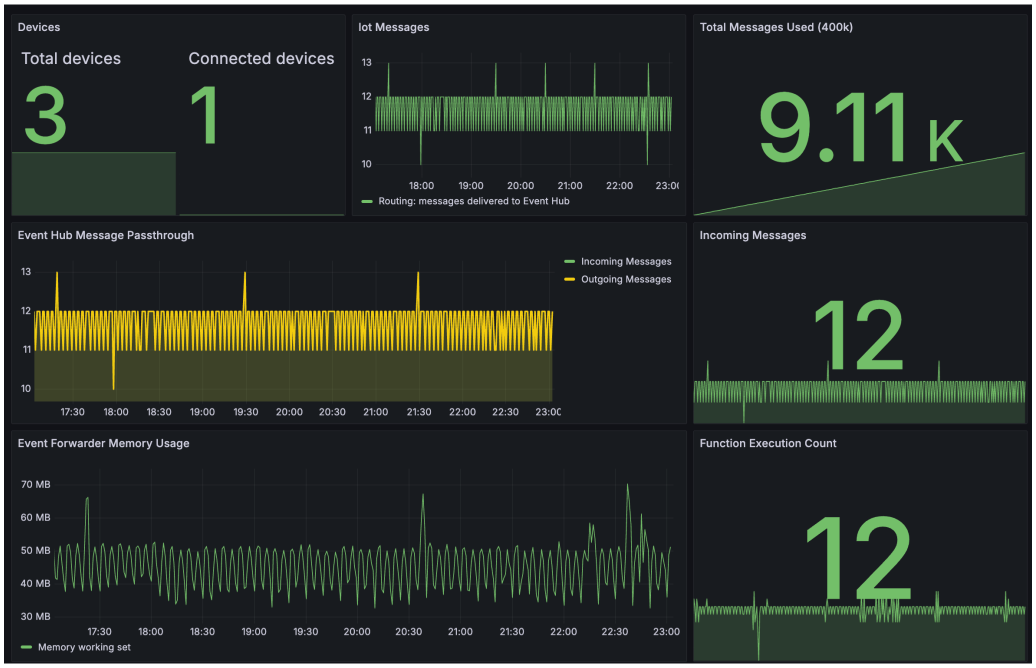
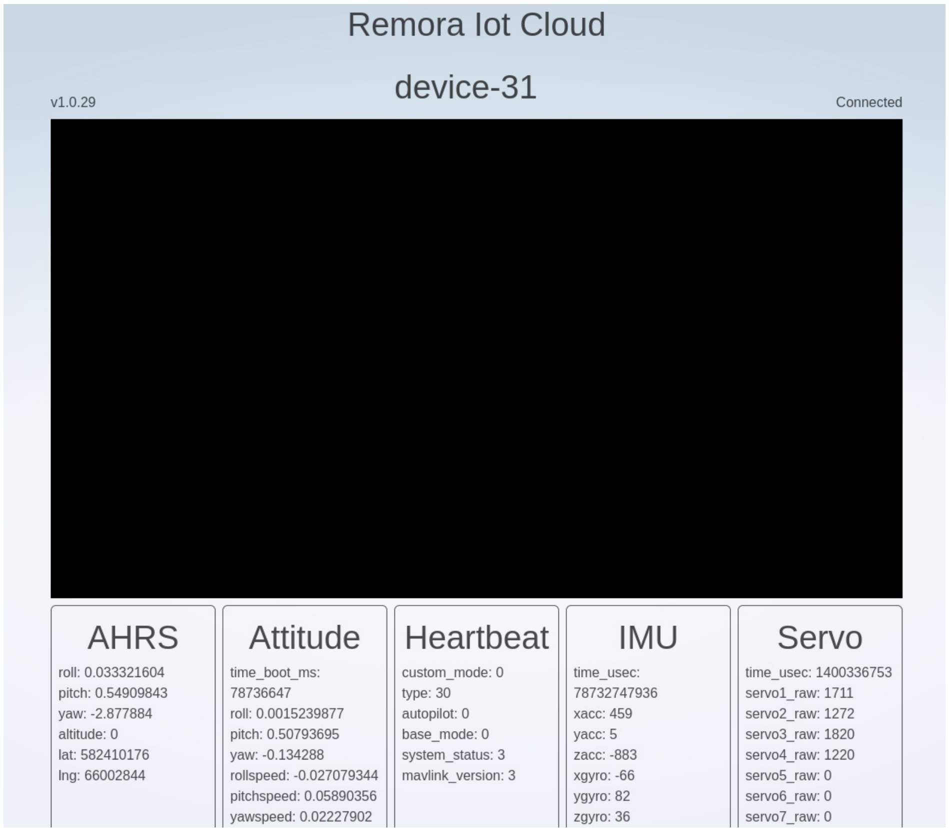


Figure 4.6: Part of the Grafana dashboard displaying connected devices, daily message consumption, message throughputs, and EventForwarder resources.

