University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Secure and Reliable Systems	Spring semester, 2022 Open
Writers: Brynjar Steinbakk Ulriksen Tor-Fredrik Torgersen	Brynjar S. Ulriksen, Tor-Fredrik Torgersen (Writers signature)
Faculty supervisor: Martin Georg Skjæveland External supervisor(s): Stian Grønås Gudsteinn Arnarson	
Thesis title: Guidelines for Developing a Secure and Reliable IoT Data Pipeline	
Credits (ECTS): 30	
Key words: IoT, Data Pipeline, Cloud	Pages: 117 + enclosure: 22 Stavanger, 15. June 2022

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Guidelines for Developing a Secure and Reliable IoT Data Pipeline

Master's Thesis in Computer Science
by

Brynjar Steinbakk Ulriksen and
Tor-Fredrik Torgersen

Internal Supervisors

Martin Georg Skjæveland

External Supervisors

Stian Grønås
Gudsteinn Arnarson

June 15, 2022

"Programming is a nice break from thinking."

Leslie Lamport

Abstract

This thesis presents 22 recommendations in the form of guidelines to consider when building IoT data pipelines. These guidelines should achieve security and reliability in said pipeline. To test the adequacy of the guidelines, a pipeline is built based on these recommendations using the Miles Connect framework. Tests are performed on the parts of the pipeline, evaluating how Miles Connect performs as one of these parts. These guidelines are evaluated to see how they impact the pipeline.

Acknowledgements

We would like to thank our supervisor for his help during the writing of this thesis. Secondly, Disruptive Technologies provided us with access to their resources. For this we are immensely grateful. Finally, we thank Miles Stavanger for sharing their expertise in the field and inspiration for this thesis topic.

Contents

Abstract **iv**

Acknowledgements **v**

Abbreviations **xi**

1 Introduction **1**
- 1.1 Background and Motivation . . . 2
- 1.2 Problem Definition . . . 4
- 1.3 Objectives . . . 4
- 1.4 Approach and Contributions . . . 4
- 1.5 Outline . . . 5

2 Preliminaries **7**
- 2.1 Quality of Service Terminology . . . 7
- 2.2 Technology . . . 8
  - 2.2.1 Internet Of Things . . . 8
- 2.3 Major Vendor's IoT Pipelines . . . 10
  - 2.3.1 Google . . . 10
  - 2.3.2 Amazon . . . 11
  - 2.3.3 Microsoft . . . 11
- 2.4 Defining an IoT Pipeline . . . 11
  - 2.4.1 IoT devices . . . 13
  - 2.4.2 Data ingestation . . . 13
  - 2.4.3 Message broker . . . 14
  - 2.4.4 Data load . . . 15
  - 2.4.5 Applications . . . 15
- 2.5 Other Relevant Preliminaries . . . 16
  - 2.5.1 Data silo problem . . . 16
  - 2.5.2 LoRaWAN . . . 16
  - 2.5.3 Enterprise Service Bus . . . 17
  - 2.5.4 Integration Platform as a Service . . . 17
  - 2.5.5 Software and hardware providers . . . 18
  - 2.5.6 Security protocols . . . 18

2.5.7 Shared Access Signature . . . 19

3 Related Work **21**
3.1 Timon . . . 21
3.2 Kafka . . . 21
3.2.1 Kafka benchmark . . . 22
3.3 Stream Bench . . . 22
3.4 Industrial Pipelines . . . 23

4 Presentation of Guidelines **25**
4.1 Methodology . . . 25
4.2 IoT Devices . . . 26
4.2.1 IoT security . . . 26
4.2.2 Signal strength . . . 27
4.2.3 Device battery and connectivity . . . 28
4.3 Data Ingestation . . . 28
4.3.1 Data ingestation example . . . 29
4.3.2 Gateway . . . 30
4.3.3 Early stage validation . . . 31
4.4 Message Broker . . . 31
4.4.1 Transfer process . . . 32
4.4.2 Message broker responsibilities . . . 33
4.5 Data Load . . . 34
4.6 Applications . . . 36
4.7 Summary of the Pipeline . . . 37

5 IoT Pipeline Architecture **39**
5.1 Why Separate All Pipeline Steps? . . . 39
5.1.1 Why have a dedicated data ingestion step? . . . 40
5.1.2 Why have a dedicated message broker? . . . 40
5.1.3 Why have a dedicated data load component? . . . 41
5.2 Why use Disruptive Technologies Sensors? . . . 41
5.3 Why Use Miles Connect? . . . 42
5.4 Azure Service Bus as an Enterprise Message Broker . . . 42
5.5 Applications From Asplan Viak . . . 44
5.5.1 Desired use case . . . 45
5.5.2 Technical issues . . . 45
5.5.3 Use cases in this thesis . . . 45

6 Building the Pipeline **49**
6.1 Connecting to DT Studio . . . 49
6.1.1 Generating data . . . 50
6.1.2 Accessing sensors . . . 50
6.1.3 Stream and REST API . . . 51
6.2 Connecting to the Message Broker . . . 54
6.2.1 Miles Connect functionality . . . 55
6.2.2 Azure Service Bus Explorer . . . 56
6.3 Constructing the Data Load . . . 57

6.4 Storing Data in the Cloud . . . 58

7 Verifying the Guidelines **59**

7.1 Guidelines for IoT Device . . . 59
7.1.1 Guideline 1.1 . . . 59
7.1.2 Guideline 1.2 and 1.3 . . . 60
7.1.3 Guideline 1.4 . . . 62
7.1.4 Guideline 1.5 . . . 62
7.1.5 Guideline 2.1 . . . 67
7.1.6 Guideline 3.1 . . . 67
7.1.7 Guideline 3.2 . . . 70
7.2 Guidelines for Data Ingestion . . . 72
7.2.1 Guideline 4.1 . . . 72
7.2.2 Guideline 4.2 . . . 75
7.2.3 Guideline 4.3 . . . 75
7.2.4 Guideline 5.1 . . . 76
7.2.5 Guideline 5.2 . . . 79
7.3 Guidelines for Data Transfer . . . 81
7.3.1 Guideline 6.1 . . . 81
7.3.2 Guideline 7.1 . . . 87
7.3.3 Guideline 7.2 . . . 88
7.3.4 Guideline 7.3 . . . 88
7.4 Guidelines for Data Load . . . 90
7.4.1 Guideline 8.1 . . . 90
7.4.2 Guideline 8.2 . . . 92
7.5 Guidelines for Applications . . . 94
7.5.1 Guideline 9.1 . . . 94
7.5.2 Guideline 9.2 . . . 97
7.5.3 Guideline 9.3 . . . 98

8 Discussion **101**

8.1 Discussing the Pipeline . . . 101
8.1.1 Are the pipeline steps correctly defined? . . . 102
8.1.2 Is the pipeline actually agnostic? . . . 103
8.1.3 Is the pipeline too complicated? . . . 103
8.1.4 General Security in the Pipeline . . . 104
8.2 Discussing Our Providers . . . 104
8.2.1 Discussing Miles and Miles Connect . . . 104
8.2.2 Discussion Disruptive Technologies . . . 106
8.2.3 Discussing Azure . . . 107
8.3 Discussing the Guidelines . . . 107
8.3.1 A secure and reliable pipeline . . . 107
8.3.2 Following the guidelines . . . 109
8.3.3 Do developers need guidelines? . . . 111
8.4 Feedback From the Industry . . . 112
8.4.1 Adjusting the pipeline . . . 112
8.4.2 Feedback from Miles . . . 112

8.4.3 Feedback from DT 113

9 Conclusions **115**

9.1 Future Work 116

9.1.1 Benchmarking Miles Connect, and the system 116

9.1.2 Real use case 116

9.1.3 Evaluate guidelines as a resource 116

A **119**

B Source code, Image Sources and Project Poster **125**

B.1 Source Code 125

B.2 Image Sources 126

B.2.1 Kafka Virtual Machine specifications 126

B.2.2 Illustration Sources 126

B.3 Poster 128

Bibliography **131**

Abbreviations

ETL	**E**xtract **T**ransform **L**oad
ESB	**E**nterprise **S**ervice **B**us
IoT	**I**nternet **o**f **T**hings
iPaaS	**i**ntegrated **P**latform **as a** **S**ervice
BI	**B**usiness **I**ntelligence
SOA	**S**ervice **O**riented **A**rchitecture
DT	**D**isruptive **T**echnologies
QoS	**Q**uality of **S**ervice
ASIC	**A**pplication **S**pesific **I**ntegrated **C**ircuit
LPWA	**L**ow **P**ower **W**ide **A**rea
LPWAN	**L**ow **P**ower **W**ide **A**rea **N**etwork
MITM	**M**an **I**n **T**he **M**iddle
SDS	**S**ecure **D**ata **S**hot
AMQP	**A**dvanced **M**essage **Q**ueue **P**rotocol
SAS	**S**hared **A**ccess **S**ignature
HTTPS	**H**ypertext **T**ransfer **P**rotocol **S**ecure
SSL	**S**ecure **S**ockets **L**ayer
TLS	**T**ransport **L**ayer **S**ecurity

Chapter 1

Introduction

Internet of Things (IoT) is a technological concept where hardware embedded with sensors are connected to the Internet, allowing other devices or software to employ data collected by sensors. In other words, IoT connects *"things"* that gather information, to *"things"* that act on it.

Businesses across the world are experimenting with new ways to use IoT in order to gain a competitive advantage in today's business environment. The possibilities for an innovative business is endless. The problem of exploiting these possibilities, however, is often a lack of specialization in IoT. In terms of both working with IoT and knowing where to invest ones money. Data generated with the help of IoT will not benefit businesses if it is only stored and never acted on.

Taking advantage of the opportunities presented by this data, and extracting the value, necessitates a system that is capable of handling the data from production at sensors, to consumption. This system is called an IoT data pipeline or simply IoT pipeline.

One of many applications that exist and takes advantage of IoT solutions, is in agriculture. Sensors can notify farmers when their crops need watering, preventing over or under-use of the irrigation system. This itself would free up time, labour and capital for the farmers [1]. Going further, the irrigation system can act on the data from the sensors, turning on when the soil becomes too dry. Connecting this system to the Internet allows for even greater savings, as it can take into account the weather forecast. The irrigation system can choose not to water the crops if rain is expected to fall later in the day.

By connecting more and more sensors to the pipeline, the farm becomes more and more autonomous. This ultimately allows the farmers to produce equal amounts as before, or even more, with less work.

Setting up an IoT pipeline as described above requires specialization. Therefore the best option is to rely on outside expertise. This results in many providers of IoT and IoT processes, all offering different solutions to the problem at hand.

When examining the largest technology companies in the world, such as Microsoft, Amazon and Google, and the services they provide, one will find that all of them have made their own versions of an IoT pipeline. These versions shared goal is to optimise their own data platforms for IoT, but their approach to this goal varies. Whilst some vary on naming conventions, others vary in the various services employed. This ultimately means that each of these companies in general offers the same services, but claiming them as their own unique solution.

Despite the number of different providers, there is a lack of knowledge in regards to what one should take into account when deciding which vendor to choose, and how to set up an IoT pipeline that best suits the needs of a given use case.

1.1 Background and Motivation

The telecom company Ericsson wrote about the massive amount of data generated from devices in IoT, stating that *"... frameworks with clear IoT requirements to enhance the computation capabilities of distributed systems are still rare"* [2].