4.9 Frontend

The frontend is a simple dashboard; it provides a list of all IoT-bridges connected to the Cosmos DB where the telemetry from our robot is stored. The frontend was developed to present the solution to Remora visually. It lists the name and ID of the robot, its online status, and its current running software version.



Remora Iot Cloud

device-31

v1.0.29

Connected

AHRS
roll: 0.033321604
pitch: 0.54909843
yaw: -2.877884
altitude: 0
lat: 582410176
lng: 66002844

Attitude
time_boot_ms: 78736647
roll: 0.0015239877
pitch: 0.50793695
yaw: -0.134288
rollspeed: -0.027079344
pitchspeed: 0.05890356
yawspeed: 0.02227902

Heartbeat
custom_mode: 0
type: 30
autopilot: 0
base_mode: 0
system_status: 3
mavlink_version: 3

IMU
time_usec: 78732747936
xacc: 459
yacc: 5
zacc: -883
xgyro: -66
ygyro: 82
zgyro: 36

Servo
time_usec: 1400336753
servo1_raw: 1711
servo2_raw: 1272
servo3_raw: 1820
servo4_raw: 1220
servo5_raw: 0
servo6_raw: 0
servo7_raw: 0

Figure 4.7: Shows one device with its associated data.

By interacting with an IoT-bridge on the dashboard, we get a display with more detailed information about that IoT-bridge as shown in figure 4.7. This data was specified by Remora.

4.10 Challenges and Solutions

Cloud computing is a new endeavor from our perspective. Understanding the field, researching it properly, and seeing the thesis's scope were challenging. We put a lot of effort into the choices made in Chapter 2 and continuously discussed with Remora how they would build and iterate on any potential changes in our solution going forward.

Architecture

Building a working architecture that scales requires much research before building. Not only have we built an entire architecture, but we accompanied it by developing 5 Function Apps, built a frontend, set up cloud monitoring, and built the software for the IoT device (IoT-bridge). This has been possible by spending time creating a step-by-step plan. Using tools such as GitHub Boards, Workflows, and their VCS has improved productivity immensely and has been the uppermost important step to build a continuously deploying and scalable solution.

Infrastructure as Code

The project's infrastructure allowed us to use Azure User Interface (UI) construct to build our platform. With the intent of being able to deploy this solution to multiple environments, this was not an option, thus the reason to build introduce IaC. IaC has enabled us to continuously integrate and deploy the solution, allow testing in one environment, and publish it to another in no time. It did improve our productivity and increased consistent deployment between the environments.

Circular Dependencies

Setting up IaC for the first time was a challenge as the way you structure code differs vastly from normal code. This resulted in a messy, unstructured code and a circular dependency between the Key Vault and Function Apps. The two services depend on data from each other, causing the solution to be unable to be built. To solve this, we refactored the entire code base from the ground up. This made it clearer to read and write and allowed us to split up the services into smaller steps, allowing both services to be created before applying data from each other.

Device Updating

Updating the device was more difficult than anticipated. The device was supposed to be easily provisional for scaling to multiple robots. Giving the IoT-bridge permission to access the private firmware container wasn't easy. Being creative, we figured we could generate a SAS-token that provides read access to the container. Creating a function that listens to new devices triggered a function that generated this token programmatically and updated the device Device Twin 3.1.2, giving access to the container.

Chapter 5

Results and Discussion

The thesis results explain how each solution we have provided Remora solves its current issue.

5.1 Scalability

All services have been chosen with regard to scalability, from the IoT Hub to the Event Grid. This automatically allows all services to scale on demand, increasing resources as Remora grows. Installing new robots and connecting them to the cloud requires no additional configuration and allows Remora to deliver a seamless solution.