The tech magazine FutureIoT discusses what an IoT pipeline needs, and how it should be set up for maximum reward. Two notable quotes are:

> *"The pipeline needs to be able to integrate data from various sources in different formats. So, it has to be an agnostic pipeline that you can expand for other protocols as well"* [3].

> *"[the pipeline] should be capable of performing data processing such as transforming and filtering your data in order to increase the quality of your data at an early stage"* [3].

The latter suggestion would result in any null or non-numeric values being deleted as soon as possible, making it easy for any employee to deal with the data later.

Furthermore, the chief IoT technologist at Deloitte Digital, claims that *"Every organization that collects data from connected devices has a data pipeline, even if they do not know it"* [4].

Today common IoT core services are Azure's IoT Hub, AWS IoT core, and Google Cloud IoT core. There also exist some IoT services that are open source, such as Apache, MongoDB, IBM and Oracle.

From these findings one can see that most companies that extract data from connected devices utilize a form of IoT pipeline, and that there are many options when reaching out to vendors for products and services. However, in order to choose wisely, certain requirements for a given IoT pipeline is necessary, especially when realizing that the number of IoT devices and IoT software services are rapidly increasing.

When evaluating the existing software engineer guidelines, they often include best practices for coding, naming convention of functions, and recommendations of how to set up tests. Similarly for data pipelines, topics such as cost, complexity, and scaling are mentioned as things to look out for. These are either too specific best practices or too vague topics. *To watch out for* means that the developer needs to spend much time researching these topics in order to understand what is actually being demanded. Relevant guidelines that are specific in regards to demands, but not so specific that they themselves become reliant on the language or the parts used, is needed. There exists guidelines and standards for IoT devices, but limited to security issues.

Therefore, for an IoT data pipeline, there is sparsity in regards to these resources. To the best of our knowledge no such list of guidelines for IoT data pipeline development exist per today.

Guidelines are valuable during both the architect-, and development phase. Ideally, the software developers of an IoT pipeline would by simply following the recommendations, easily choose and implement their own IoT pipeline, avoiding common pitfalls.

To ensure that a certain standard of quality is created in this thesis, characteristics of what makes an IoT pipeline a *"good"* pipeline are presented. A *"good"* pipeline is defined to contain definite Quality of Service (QoS) requirements suggested in an extensive report [5]. These requirements include reliability, performance efficiency, portability and security. A detailed description of these requirements are found in the terminology section in Chapter 2.

1.2 Problem Definition

Given the overflow of IoT pipeline vendors and lack of standardization, guidelines regarding the requirements of an IoT pipeline are needed to be able to create a secure and reliable IoT pipeline.

As mentioned, IoT technology is in high demand and there are a lot of providers who offer their services to be included in a pipeline. The problem is, however, that it is difficult to properly develop, and handle an IoT pipeline to get the desired result. As more parties seek to introduce IoT devices to their solutions, security ought to be ensured to protect the consumers. Additionally, it is important to provide the client with a reliable and efficient solution, without degrading the productivity of the business.

1.3 Objectives

The objectives of this thesis are:

- Identify and characterize the different parts of a typical IoT pipeline.
- Locate areas of concerns for each part of an IoT pipeline.
- Present recommendations in the form of guidelines that are important when setting up each part of an IoT pipeline.
- Build and implement a working IoT pipeline using Miles Connect, a framework used for operations on data in applications.
- Use the guidelines to test and evaluate the IoT pipeline on real world applications.
- Evaluate if the guidelines themselves are useful and effective.

1.4 Approach and Contributions

In this thesis a literature study is conducted to find the requirements for an IoT pipeline system. These requirements will form the foundation for the proposed guidelines. In conjunction with this, we have built an IoT pipeline in accordance with the mentioned guidelines, using the Miles Connect integration framework. The Miles Connect framework handles task scheduling, integration, validation, logging and formatting. The pipeline is validated up against the guidelines with real world use cases.

We suggest 22 guidelines for creating an IoT data pipeline. In addition, this thesis will examine how the Miles Connect framework would work as a part of an IoT pipeline.

1.5 Outline

After the introduction in Chapter 1, Chapter 2 starts by introducing the terminology that is used in this thesis.In this chapter, the various parts in an IoT pipeline is defined and explained in detail. In addition, preliminaries that will give contexts around concepts that are discussed and worked with, are included.

Chapter 3 explores related works in the subject area: Timon, Kafka and Stream Bench, as well as examining the demands for an industrial pipeline. This gives a sense of what existing work there is today on similar topics.

Chapter 4 presents the different requirements in each of the pipeline steps, ending each section with relevant guidelines. In Chapter 5, the pipeline's architecture is designed in regards to the previously presented guidelines.

After presenting the architecture, Chapter 6, concerns the actual building of the pipeline, before testing and validating said pipeline in Chapter 7.

In Chapter 8, this thesis evaluates the tests, as well as the guidelines, and concludes the thesis in Chapter 9.

After that follows attachments in Appendix A and the project code in Appendix B.

Chapter 2

Preliminaries

This chapter presents preliminaries that will give contexts around concepts discussed and worked with in this thesis.

2.1 Quality of Service Terminology

This section lists Quality of Service (QoS) terminology used in this thesis. The following descriptions are how the terms listed below are interpreted and used in this thesis, given the context of IoT, data, and software engineering. These are also the QoS requirements the pipeline should adhere to.

- **Security -** concerns the domains of confidentiality, integrity and authenticity
- **Reliability -** takes into account availability and fault tolerance
- **Portability -** an agnostic, adaptable and modular approach
- **Performance efficiency -** efficiency regarding time utilization, resource usage and capacity
- **Confidentiality -** data, objects and resources are protected from unauthorized viewing and other access
- **Integrity -** data is protected from unauthorized changes to ensure that it is reliable and correct
- **Authenticity -** the user's identity is verified and trusted
- **Agnostic -** a system that is independent of a specific provider and interoperable among various systems

- **Availability -** authorized users have access to the systems and the resources they need
- **Fault tolerant -** the capability of a system to suffer a fault, but continue to operate
- **Value and Insight -** the system's capability of providing value to the customer

2.2 Technology

2.2.1 Internet Of Things

The Internet of Things, also known as the industrial Internet or the Internet of Everything, is the idea and technological advancement of taking physical locations and things, and connecting them to the Internet. When these physical things are connected to the Internet they can transmit or receive data, or both. The ability to exchange information makes the things smart, and have benefits that compound on each other.

Collecting and sending data through the Internet, and receiving a response means that physical devices can act and react to real world events, without the need for human interaction.

When a global network of machines, smart devices, monitors and sensors are capable of interacting and affecting one another, without human interaction, it enables what is called the fourth industrial revolution [6].

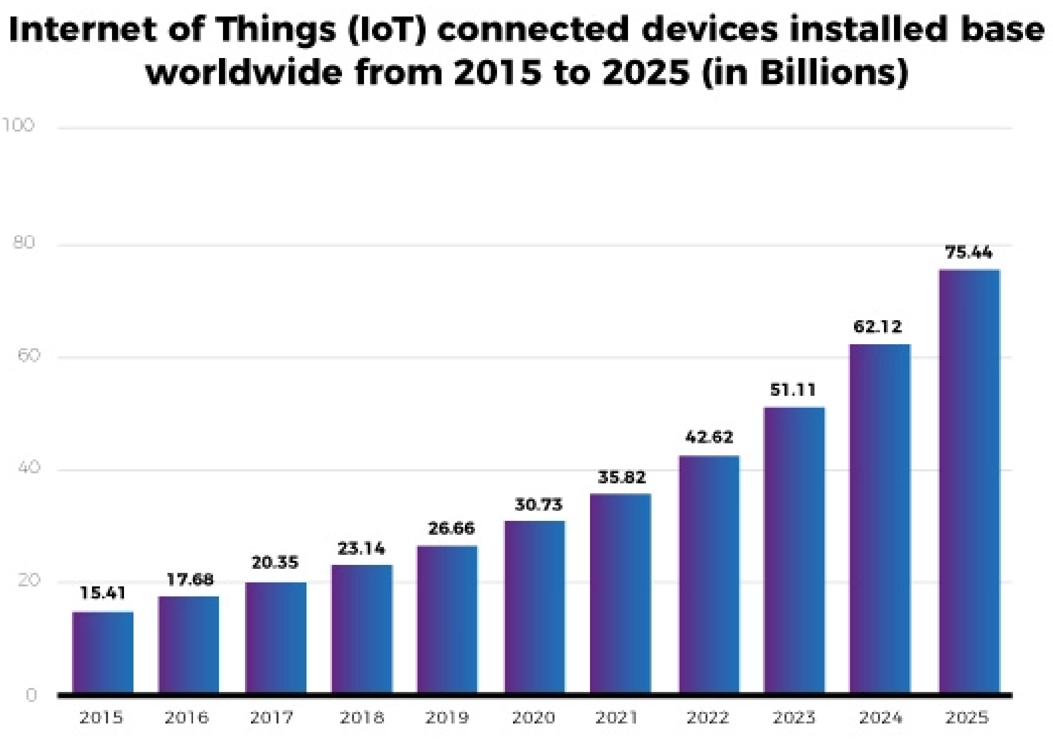


Figure 2.1: IoT Connected devices from 2015 to 2025[7]

In 2009 there were an estimated 0.9 billion IoT devices in use [8]. At the end of 2021 over 30 billion IoT devices were in use, and the number is estimated to be between 38 and 75 billion connected devices in 2025 [9], illustrated in Figure 2.1. This indicates that the IoT technology is rapidly gaining momentum with yet unseen potential, generating enormous amounts of data, 79.4 Zettabytes (10^{21} bytes) according to the International Data Corporation.

The value of IoT is realized when these connected devices communicate with each other, and are integrated with business intelligence applications, business analytics, and other service systems as inventory management [8]. Additionally, if used correctly, IoT can lower the cost of doing business. However, given the massive amount of data generated, the processing power required to extract value from IoT data may be significant.

Within IoT there are multiple technologies used for connecting devices such as WiFi, Bluetooth, ZigBee and LoRaWAN. The latter focuses on low power consumption and communication over long distances. From one of the leading telecom companies in Norway on the subject of batteries in sensors they emphasize that *"Batteries are cheap, but changing them isn't"* [10]. Thus, these sensors often transmit little data at steady intervals and go in sleep-mode between the intervals.

Another focus area within IoT is high performance. This is best described as technology used in self-driving cars. These requirements differ from Low Power Wide Area (LPWA) solutions in that a large amount of data must be transmitted with a low latency. Because the IoT ecosystems are constantly expanding and the number of connected devices appears to be increasing, it is critical to use software that can handle high-speed data traffic while keeping costs low. In these settings, low power consumption is not as critical. Technologies within this focus area is 5G and similar.

2.3 Major Vendor's IoT Pipelines

There exists no established standard for what parts an IoT pipeline should contain. As mentioned there are several providers of pipeline technology, including a few open-source ones. Our research found a few providers which define their IoT pipeline with various number of steps, and different naming conventions.

The three major vendors all present varying IoT pipelines, focused around their own IoT solution. Figure 2.2 present how the largest actors within cloud technology present their solutions. All solutions start with an IoT device, for example a temperature sensor.

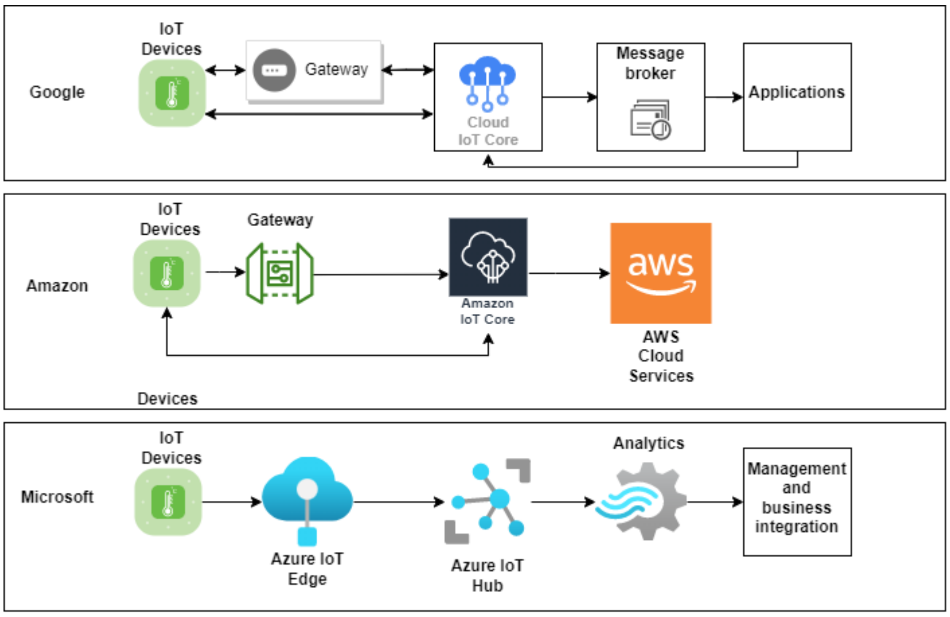


Figure 2.2: IoT pipelines from major vendors

2.3.1 Google

Google's IoT pipeline is focused around the Cloud IoT Core system. IoT Core connects, manages, and ingests data from IoT devices. Google's IoT pipeline starts with IoT devices, communicating with the Cloud IoT Core either directly or via a gateway. A gateway is a device that collects data from a multitude of IoT devices, before forwarding it to the Internet. Cloud IoT Core sends messages to a publish-subscribe message broker. A message broker is the system in charge of delivering messages to their destination. Varying applications subscribe to the messages from the message broker, and if any applications need to send info back to the sensor, they send it directly to Cloud IoT Core which in turn communicates back to the sensor. Applications can be analytical processing, business integration solutions, storage solutions or similar [11].

2.3.2 Amazon

Amazon Web Services base their pipeline around Amazon IoT Core, which connects and manages IoT devices. Amazon IoT Core can operate as a publish-subscribe message broker between IoT devices themselves, or operate as an ingestation step for the gateways to the rest of the AWS Cloud Services [12].

2.3.3 Microsoft

Microsoft's Azure uses Azure IoT Edge as a gateway between the IoT devices and Azure IoT Hub in the cloud. In addition to being a gateway it opens up for edge processing on the IoT Edge devices. Edge processing is when you process data close to where the data was produced, thus not needing to do the processing in the cloud. Data sent to Azure IoT Hub is forwarded through an analytical part before finally being sent to management and business integration logic [13].

2.4 Defining an IoT Pipeline

In order to give guidelines that will be generally applicable, all the varying IoT pipelines shown in Figure 2.2 are taken into account, creating a general IoT pipeline. This means that our definition of a pipeline will be generally applicable to all IoT pipelines, however not all IoT pipelines will need separate parts for each section in our pipeline.

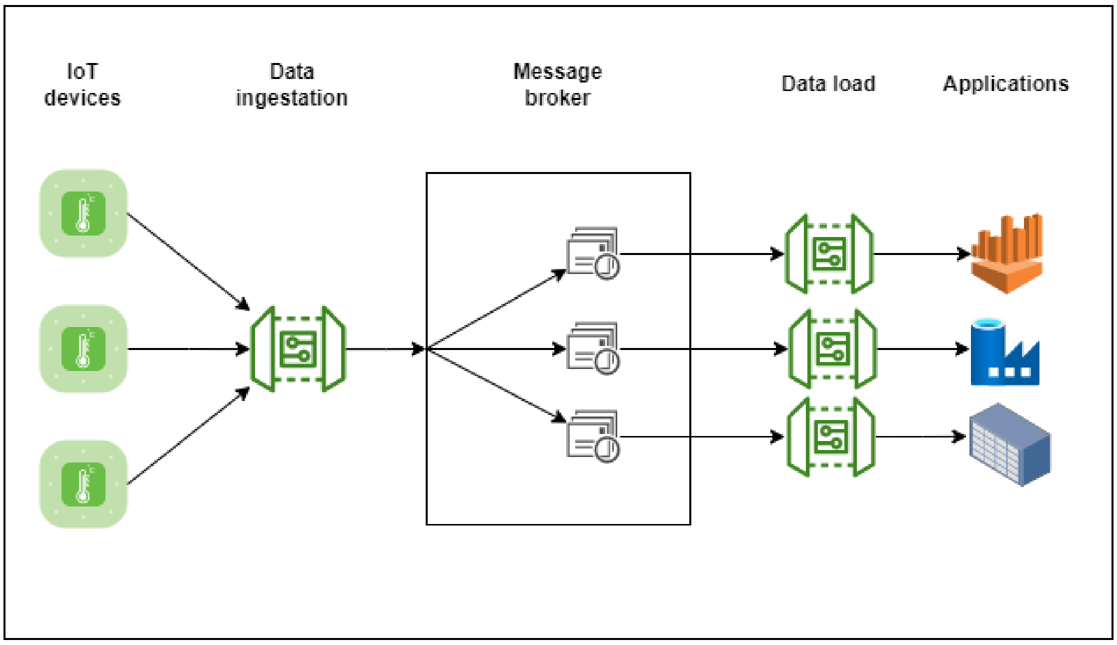


Figure 2.3: Pipeline definition

The pipeline is divided into five separate steps: IoT devices, Data ingestation, Message broker, Data load, and Applications, as shown in Figure 2.3. Each part has specific demands, needs and responsibilities. All parts must be present, either separate or together, in one form or another to have a fully functioning IoT pipeline that provides value. The IoT devices generate data to be consumed and analyzed. Data ingestation retrieves the data, sending it to the message broker, with relevant validation and formatting. The message broker provides a way for multiple applications to listen to the incoming data, and use it as intended. The data load framework is the way the data exits the message broker to the relevant application. The applications handles storage, analytics or any other task to provide value. As there exist many different systems, some systems incorporate multiple parts of this pipeline, while others incorporate only one part. Thus, when building an IoT pipeline one must choose systems that combined go through all these steps.

This section goes through each of the steps mentioned above that is necessary in a given pipeline. Each subsection gives an introduction to each step. In Chapter 4, the varying requirements for the steps are introduced.

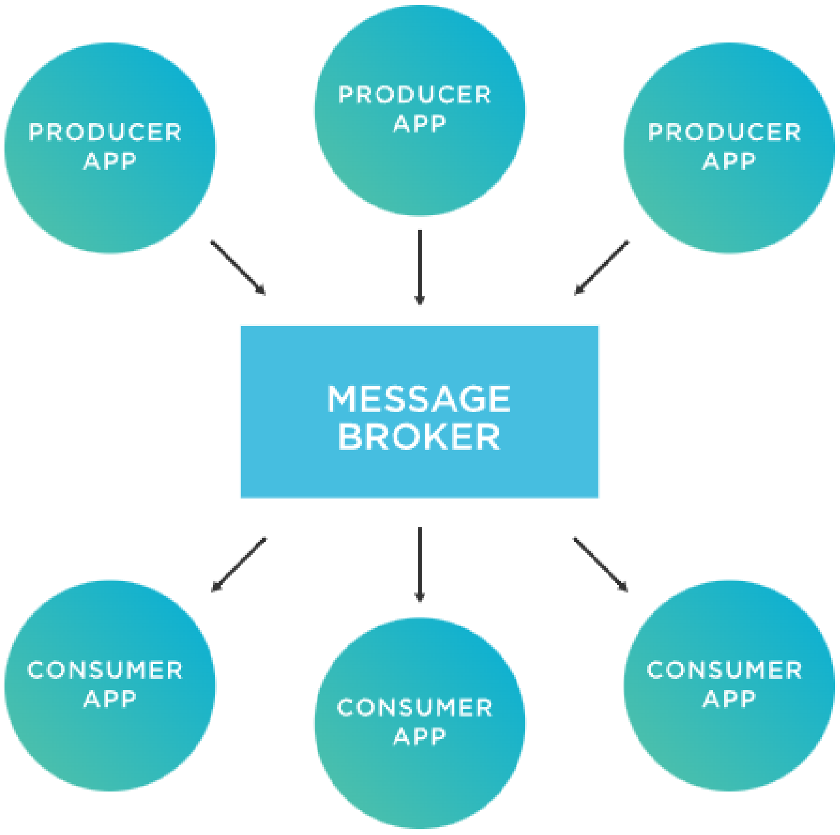
2.4.1 IoT devices

Devices that measure environmental effects, machines that automate factories, and gadgets found in smart homes are hardware equipment that make up the "things" in the Internet of Things. In a smart home or office space, one may have a handful of unique devices that each monitor and operate different tasks. Connecting them to the Internet and establishing communication with other things and applications, is when the solution becomes smart. IoT devices can be used in a personal, business or industrial setting. In 2021, the top use cases for IoT devices were remote monitoring, process automation, location tracking and vehicle fleet management [14]. IoT is mainly used in factories as of now, but can offer extensive aid in areas such as smart cities, personal health and agriculture, to name a few. The communication between IoT devices can happen in many forms, devices can transmit their data by using technologies such as Wi-Fi, Bluetooth, 5G, LoRaWAN or ZigBee [15].

2.4.2 Data ingestation

Data ingestation is the step where data is sent from the sensors and inserted into the message broker, often using gateways. A gateway is hardware that is directly in contact with the IoT sensors. Gateways are introduced to help IoT devices to stay small, cheap and battery-based. The gateway locally processes the data and forwards any messages to the Internet. An example is a smartwatch connected to a smart phone via Bluetooth, where the phone acts as a gateway for the watch. After receiving the data, it must be formatted and validated before put on the message broker. This is to ensure that the listeners on the message broker receive the message they listen for on the correct format.

2.4.3 Message broker



A message broker has the responsibility of receiving and sending messages across platforms. They must work with a formally established message format. It also has the characteristics of allowing multiple applications to receive the same message, with each application having sole ownership to its own message. As a result, the applications can process the messages asynchronously. A message broker also has a form of structure in order to prevent messages being lost, like a dead-letter queue. Choosing an ineffective message broker can be the limiting factor in terms of latency and scaling. Aside from handling multiple applications, the message broker must also provide reliable storage, guaranteed message delivery, and transaction management as needed.

A further critical consideration is that the message broker chosen must be capable of handling the various communication paradigms that can be present in the pipeline. A few available options are batch processing, publish–subscribe, request–response, near real-time processing, or fire and forget. The message broker must also lay the groundwork for event-driven architecture in order to scale distributively on cloud, or in a hybrid solution. Using an event-driven architecture also enables event-driven processing of big data, which can easily be generated by IoT sensors, if allowed to. A quick online search reveals that over 30 message brokers are currently in use, including Apache Kafka, Azure Service Bus, IBM, and Amazon. With so many options, it is important to select a message broker that fulfills the needs of the pipeline, and there is no reason to settle for the second best.