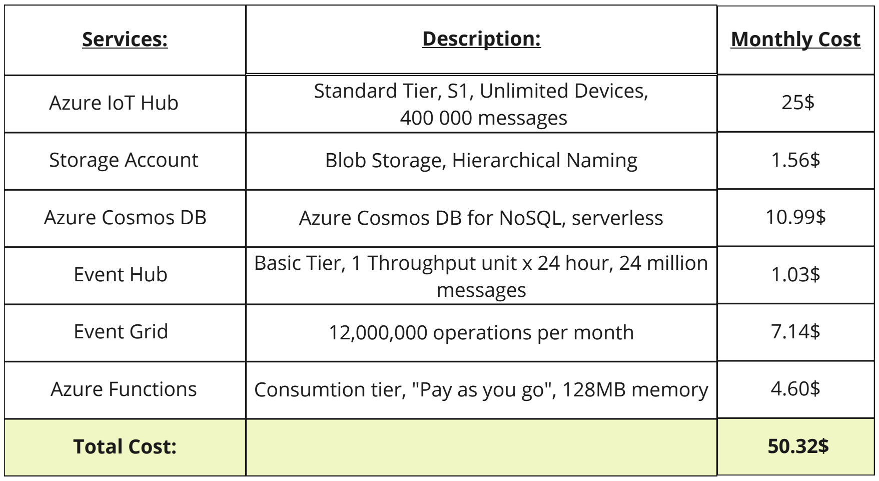
5.2 Flexibility

The cloud-based solution has increased the flexibility of the system. Remora can now deploy updates, modify configurations, and scale operations without previously imposed limitations. The use of Azure IoT Hub and DPS allows for the dynamic management of thousands of IoT-bridge devices. This system supports real-time data handling and device management across geographical locations, without the need for significant system overhauls or downtime. The initial system used by Remora, which was based on a VPN setup, presented many challenges regarding remote access. Each robot required individual updates and configurations, which was time-consuming and limited to on-site maintenance. Remora can now access any IoT-bridge from anywhere globally, provided the user has the proper credentials. This is done through Azure accounts linked to Remora, allowing updates and monitoring of all IoT-bridges without the need to be physically present. This transition supports Remoras remote work capabilities, aligning with contemporary work trends and enabling a more flexible working situation for the company. Using event-driven architecture allows us to easily expand on additional features for future work.

5.3 Cost

By migrating to Azure, Remora utilizes a "pay-as-you-go" model that scales with usage. This model removes upfront costs and allows Remora to adjust resources based on current needs, avoiding unnecessary investment in infrastructure. This is particularly cost-effective for Remora as it grows and needs to scale operations.

The figure 5.1 shows the entire monthly cost of the solution. These costs have been calculated with 100 robots in mind [31] to ensure that Remora can expand without considering any eventual scaling. Each service tier has been considered heavily, which has proved that a cost-effective cloud solution is possible. Suppose Remora expands its fleet beyond our solution. In that case, Azure’s automatic scaling will increase the price, and the configuration must be reconciled regarding the new pricing. The IoT Hub uses a higher tier as it provides the much-needed bi-directional communication. In addition to the



<u>**Services:**</u>	<u>**Description:**</u>	<u>**Monthly Cost**</u>
Azure IoT Hub	Standard Tier, S1, Unlimited Devices, 400 000 messages	25$
Storage Account	Blob Storage, Hierarchical Naming	1.56$
Azure Cosmos DB	Azure Cosmos DB for NoSQL, serverless	10.99$
Event Hub	Basic Tier, 1 Throughput unit x 24 hour, 24 million messages	1.03$
Event Grid	12,000,000 operations per month	7.14$
Azure Functions	Consumtion tier, "Pay as you go", 128MB memory	4.60$
Total Cost:		**50.32$**

Figure 5.1: Services are configured to handle the expected load of 100 robots and 400 thousand messages every day.

Azure cost, Remora will also have to consider the Raspberry Pi's (IoT-bridge) cost.

5.4 Efficiency

With the help of the DPS, the automation reduces the time spent on manual tasks. The dashboard allows for immediate access to the IoT-bridges, with their status and other measurements. This gives quicker response time since the data is readily available on each IoT-bridge and a better foundation for a more informed decision. This provides more proactive maintenance, reducing downtime and boosting the overall reliability and performance of the operation.

5.5 Telemetry

Through time-stamping any telemetry, we have been able to time the packets being transferred from the IoT-bridge all the way to the DB. Though the time to save the data as fast as possible after the IoT-bridge receives the packet from the robot is not crucial, it is a good estimate for how solid, and reliable our solution will prove to Remora. In A we have two timestamps we have highlighted. The first refers to the package creation in the IoT-bridge, and the second is when the packet has been stored in the database. This process takes two seconds.

5.6 Discussion

The newly developed solution allows Remora to scale its production with seamless integration to the cloud. It allows them to install new devices on existing infrastructure and move their focus from maintenance to development. The event-driven architecture allows for flexible development and new features are easily implemented. When Remora decides to expand outside the country, the cloud infrastructure expands easily.

However, developing and maintaining this solution requires experienced developers, as the cloud is large and complex.

Using an Azure Cloud solution with a high market share in the region introduces talented developers to acquire from within the region, reducing the cost of scouting for new hires.

Remora must decide whether to invest in maintaining control of its product or delegate that responsibility to someone else. While this option comes with its own costs and risks, we believe it would be detrimental to the company. In our opinion, most companies aiming to impact the future must build a foundation that allows them to remain as flexible as possible.

Chapter 6

Future Work and Conclusion

As this is development for a company that wishes to expand, much additional work must be done. Potential future problems have emerged throughout the thesis, and so these problems have been discussed thoroughly.

6.1 Future Work

This new solution has a lot of potential, with features to expand upon that greatly improve the Remora product.

6.1.1 Video Feed and Control

Remora has already installed a camera on the robot, and providing a video feed would be the ideal next step in the solution. However, streaming video through the cloud comes with a large cost, as video requires a lot of data throughput. Implementing a peer-to-peer Real Time Connection (RTC) connection between client and robot would be the correct solution.

6.1.2 Controlling the Robots

When an RTC connection has been established between the robot and client, a low latency connection is already implemented. Using this same technology, controlling the robot would be as responsive as possible.

6.1.3 Reducing Telemetry

A lot of the telemetry the robot shares is used to monitor its actions temporarily. Moving this data over to the same RTC connection would reduce the amount of data going through the cloud. This reduction in load from each robot increases the number of devices on a single IoT Hub.

6.1.4 Image Recognition

Currently, the IoT-bridge is only used as a mediator between the robot and the cloud. Using the powerful capabilities of the Raspberry Pi, image recognition software could be applied using its cameras to locate the robot's position more easily or detect and report anomalies.

6.1.5 Relational Database

As the robot is still developing, new features will be added to replace the old ones. The need to continuously update the system accordingly is necessary for the data to be accepted into our filter in the IoT-bridge program. When the robot has a stable physical solution, a Relational DB would be preferable over a schema-agnostic DB

6.1.6 Improved User Experience

Currently, the client's frontend solution is only designed to display minor details about the robot. It should be further developed to monitor the robots properly, provide a video feed, and allow users to control them.

6.1.7 Robot Firmware Update

Currently, the robots are updated manually by connecting to them using the VPN solution. As the IoT-bridge has direct connection to the robot, the bridge can be responsible for downloading its update and forwarding it to the robot automatically.

6.2 Conclusion

The prospect of producing an entire cloud architecture for a company that wants a working product is daunting but extremely rewarding. We have proved that a cloud solution would improve Remora‘s flexibility in their working condition, and through config files with IaC, the operational capabilities are unrivaled. The flexibility is again proved by considering geographical obstacles, where everything can be managed through the IaC. All cloud services consider scaling and price, and we have created a product that Remora is happy with. The new infrastructure has made it easy to install and update each IoT-bridge. Integrating advanced Azure services, like IoT Hub, DPS, and Azure Functions, has automated many of the once manual processes, reducing error rates and operational delays.

Appendix A

Associated code

```
var context = builder.GetContext();

var configuration = builder.ConfigurationBuilder
    .AddJsonFile(Path.Combine(context.ApplicationRootPath,
    "appsettings.json"), optional: true,
        reloadOnChange: false)
    .AddJsonFile(Path.Combine
    (context.ApplicationRootPath,
        $"appsettings.{context.EnvironmentName}.json"),
            optional: true, reloadOnChange: false)
    .AddEnvironmentVariables().Build();

    var keyVaultEndpoint = configuration["KeyVaultEndpoint"];
    if (!string.IsNullOrEmpty(keyVaultEndpoint) &&
    context.EnvironmentName != "Development")
    {
     builder.ConfigurationBuilder.AddAzureKeyVault(
    new Uri($"{keyVaultEndpoint}"),
    new DefaultAzureCredential());
}
```