2.4.4 Data load

Data load is defined as the process of transferring data from the message broker to storage or other applications. Such an application can for example include analytical tools, like machine learning, or other value-adding or legislative applications. Data load is responsible for ensuring that the data arrives at the destination specified in the message broker service. This step, like ingestion, includes a formatting and validating part to ensure that the application receives correct data on the correct format. Because the message broker uses a unique format to transfer the data, a new formatting may be required to meet the requirements of the given application.

2.4.5 Applications

The application section is where the captured data from the IoT sensors will be put into action to provide insight for the user. The application's possibilities are endless, and offer a large range of variety and use cases, each with their own requirements depending on the job to be done. The most common applications include storage and analytics.

2.5 Other Relevant Preliminaries

Further on in this section other relevant preliminaries are presented. These preliminaries are referred to in later chapters, and give context around the subject.

2.5.1 Data silo problem

A data silo is when you isolate source data. This can happen unintentionally or intentionally. It is however, not necessarily a problem with many data silos, the main problem with data silos comes when they are not connected to each other. When not connected, someone must go through different sources, connect them and provide insight. This leads to wasted resources such as manpower and time. Data silos often occur in large organizations but it can also happen inside a department. The opposite of multiple silos is a single source of truth, which is deemed more effective, as every department that needs certain data knows where to look for it [16].

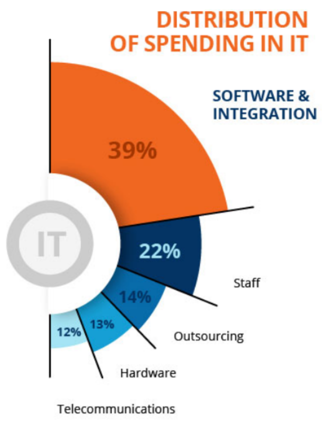


Figure 2.4: IoT spending distribution in 2020 [17]

This however requires a robust integration strategy which may be expensive, in 2020 39% of spending in IT went towards software and integration [17].

2.5.2 LoRaWAN

LoRa technology sends small portions of data at a low bit rate, but at a larger distance than other wireless technologies, such as WiFi and Bluetooth. The LoRaWAN is a protocol at a software layer which details how to make use of the LoRa hardware devices including formatting and frequency of messages [18]. In a LoRaWAN network each device uses minimal power consumption and can operate up to 10 years without changing batteries.

The signal sent from these devices can reach distances from 3–10km, based on the surrounding environments. LoRaWAN is cheap, has open-source software and requires no license to deploy a network. A Norwegian municipality, Asker, has started using a LoRaWAN network with thousands of sensors to monitor household waters for 3000 houses [19]. Last Mile Solutions, the company in charge of this network in Asker, believes that IoT with LoRaWAN have several possible applications for making monitoring of municipal services more efficient [20].

2.5.3 Enterprise Service Bus

An Enterprise Service Bus (ESB) is a form of message broker using the publish-subscribe communication paradigm. ESB is often used in a service-oriented architecture (SOA). This type of architecture consists of multiple software applications that are reusable and inter-operable through the use of service interfaces. Because of the decoupling provided by these interfaces, little to no knowledge of how the service is implemented is required [21]. The ESB is the component that allows applications to communicate with one another while also ensuring that messages are delivered in the expected data format [22]. In addition, rather than having an interface for each application, the ESB ensures communication through a single interface. It is a subset of the client-server model, where each application can act as a client or a server. ESB employs a message queue system on the bus so that each application can publish their message to the bus. This relieves the application of responsibility for delivery, allowing it to focus on other tasks. Another advantage of the ESB is that it adds an additional layer of security. Each message is authenticated by the ESB before it is sent, and it can even log actions to another application component, if desired. A centralized ESB's goal is to standardize and greatly reduce the complexity in communication, messaging, and integrating services between enterprise systems. From a business standpoint, the time previously spent integrating various applications can now be spent on improving the applications. Furthermore, because the integrations are reusable across projects, higher productivity and lowered costs are possible.

2.5.4 Integration Platform as a Service

Integration Platform as a Service (IPaaS) is another way of dealing with the same issues as ESB. IPaaS has the added benefit of being scalable on cloud while at the same time enabling integration with off-cloud integration. This represents the new generation of integration as it is transforming the on-premise integration, allowing the service to be subscribed to as any *aaS service [23].

2.5.5 Software and hardware providers

Miles AS

Miles AS is a Norwegian IT consulting firm founded in 2005 with offices in the four largest cities in Norway. In this project, we conduct our development, research and experiments on behalf of Miles AS in Stavanger. Miles is also the product owner of the Miles Connect framework that will be used in this thesis.

Disruptive Technologies

Disruptive Technologies (DT) is a tech company based in Norway and produces small wireless sensors and IoT infrastructure. Their sensors operate with their own design for an Application Specific Integrated Circuit (ASIC), named DT Silicon. This technology allows their sensors to use ultra-low power which is 100 times more power-saving than other market-leading Bluetooth controllers [24]. DT's products have led them to win several awards including Sustainability Product of the year 2021 and IoT Sensor Company of the year 2022.

2.5.6 Security protocols

In this section security protocols used in this thesis is presented.

HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is a secure transfer protocol that is used in communication between a user's personal computer and a website. The goal of HTTPS is to ensure encryption, authentication and integrity [25]. The content of messages are secured by encrypting the data with a public-key cryptography, which strengthen the defense against various cyber attacks.

HTTPS also includes a Secure Sockets Layer/ Transport Layer Security (SSL/TLS) protocol, which is a website's certificate that includes a public key, that is used to confirm that the documents on a server is legitimate. The documents can be confirmed as safe by a digital signature signed with a legitimate person's private key. When enabling HTTPS to a website, a certificate is issued by a certificate authority.

The documents such as images and code files that are stored on a server through HTTPS, have a digital signature that proves that the data has not been altered during transportation. If any of the data has been modified or corrupted, it will be detected [26].

2.5.7 Shared Access Signature

The .NET packages used for communication with our storage in Azure are *Azure.Messaging.ServiceBus* and *Azure.Storage.Blobs*. They are both verified as official packages in .NET package manger, and are published by Microsoft. When using these packages, we also use the official documentation provided by Microsoft on how to correctly set them up and enable their safety conditions. Shared access signature (SAS), is a security protocol used within these packages. When using SAS, keys are used to cryptographically sign information that can be verified by the service in use [27].

Chapter 3

Related Work

This chapter introduces literature and research on data ingestation benchmarking, building, and testing data pipelines in order to provide an insight to existing work on similar topics to our thesis topic.

3.1 Timon

Timon is a testing framework, aimed at testing the configuration of IoT pipelines [28], in order to find the best configuration of the specific pipeline. The framework focuses on testing multiple configurations of the same system, which makes it relevant in order to optimize a single pipeline. It is however, pipeline-specific, such that using Timon to compare varying pipelines would be problematic. For an industrial use case where a pipeline already exists, Timon would be of great support to optimize said pipeline. While our thesis focuses more on choosing the correct pipeline amongst several options.

3.2 Kafka

Apache Kafka is the most popular open-source event streaming platform to date. From a list of the top 100 most successful American companies, more than half of them use Kafka as their data stream service provider [29]. Similar solutions include Google Cloud Pub/Sub, RabbitMQ, IBM MQ, Amazon MQ and Amazon Kinesis [30]. In one study where Kafka was examined, it focused largely on the insert rate factor [31]. In another study covering the same technology, it examined how parameters affected the performance. [32].

3.2.1 Kafka benchmark

The first paper evaluates Kafka by benchmarking the number of messages processed per second [31]. In the study they put three identical setups on three servers, inserting data into Kafka as fast as possible in order to benchmark it. The second paper mentioned tested Kafka in sections to clearly identify bottlenecks [32]. The authors separately test the process with producers and consumers. They also utilized a variety of parameters in these tests and list a set of performance metrics. Parameters to include and adjust are message size, batch size, acquisition strategy, hardware used, and network threads. Performance metrics to record are throughput, latency, CPU usage, disk usage, memory usage, and network usage.

These studies are both of significant relevance, but only as a portion of our validation section for our guidelines. The scenarios in these papers focus on inserting user-created documents, whilst we focus on IoT data collection. Reading these papers gave us great insight to benchmarking the insertion of data, and impact of different parameters, and will be taken in account when creating our tests in this thesis.

3.3 Stream Bench

Stream Bench [33] is a framework for benchmarking streaming services. The authors look at multiple streaming services and seek out to establish criteria for these modern distributed stream processing frameworks. Stream Bench aims to take an early step towards establishing a benchmark for the mentioned frameworks. They employ their framework to Apache Storm and Apache Spark to see the impact. They highlight the challenge of massive data generation on the fly and point out to take both burstiness and durability of stream computing as the target of measurement. In their tests they measure how performance, fault tolerance, ability, and durability act during workload of streaming processes.

A challenge with streaming is that a computation framework can grow as needed, and if the input is not scaled, the computation ability of the system may be overshadowed by bandwidth bottleneck or undersized input, this results in loss of preciseness of benchmark. This challenge lays the foundation for why Stream Bench was created. However, these solutions mainly focus on streaming frameworks, whereas we have a more general approach in regards to our contributions.

3.4 Industrial Pipelines

A paper from 2020 [34] introduces a framework for designing data pipelines for manufacturing systems where they depict possible technologies available for such a pipeline. In conjunction with this, the authors comment on the pipeline design: *"Frameworks for data driven manufacturing are available, but they are not useful for guiding one how to design a data pipeline."* Similarly, we encountered an issue that supports this critique: most researchers present a pre-designed data pipeline as a framework or architecture they use, without including the process of design. The authors deliver a simplistic framework for selecting solutions for each step in a pipeline, and a guideline for how to use their framework. Their research provides us with inspiration and insight in their process of developing a pipeline with focus on manufacturing processes.

While applying an IoT pipeline to manufacturing facilities that mostly operate with legacy equipment, numerous challenges are expected to occur. These challenges are discussed later in a paper, where also the possibilities of introducing such a pipeline in this scenario are introduced [35]. In the paper they list a number of criteria for their suggested pipeline which includes fault tolerance, accessibility and scalability, which are also applicable for our use case. Especially on scalability they mention the rapid expansion of digitization and that the number of connected devices to be used may be unknown, thus dynamic scalability is a desired attribute in such a system.

Even though the authors have provided a thorough research on the topic of industrial big data pipeline, they merely suggest a single pipeline for a manufacturing scenario, where we aim to introduce general guidelines for pipelines.