Function App Startup

```
public class Startup : FunctionsStartup
{
    public override void ConfigureAppConfiguration(IFunctionsConfigurationBuilder builder)
    {
        var context = builder.GetContext();

        var configuration = builder.ConfigurationBuilder
            .AddJsonFile(Path.Combine(context.ApplicationRootPath, "appsettings.json"), optional: true,
                reloadOnChange: false)
            .AddJsonFile(Path.Combine(context.ApplicationRootPath, $"appsettings.{context.EnvironmentName}.json"),
                optional: true, reloadOnChange: false)
            .AddEnvironmentVariables()
            .Build();

        var keyVaultEndpoint = configuration["KeyVaultEndpoint"];
        if (!string.IsNullOrEmpty(keyVaultEndpoint) && context.EnvironmentName != "Development")
        {
            builder.ConfigurationBuilder.AddAzureKeyVault(
            new Uri($"{keyVaultEndpoint}"),
            new DefaultAzureCredential());
        }
    }
}
```

Listing A.1: Initializing Key Vault in function app startup

Initializing IoT Hub Connection

```

var derivePrimaryKey = ComputeDerivedKeySample.
ComputeDerivedSymmetricKey(PrimaryKey, _registrationId);
var deriveSecondaryKey = ComputeDerivedKeySample.ComputeDerivedSymmetricKey
```

```
(SecondaryKey, _registrationId)

using var security = new SecurityProviderSymmetricKey(
    _registrationId,
    derivePrimaryKey,
    deriveSecondaryKey);

var transportHandler = new ProvisioningTransportHandlerHttp();

var provisioningClient = ProvisioningDeviceClient.Create(
    GlobalProvisioningEndpoint,
    DpsIdScope,
    security,
    transportHandler);

var result = await provisioningClient.RegisterAsync()

_deviceClient = DeviceClient.Create(
    result.AssignedHub,
    new DeviceAuthenticationWithRegistrySymmetricKey(
    result.DeviceId,
    security.GetPrimaryKey()),
    TransportType.Mqtt)
```

Listing A.2: Initializing cloud connection to the IoT Hub using DPS

Pulumi Github Workflow

```yaml
- name: Setup Node
  uses: actions/setup-node@v4
  with:
    node-version: "18.x"

- name: Install dependencies
  working-directory: ${{ env.WORKING_DIR }}
  run: npm install
```

```yaml

- name: Pulumi preview and deploy
  uses: pulumi/actions@v5
  with:
    command: "up"
    stack-name: ${{ env.PULUMI_STACK_NAME }}
    work-dir: ${{ env.WORKING_DIR }}
  env:
    PULUMI_ACCESS_TOKEN: ${{ secrets.PULUMI_ACCESS_TOKEN }}
    ARM_CLIENT_ID: ${{ secrets.AZURE_CLIENT_ID }}
    ARM_CLIENT_SECRET: ${{ secrets.AZURE_CLIENT_SECRET }}
    ARM_SUBSCRIPTION_ID: ${{ secrets.AZURE_SUBSCRIPTION_ID }}
    ARM_TENANT_ID: ${{ secrets.AZURE_TENANT_ID }
```

Listing A.3: Github workflow yaml file

Telemtry logs in Azure

```
2024-05-13T14:45:59Z   [Information]   Executing 'DeviceTelemetryHandler' (
    Reason='EventGrid trigger fired at 2024-05-13T14:45:57.8884690+00:00', Id=
    fc736ccd-57dd-413a-af1e-6439778cc4d9)
2024-05-13T14:45:59Z   [Information]   {

  "id": "625a3870-b6a1-4a5a-9f7f-850ab29e2846",
  "subject": "Iot Message",
  "data": {
    "id": "e5a670ef-f1a9-4e61-a3b6-db94346ff640",
    "eventTime": 2024-05-13T14:45:57.203585Z,
    "deviceId": "device-31",
    "subject": "Iot Message",
    "eventType": "IotBridgeTelemetry",
    "data":
    ...
    WRITING TO DATABASE:
    2024-05-13T14:45:59Z
[Information]   Saving to Cosmos: {
  "deviceId": "device-31",
  "payload": {
    "time_boot_ms": 882535494,
    "lat": 71270,
    "lon": 157057,
    "alt": -5843,
    "relative_alt": -5843,
    "vx": -26,
    "vy": -249,
    "vz": -278,
    "hdg": 0
  },
  "id": "667dcf6c-820b-4695-bf5f-007951d795b7",
  "timestamp": 1715611556914,
  "type": "global_position_int"
}
```

Listing A.4: Timestamps for received telemetry

Bibliography

[1] Per Fauchald Marina Espinasse, Eirik Mikkelsen. Forurensing fra lakseoppdrett. https://kystbarometeret.no/indikatorer/matproduksjon-akvakultur/artikkel/forurensing-fra-lakseoppdrett. Accessed 2024.01.28.