Chapter 4

Presentation of Guidelines

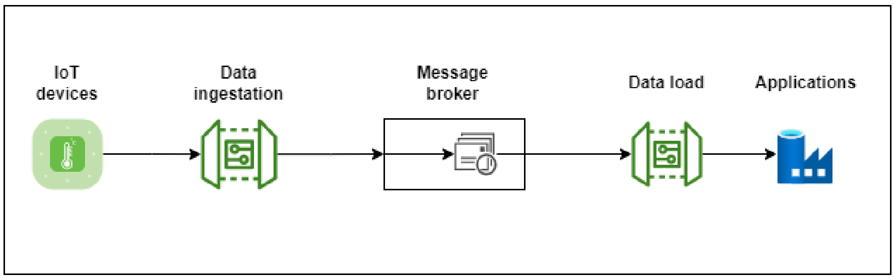


Figure 4.1: The five steps in an IoT pipeline, used to illustrate guideline affiliation

This chapter explores the requirements of the different steps of an IoT pipeline. In addition to this, it presents the guidelines that should be adhered when building and implementing an IoT pipeline.

4.1 Methodology

The methodology used to locate and define relevant guidelines in this chapter is threefold: 1) lessons learned from real world examples, 2) a literature study from relevant academic sources and 3) speaking with industry experts.

When a pipeline is created it is often to solve a specific use case, and an IoT pipeline is no different. Guidelines that come from real word applications and use cases grant depth and relevance. By following them, one often avoid practical problems not thought of before the pipeline is in production, and it is too late. Thus, in order to define and present important guidelines, studying issues that have arisen in actual IoT pipelines are conducted, meaning that some guidelines come from *lessons learned*. This ensures that the guidelines prevent repeating previous mistakes.

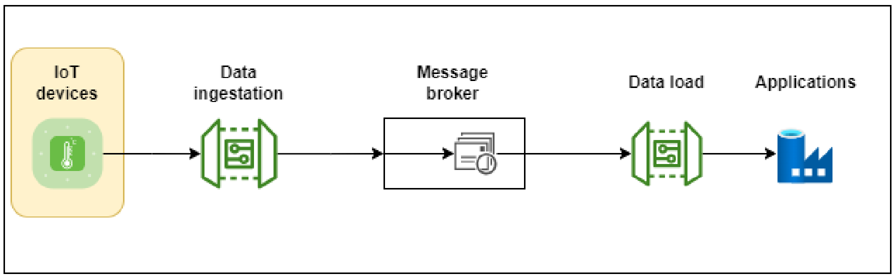
To gain theoretical knowledge, a literature study is conducted on relevant papers. These papers describe how things are done in the past, as well as relevant technology. These are helpful when introducing new guidelines that are better suited for technological advancements. This results in better quality of the guidelines, in turn preparing the IoT pipelines for the future.

In addition to this, talking to industry experts from Disruptive Technologies, Miles and Miles' customer, Asplan Viak, provides this project with relevant knowledge of the IoT area at the current moment, and learn how it is used today.

With these three approaches, the guidelines cover many of the relevant aspects when building an IoT pipeline.

The guidelines will be presented in sections based on the parts of an IoT pipeline as defined in Chapter 2. The methodology described above is used when creating the guidelines.

4.2 IoT Devices



To maintain the communication and stability between the IoT devices and the rest of the system, configuration, authentication, and the software must be up-to-date. Issues that occur when dealing with IoT devices is lack of security, overloading the device or lack of connectivity [36].

4.2.1 IoT security

When dealing with a large number of wireless devices, IoT security is critical because any breach at any of the many sensors will compromise the solution's integrity.

As recently as 2017, the Norwegian Consumer Council discovered significant security and privacy flaws in smart watches for kids sold from many known vendors in Norway. This enabled strangers to take control, eavesdrop and communicate with children via their smartphone. This could be done without their parents' realization. Furthermore, they could alter the clock's GPS signal leading parents to believe the clock and child were in a different location than they actually were. In addition, data stored on the watch was not encrypted, and it was impossible for users to delete their own data [37]. All serious flaws caused by poor choice of IoT equipment.

Encrypting the signals sent by the IoT device to the gateway with a standardized encryption algorithm is a low-cost and dependable way to prevent sensitive data from being read. With small and simple devices used for monitoring and transmitting data, the devices lack the software for built-in security, hence limiting the possible security measures. Another regard to security is budgeting, as many industrial companies want hundreds or thousands of sensors and want to keep the price per sensor low.

It is very common that devices do not enforce sufficient limiting in terms of access control. In particular when the access should be limited to the owner, or any legitimate user. Without any limiting factor any person connected to the network can access the device [38]. When vulnerabilities are discovered by developers, they roll out new and patched software updates for its devices. The installation of the latest software might be required to be executed manually, and administrators of devices should be up-to-date with the latest software [39]. Based on these findings the following guidelines related to IoT security are introduced. The format will continue in a similar manner for the following chapter.

Guideline 1.1. The IoT devices in use should be created by a trusted source.

Guideline 1.2. Users of IoT devices should be authenticated with modern standards.

Guideline 1.3. Common IoT security measures such as encryption should be implemented on the IoT device.

Guideline 1.4. To ensure that each IoT device has the latest software which patches security flaws, software used in IoT devices should have the newest version available.

Guideline 1.5. The access control of the IoT devices and data produced, should be limited to the owners or certified operators.

4.2.2 Signal strength

In some use cases, sensor data availability must be fast and stable, therefore signal strength must be prioritized in critical solutions or when sensors are located in poor signal strength zones. While in dead zones or other use cases where poor signal is a reality, such as underground parking or with flood sensors in basements, one must employ solutions to handle these environments. According to an article, as the development of IoT applications accelerates, reliable network connectivity becomes increasingly important for all aspects of IoT. Weak or intermittent cell signals can have an impact on inventory, revenue, and security [40].

Transmitting data is considerably power consuming, thus poor signal can have a direct impact on battery life as the sensor must transmit data over a longer period of time [41].

Guideline 2.1. Ensure that the IoT device and gateway are able to handle poor signal strength.

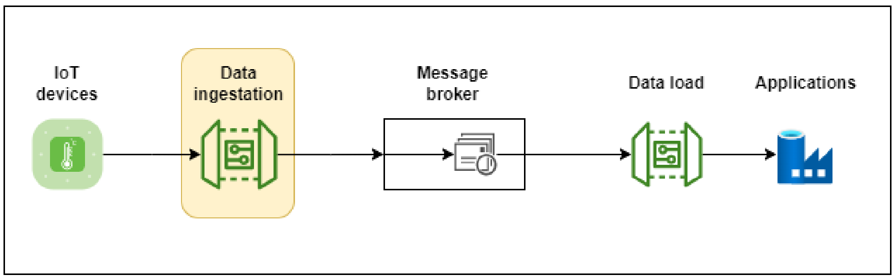
4.2.3 Device battery and connectivity

Device connectivity includes efficient communication, reliability, and that the solutions is fast to scale. As stated before, *Battery packs are cheap, changing them isn't.* Thus, it is ideal in a LPWAN scenario that each sensor does as little as possible to prevent excessive drainage on the battery. In another scenario, such as high performance IoT setting, the connectivity to devices is essential for making an IoT pipeline operational, so maintaining a stable connection should be prioritized. IoT device technology has improved dramatically in recent years. In addition to the sheer number of devices in use increasing, it is possible that the system can be overloaded due to the huge amount of data generated. Therefore device load and bandwidth in an IoT pipeline must be controlled by some form of scaling functionality.

Guideline 3.1. Employ a battery-saving scheme that ensures sufficiently long battery life for the devices.

Guideline 3.2. Control that the IoT device load and bandwidth is not exhausted, and scale resources if necessary.

4.3 Data Ingestation



The data ingestation part of a pipeline is responsible for connecting data producers to the rest of the pipeline and fully integrating them. In addition to this, it is responsible for validating and formatting the data in the proper format.

A study from 2002 [42] argues that integration of applications will be the major challenge to Application Service Providers (ASP) for their competitiveness in the modern business environment. Due to the wide range of software available that can be used to perform any job with little to no training, different departments within a company can choose their own app to meet their specific needs. Business processes like procedure-to-pay, quote-to-cash, and item management necessitate a large number of applications and departments in order to operate efficiently. All of these apps generate a set of new data while also requiring a data input, and as a result, data is stored within departments as ever-expanding data silos as discussed in Section 2.5.1. This can lead to organizational invisibility, management via e-mail or spreadsheets, and human errors.

The process of connecting all these different applications is referred to as integration. To handle these various applications and avoid data silo build-up and manual organization between departments, a robust integration automation strategy is required. Data ingested into a central message broker is an efficient way of addressing these challenges of integration. The way the ingestation process in the pipeline is represented is with the data ingestation step.

4.3.1 Data ingestation example

Major cloud service providers offer specialized software for data ingestation in IoT. Amazon as an example, uses AWS IoT as a data ingestation step. AWS IoT functions as a connector from IoT devices to the cloud, operating as a gateway between the devices and other AWS services [43], as shown in Figure 4.2

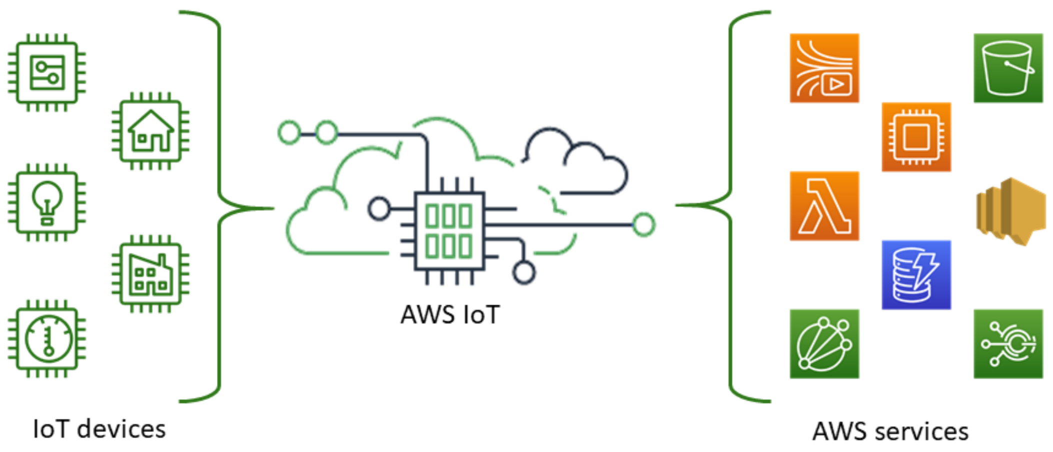


Figure 4.2: AWS IoT

In order for AWS IoT to work, the IoT devices need AWS software installed on them to communicate with AWS. This requires either choosing IoT equipment that already is optimized for AWS IoT, or installing said software in the IoT sensors in use.

4.3.2 Gateway

As gateways are used to collect data from IoT devices and send them forwards, they are a natural part of the data ingestation step. To start off, the gateway must be able to integrate with the varying sensors it will communicate with.

The gateway must be able to handle everywhere from one to many sensors. Thus a gateway that is able to handle the load it is assumed it will be under is needed. If the amount of devices is too much, multiple gateways must be able to work concurrently to share the load of the system. Each IoT device can also lose connection for varying lengths of time. This results in the message size varying greatly based on how much data was generated since last time it made contact.

The gateway also needs to be able to handle varying signal strength, as well as recognising what to do if it finds out that the data is corrupt. Data can be classified as business as usual, nice to have, essential or similar categories. Some data can arrive late, but never be lost, like logging data. Other data is nice to have, but can be dropped to manage load or reduce replication, examples of this is metadata. Hence, an IoT pipeline needs different mechanisms to handle different message types and priorities. This can be handled in the gateway as it is the entry point for data to the system. By adding an IoT gateway to the system architecture one can enable security measures such as encryption, tamper detection and user access management [44][45].

Guideline 4.1. The gateway should scale gracefully with a growing number of devices in the architecture.

Guideline 4.2. The gateway in the IoT pipeline should be able to connect to other gateways to expand the load of data, working in parallel.

Guideline 4.3. The gateway should be capable of performing edge computing tasks such as pre-processing, cleansing, and filtering.

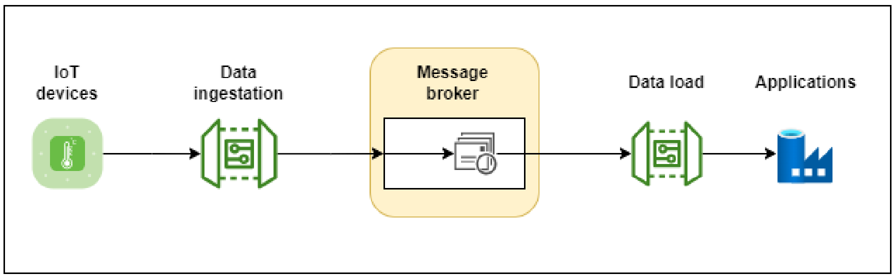
4.3.3 Early stage validation

To be able to put the data on the message broker the data needs to be in the correct format, with the correct tags. In addition to also making sure that the data is valid and not corrupt. Examples of validation can be checking for missing data, removing invalid or redundant data, and clean *noisy* data [46]. Such actions at an early stage will reduce the amount of useless bits of data from ever entering the cloud, which will make the system more resource efficient. Thus, the main features that need to be present is:

Guideline 5.1. The data ingestation step should ensure that the format of the data is correct so that the data transfer will work as expected.

Guideline 5.2. The data ingestation step should validate that the incoming values are correct and usable, removing any invalid data.

4.4 Message Broker



An important part of a pipeline is to transfer data from the producer to the consumer. In large organizations this is often done by a central message broker, to handle some of the integration challenges that lead to data silos. How data is transferred however, has changed over the recent years.

4.4.1 Transfer process

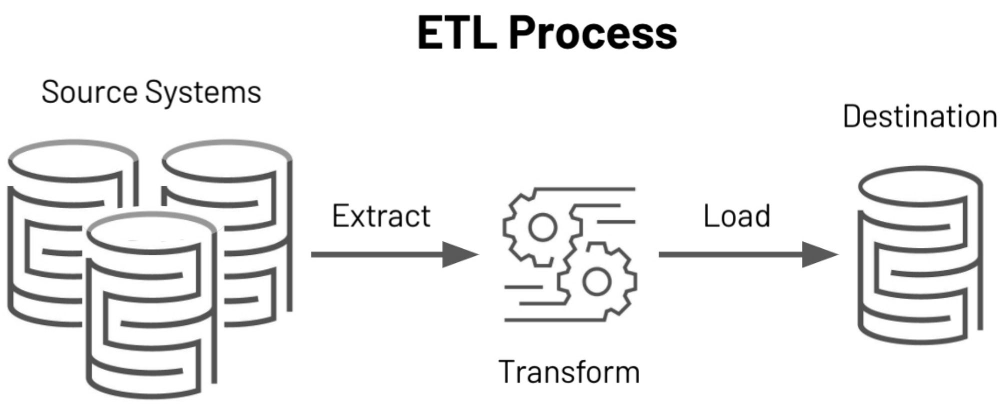


Figure 4.3: Extract, Transform, Load - Process

A common way to move data is via the Extract, Transform and Load (ETL) [47] method, meanwhile a new way of using event streaming has seen increasing popularity. The ETL approach is the general approach used when copying or moving data from one producer or storage to a data consumer, shown in Figure 4.3. It gained traction in the 1970's and has been popular since then, often used in data warehouses. In an ETL process, data is first extracted from the database of the source's system. The second step is to transform the data extracted to meet requirements of the target system. When transformed, the data is loaded into the target's database. This concludes the ETL process. From the target's database, varying forms of analytical processes can happen and give insight in the data via different Business Intelligence (BI) applications. A key challenge with this way of moving data is that all data needs to be structured, neatly organized in rows and columns. Unstructured data like audio and video however, struggles with this approach. There also exist issues with the actual processing speed and scalability [47].

The ETL process of extracting data can be thought of as pull queries, where the system asks what data exists in the database at this time. In event streaming, data is pushed instead, illustrated in Figure 4.4.

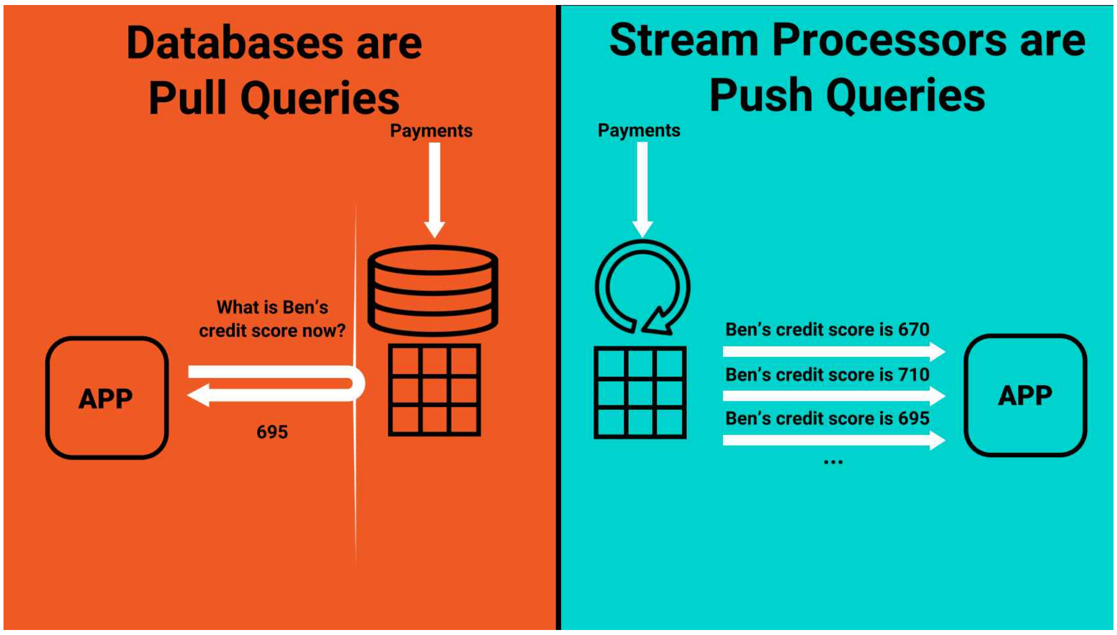

Databases are Pull Queries
Payments
What is Ben's credit score now?
APP
695
Stream Processors are Push Queries
Payments
Ben's credit score is 670
Ben's credit score is 710
Ben's credit score is 695
...
APP

Figure 4.4: ETL vs event streaming [48]

Event streaming allows us to treat all data in near real time by leveraging events rather than data stores as a core principle. In ETL, one can think of data as data at rest, something to be generated, transferred and analyzed. This prevents us from building agile and flexible services that can act on the data while it is relevant and newly generated. Using an event-driven architecture allows the consumption of data closely after the data is generated, and while in motion. This event-driven architecture gives much more flexibility when it comes to continuous analyzing, which is often what is desired in IoT.

As the data is generated continuously for a multitude of purposes, an ETL method would be a slow way to handle this. Thus, the event-based data ingestation method with a publish–subscribe communication paradigm is a better option. With this one can send the event to all relevant parts via a service bus.

Guideline 6.1. Data transfer should happen with event streaming or similar technology, avoiding ETL. This allows for fast and reliable data transfer.

4.4.2 Message broker responsibilities

As stated in Section 2.4.3, a message broker is the central authority from which all applications connect and communicate with each other. ESB and iPaaS are two techniques employed to move data from producers to consumers. Both ESB and iPaaS integrate systems and applications to share data between applications. IPaaS is centered around public and private clouds, trying to eliminate having to connect systems through on-premises software and hardware. ESB, on the other hand, are usually designed for on-premises integration rather than cloud integration.

In a system, message brokers can be used to validate, store, route and deliver messages to the desired destinations. *Most importantly brokers ensure that recipients receive the message even if they are not online or active* [49]. Message brokers utilize a queue system, so if any message is unsuccessful in its delivery, it will be delivered at a later point. A benefit of message queues is that applications can function asynchronously so that processes and applications are kept separate. So, in the event of a process or connection failure, the system remains active.

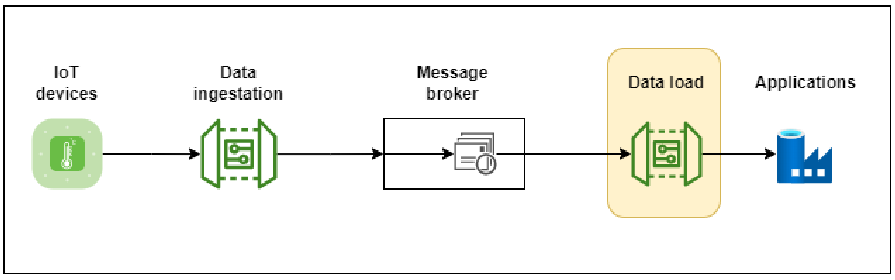
In an application workflow where senders and receivers are coordinated to send to, and read from a queue, one may be able to eliminate the dependencies between each step. This can be done with careful design principles. And if implemented, multiple receivers can process messages from senders in parallel [50]. With this in mind, the following guidelines are presented:

Guideline 7.1. Message brokers must ensure that the consumers will receive its messages by employing backup strategies. These strategies include external storage for message retrieval, and delivery protocols such as 'at-least-once'.

Guideline 7.2. Message brokers should have the ability to perform asynchronous processing.

Guideline 7.3. Message brokers should allow different applications to communicate regardless of the programming language.

4.5 Data Load



This step is responsible for moving data from the message broker to the specific application. The data load is defined as application-specific. This means that each application that wants the specific data, has its own data load step.