[2] Datatilsynet. What is telemetry data? https://logit.io/blog/post/what-is-telemetry-data/, . Accessed 2024.01.04.

[3] Redhat. Whas-is-a-webhook. https://www.redhat.com/en/topics/automation/what-is-a-webhook. Accessed 2024.01.6.

[4] Mozilla. An overview of http. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Accessed 2024.02.15.

[5] MQTT. https://mqtt.org/. Accessed 2024.01.21.

[6] Datatilsynet. Virksomhetenes plikter. https://www.datatilsynet.no/rettigheter-og-plikter/virksomhetenes-plikter/, . Accessed 2024.03.22.

[7] Google Cloud. Migrate environments from iot core. https://cloud.google.com/architecture/connected-devices/iot-core-migration. Accessed 2024.02.2.

[8] Microsoft. Azure serverless. https://azure.microsoft.com/en-us/solutions/serverless, . Accessed 2024.01.29.

[9] AWS. Aws launch. https://aws.amazon.com/about-aws/, . Accessed 2024.01.28.

[10] Microsoft. Azure global coverage. https://datacenters.microsoft.com/, . Accessed 2024.01.29.

[11] AWS. Aws global coverage. https://aws.amazon.com/about-aws/global-infrastructure/, . Accessed 2024.01.29.

[12] AWS. Aws iot core pricing. https://aws.amazon.com/iot-core/pricing/, . Accessed 2024.01.29.

[13] Microsoft. Azure iot hub pricing. https://azure.microsoft.com/en-us/pricing/details/iot-hub/, . Accessed 2024.02.5.

[14] Ole Petter Baugerød Stokke. Nå er det flere som vil at du skal kunne azure enn aws. https://www.kode24.no/artikkel/na-er-det-flere-som-vil-at-du-skal-kunne-azure-enn-aws/76113836. Accessed 2024.05.3.

[15] Microsoft. Bicep. https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview?tabs=bicep, . Accessed 2024.01.14.

[16] HashiCorp. Terraform. https://www.terraform.io/. Accessed 2024.02.9.

[17] Pulumi. Pulumi. https://www.pulumi.com/. Accessed 2024.02.9.

[18] RaspberryPi. Raspberry pi pico. https://www.raspberrypi.com/products/raspberry-pi-pico/, . Accessed 2024.01.15.

[19] RaspberryPi. Raspberry pi zero. https://www.raspberrypi.com/products/raspberry-pi-zero/, . Accessed 2024.01.15.

[20] RaspberryPi. Raspberry pi 5. https://www.raspberrypi.com/products/raspberry-pi-5/, . Accessed 2024.01.15.

[21] Microsoft. .net and .net core support policy. https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core, . Accessed 2024.02.3.

[22] Microsoft. Iot hub quotas and throttling. https://learn.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-quotas-throttling, . Accessed 2024.02.5.

[23] Microsoft. https://azure.microsoft.com/en-us/pricing/details/event-hubs/, . Accessed 2024.05.09.

[24] Microsoft. Azure functions overview. https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp, . Accessed 2024.01.12.

[25] Microsoft. What is azure event grid? https://learn.microsoft.com/en-us/azure/event-grid/overview, . Accessed 2024.01.10.

[26] Microsoft. What is azure cosmos db analytical store? https://learn.microsoft.com/en-us/azure/cosmos-db/analytical-store-introduction, . Accessed 2024.02.12.

[27] Microsoft. Introduction to azure blob storage. https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction, . Accessed 2024.02.12.

[28] Microsoft. What is azure iot hub device provisioning service? https://learn.microsoft.com/en-us/azure/iot-dps/about-iot-dps, Updated: 2024.

[29] Microsoft. Key vault. https://azure.microsoft.com/en-us/products/key-vault, . Accessed 2024.02.12.

[30] Microsoft. https://azure.microsoft.com/en-us/pricing/details/iot-hub, . Accessed 2024.02.15.

[31] Microsoft. Azure pricing calculator. https://azure.com/e/207f332caf0d49fc850136b12c51e3dc, . Accessed 2024.04.15.