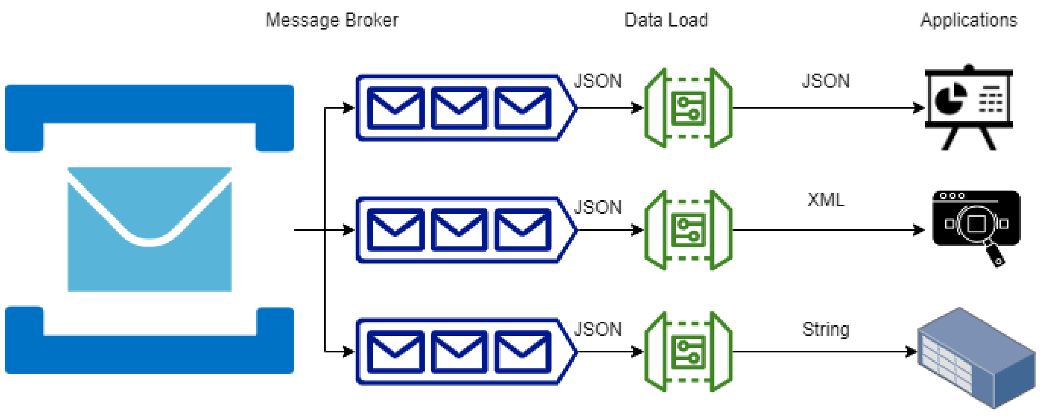


Figure 4.5: Data load step, illustrated

As can be seen in Figure 4.5 three applications want the same set of data to conduct operations on. The message broker creates three queues, one for each application. To illustrate the issue it is assumed that the first application takes JSON as input, the second XML and the third demands a string format. As the message broker is agnostic and does not know or care which applications demand what, a converting step is needed. This means converting the data from the format the message broker delivers to the format the application accepts. Additionally, an application-specific validation to ensure that no invalid data passes through to the end application. This validation can be much more thorough than the ingestation validation. This step may be optional if the application itself can do this, but given the vast amounts of available applications it is not possible to guarantee that every application in use can do this. Common data validation rules that can be application specific include [51]:

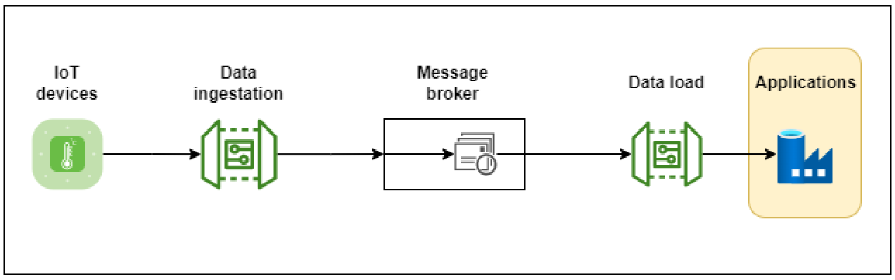
- Data type (ex. integer, float, string)
- Range (ex. A number between 35–40)
- Uniqueness (ex. Postal code)
- Consistent expressions (ex. Using one of St., Str, Street)
- No null values

A key element of this is to ensure that this extra step does not dampen the effectiveness of the pipeline while at the same time ensuring that the application receives quality data in the correct format.

Guideline 8.1. The data load step should perform small data processing validations to increase data quality.

Guideline 8.2. The data load step should conduct application-specific operations before delivery is complete.

4.6 Applications



On average each company uses 1 742 different applications [52], resulting in a growing threat surface and an inability for IT professionals to keep up with the ever changing architecture, platforms and standards. This has a direct impact on app security, visibility and compliance [52].

At the same time it must be acknowledged that each department in a company on average uses 30-60 different applications, and that applications bought directly by the departments that want to use them have higher engagement rates than those handed down by IT [53].

This results in a dilemma for tech leadership where the security risk, threat surface and amount of money the business spend on applications stands against engagement rates and consequently effectiveness of the employees.

It is reasonable to assume that the company implementing an IoT pipeline is not immune to these effects, and thus this must be taken into consideration when choosing relevant apps.

As there exists a tendency to use whatever apps are provided by a cloud provider to solve a certain problem [52], it is necessary to focus on application security. Hackers are targeting the applications more and more instead of the network [52]. The data load step helps mitigate this by decoupling the applications from the rest of the pipeline.

In addition, security breaches may come from different sources as well. A study from 2021 discovered that 63% of employees acknowledged that they used unauthorized applications daily to share files with co-workers [54]. This can be mitigated with a solid integration strategy, as Amazon is the most common example of, with their API- mandate from 2002, stating that all services shall be exposed to other teams via an API [55].

In general there exist a wast amount of applications to solve specific problems, thus the main issue becomes to choose applications that are company-approved, cheap and secure. As well as offering all services necessary for employees to prevent the use of unauthorized apps. As it is nearly impossible to control all applications regardless, security measures should be taken to prevent an intruder from breaking into one application, in order to break further into the system.

Guideline 9.1. The applications in a pipeline should be verified as secure.

Guideline 9.2. Applications in use should generate valuable insight from the data they receive, in order to make data analysis easier for the user.

Guideline 9.3. The selection of applications in an IoT pipeline should benefit the company in a cost-efficient manner.

4.7 Summary of the Pipeline

After the literature study is complete the guidelines listed in this chapter is the recommended assumptions and measures one should implement in a pipeline to ensure the efficiency, security and stability of the system.

As many huge vendors already follow these guidelines by default, using trusted vendors might already mean that the IoT pipeline follows these guidelines. The findings may not be exhaustive, but following them should provide greater quality to the pipeline in question. However, these guidelines are general enough to cover the majority of problems that can occur while simultaneously not being application-specific. Following these guidelines should benefit the reader of this paper when creating a pipeline. The guidelines help in the prevention of future problems caused by incomplete architecture or insufficient technical justification.

Chapter 5

IoT Pipeline Architecture

This chapter introduces the architecture of our IoT pipeline. The architecture that scaffolds the IoT pipeline that is used for this project is based on the guidelines presented in Chapter 4. Our pipeline uses emulators and sensors provided by Disruptive Technologies, using the Miles Connect framework as a data load step to connect to Azure Service Bus. From the service bus an application-specific instance of Miles Connect is used to connect to third party applications. The applications are based on three distinct use cases presented in Section 5.5.3.

5.1 Why Separate All Pipeline Steps?

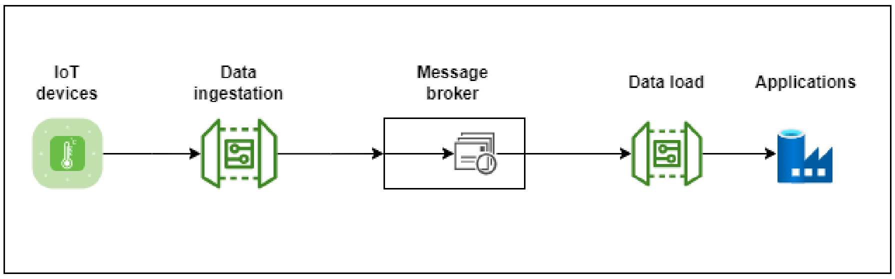


Figure 5.1: Parts of an IoT pipeline

It is already established that all five steps shown in Figure 5.1, must be present in an IoT pipeline for it to function properly, but why have five separate softwares to handle each step? IoT sensors and applications must be present as it is the start and end point of the pipeline. Data ingestation, message broker and data load step, however is not so clear as to why one would want them to operate separately from each other.

5.1.1 Why have a dedicated data ingestion step?

To keep the pipeline simple it is possible to try to skip the data ingestation step, letting the IoT gateway or the sensors go directly to the message broker. This would result in losing validation and formatting that is required to give the message broker the expected format. This means that invalidated and possibly corrupt data is sent on the message broker, taking up unnecessary bandwidth. In addition, the validation process becomes the responsibility of a later part of the pipeline. This would not be good in regards to edge computing principles [56], where computation should happen as close to the production of data as possible. As IoT sensors send data via analog signals a gateway must be implemented to collect data from the sensors, thus, it is not possible to go directly from the IoT sensors to the message brokers. As this step has to exist nevertheless, it can include the recommendations for the data ingestation step, ensuring QoS and that the recommendations are followed. Keeping validation close to production, as well as formatting, to a standardized format also helps limit technical debt, and addresses scaling issues. Any developer working with the pipeline also knows what format is used before they implement anything new, no matter the origin of the data source.

5.1.2 Why have a dedicated message broker?

As the need for a data ingestation step has been established, one can wonder if it is possible to skip the message broker altogether. Using a data ingestation step to simply ingest data into the application layer would greatly simplify the pipeline, as well as removing costs associated with operating a centralized message broker.

Often the problem with data ingestation is that the communication paradigms used are quite basic, usually operating in a 1:1 relationship. This means that only one application would be able to get the data, preventing more complex communication paradigms such as publish-subscribe, as well as creating topics, etc. In addition, the message broker would not be an agnostic central authority which is recommended to prevent data silos. As long as only one application needs the data from the sensors this is not a problem, but the second more applications could make use of the data a message broker would be necessary. This is therefore an interesting conflict between the 'You Aren't Gonna Need It' coding principle [57], and standardizing pipelines inside the organization as well as the general recommendation of a central message broker as discussed in Section 4.4.

To summarize, in order to centralize the message broker and also use the complex communication paradigm publish-subscribe and topic handling, the choice of using a dedicated message broker was made.

5.1.3 Why have a dedicated data load component?

To keep the pipeline simple one could trust the application component to be able to handle data directly from the message broker, or choose an application that accepts the format already established in the data ingestation step.

This breaks the separation of concerns coding principle, where each component should have responsibility of a specific part [58]. Often the application chosen is a third party application for a specific use case, and generally it is not ideal to be reliant on third party applications for logging, validation and formatting. Some applications would manage this great, but as mentioned in Section 4.6, it is not possible to ensure that all applications chosen can manage this. Hence, we choose to have a dedicated data load component that will format and have application-specific validation. For this reason, any application that is deemed necessary, can be used. Even though it only accepts a specialized format, or has poor logging and error handling capabilities.

5.2 Why use Disruptive Technologies Sensors?

Disruptive Technologies is an award winning company famous for the smallest wireless sensors in the world. Their sensors are known for being tiny, robust, affordable, and adaptable. Already in use in offices, hospitals, warehouses and in water quality control, their value-adding capabilities are well documented. In addition, they have quite good emulators as well as documentation, allowing us to develop easily. The emulators operate in the same environment as real IoT sensors, allowing us to easily replace the emulator with their own sensors, should that be applicable.

Their sensors focus mainly on energy efficiency, sustainability, workplace health and well-being, desk occupancy, smart cleaning, and feedback and service. This opens up the possibility to use a wide range of applications. To ensure security in their sensor they hire an external security firm each year to go through the devices, trying to detect security flaws.

Using this cutting edge technology firm as the first part of our pipeline ensures that the IoT devices are well suited for today's business environment, while at the same time following our recommendations for secure devices (Guideline 1.1–1.5). The sensor battery has an expected lifetime of 20+ years, with a 10 year guarantee which means that there won't be any cost associated with changing the battery in the near future (Guideline 3.1).

5.3 Why Use Miles Connect?

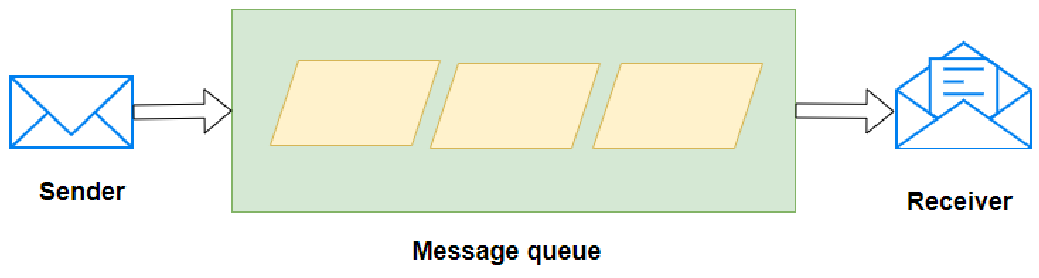
The sensors from Disruptive Technologies operate in a way where a stream of data from IoT sensors end up in DT's servers. From there it can be accessed either via a REST API or a stream. As the data must be inserted into the message broker, an endpoint must be set up, capable of handling incoming messages before inserting it into the message broker. One could set up a stream that would forward any incoming messages from the stream to the message broker. However, then we would have to build our own converting and validation schemes, as well as logging etc.

Miles AS has developed an extensive framework called Miles Connect, which is used to implement services and integrations that run tasks for their customers. These tasks include data retrieval, conversion, validation, and consumption. Miles Connect is built with the two components: *Miles.Connect* and *Miles.ServiceProcess*. The former is used to generate Business Object Documents (BOD), whereas the latter is a framework for scheduling and optimizing integration tasks. Miles.ServiceProcess offers logging, diagnostics, error handling, testing, and life cycle management. In other words, Miles.Connect is the link that Miles.ServiceProcess controls between various services and integrations.

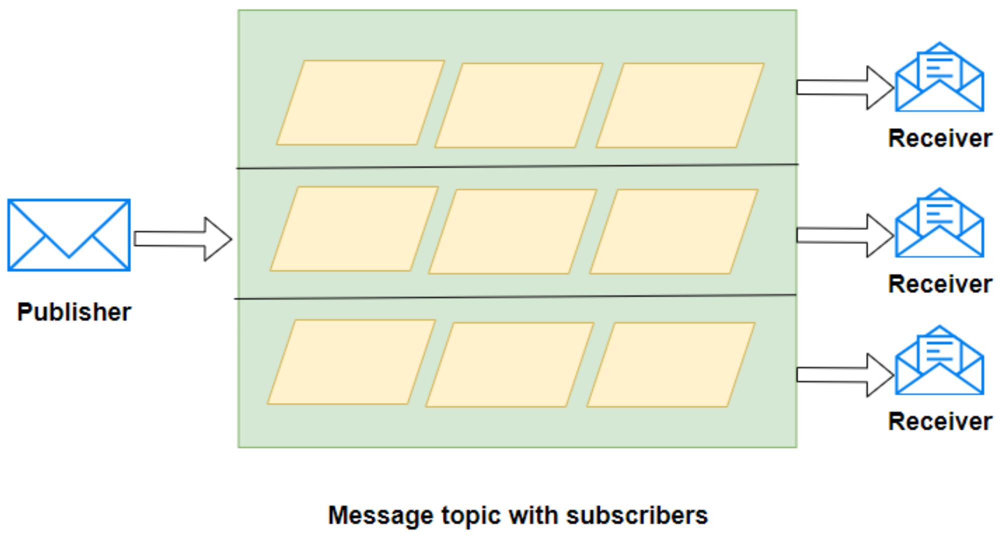
One can use the data retrieval properties to retrieve data from DT and also from the message broker in the data load component. Using the validation and conversion properties to adhere Guidelines 5.1 and 5.2 as well as Guidelines 8.1 and 8.2. The consumption property is used to deliver data to the message broker, in addition to the application in the data load step. As Miles Connect fit our requirements for the data ingestation component, we choose to use this framework for both data ingestation and data load.

5.4 Azure Service Bus as an Enterprise Message Broker

Azure Service Bus is a message broker known for reliability and scalability, while offering message queuing and publish-subscribe communication along with topics. This queuing system is used for sending and receiving messages. The sender will store a message in the queue, and it will stay there until the receiver is ready to accept the incoming message (Guideline 7.2). The messages are safely stored in redundant numbers of storages, and never in volatile memory (Guideline 7.1). The message queue operates in a pull mode, only sending messages when requested [59].



While queuing is often peer-to-peer messaging, the topics operate in a publish-subscribe manner. A message is sent to a certain topic, and all subscribers following such a topic, will receive a copy of the message regardless of their programming language (Guideline 7.3). The messaging is similar to that of message queue, but enables multiple recipients of the same message. With additional filtering and actions, one can limit which messages the subscriber will receive, and see if any extra metadata is attached to a message.



Azure Service Bus offer some additional features that is found compelling and worth mentioning. The auto-forwarding feature enables messaging from a queue as a source to another queue, which enables chaining of queues and topics in the environment. They also have a dead-letter queue which can hold on to any message that is not delivered to a receiver. This queue is also used in messages that cannot be processed. If desirable in a situation, messages can also be delayed for later processing at a specified time. By using a message broker developed by one of the largest distributors of IT service, it is ensured that security measures and protocols are up-to-date for the users' privacy and security [59].

In a research paper about ESB [60], they conduct an empirical survey of all enterprise level service buses and how they compare. When they present the comparison of existing ESBs in the market, we can see that Azure Service Bus and Oracle ESB are the ones that score the best [60].

After a discussion with Miles we chose Azure Service Bus going forward as we have relevant experience working with Azure.

5.5 Applications From Asplan Viak

To make sure our tests were as close to a real life scenario as possible, we reached out to one of Miles' customers, Asplan Viak, to get a use case suitable for them. Our plan being that this would ensure that the pipeline built would be usable in the real business world.

5.5.1 Desired use case

Asplan Viak presented their desired use case as any form of people counting, bicycling counting or car counting as viable use cases. And that this was information they in turn could sell or make use of in a value-adding way.

5.5.2 Technical issues

Given the specific use case, special sensors were needed in order to be able to actually count the people. Reaching out to DT and explaining the use case, they checked if they had any sensors that would work for this use case. Their response being this specific sensor is not a sensor that DT are offering at the moment. As we already were in a process with DT it was decided not to change IoT provider. This however, means that the IoT pipeline's applications could not be based on use cases provided by Asplan Viak.

5.5.3 Use cases in this thesis

Instead of performing the use case presented by Asplan Viak, three fictional use cases are created. These three use cases are created to illustrate different scenarios where IoT devices might be used, and to ensure that the pipeline is able to function in various environments. More use cases could be created and tested upon, but as IoT can be applied to any scenario one wishes, three distinctly different use cases with different requirements are deemed sufficient for testing.

The three use cases chosen for this thesis are chosen as they cover typical scenarios where monitoring via IoT devices is applicable. In addition, these three cover a broad range of environmental characteristics, which is helpful when verifying some of the guidelines.

As our pipeline takes input from Disruptive Technologies' sensors, all three use cases focuses on storage and analytics, with different premises for the sensors and the particular use case. The use cases are:

Storage and Analytics

Our first use case involves IoT sensors monitoring temperature and motion data in a typical office building. A multitude of sensors will send temperature and motion detection data at regular intervals. These data messages will be processed by the pipeline and stored in the cloud. A large number of sensors sending data is expected in this use case. This is the thesis' general use case, and it will serve as the foundation for most guideline validations in Chapter 7.

Basement monitoring

In this use case, sensors for monitoring a basement deep underground are needed. In our fictitious scenario, heavy rain is common, and the risk of flooding is present. To combat this, a couple of water-detecting sensors in the basement are required. This way, if water is detected, one can be notified and initiate counter-measures to prevent flooding.

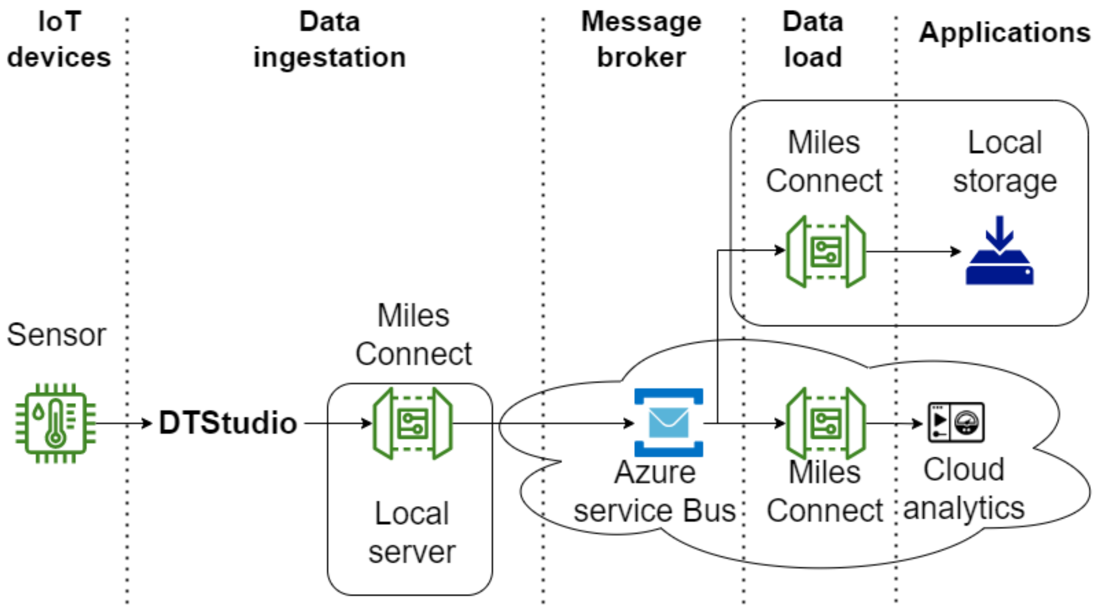
In this use case poor signal quality, a few sensors, and little to none data to be transferred is expected. This use case is referenced primarily when verifying Guideline 2.1 in Section 7.1.5.

Mr. Freezer

In this final use case, the fictional client Mr. Freezer owns a shop. In the shop there is a freezer where he stores heat-sensitive products. To maintain the highest possible quality of his products, the freezer temperature should not surpass 0° Celsius. Mr. Freezer installs temperature sensors in his freezer to monitor this, and keeps track of the temperature using visualization tools on his data. This use case is used when verifying Guideline 9.2 in Section 7.5.2.

Chapter 6

Building the Pipeline



In order to build the pipeline Disruptive Technologies' framework, DT Studio, had to be set up. DT Studio is how we access DT's sensors, and determine where to send the data. Then Miles Connect was used as data ingestation to connect data coming from DT Studio to Azure Service Bus. Before another instance of Miles Connect picks it up and loads it into the relevant application, either in the cloud or locally.

6.1 Connecting to DT Studio

DT Studio has the benefit of using the same interface for both emulated sensors and real ones, thus, one can easily swap out emulators with real sensors with little to no additional programming, should that be required. With access to DT Studio emulated sensors can be created inside their studio, or via a REST API. With their REST API one could easily setup a new device within seconds. Both real and emulated devices have a unique device ID and properties relating to the type of sensor. Creating a temperature sensor, it could report a temperature between -40°C and +85°C, just like a real sensor.

6.1.1 Generating data

Operating with emulators while building the pipeline, it was necessary to create a C# script that on timed intervals sends a request to the emulators in DT Studio. The request contains data the emulated sensors should mock. This is not needed when using real sensors, but a way to make the emulated sensors create data that is expected. This allows us to generate valid and invalid data, and different quantities of data. This flexibility helps when testing the pipeline in Chapter 7.

6.1.2 Accessing sensors

To be able to access these sensors and their data, one needs to authenticate oneself to DT Studio using a service account. A service account is used to create a service account key (username) and secret (password), which is known as basic authentication. DT Studio also supports OAuth which is recommended in a production setting. In OAuth, values are disabled after one hour. Therefore a scheme must be created to update these properties regularly if using OAuth. For testing purposes, basic authentication works fine.

By connecting with our service account details the pipeline fulfills Guideline 1.5: *The access control of the IoT devices and data produced, should be limited to the owners or certified operators*. On the topic of whether or not basic authentication is enough to ensure the security of the pipeline will be discussed in Chapters 7 and 8. After a successful connection, one can manipulate the digital sensor to send and receive events from DT Studio.

6.1.3 Stream and REST API

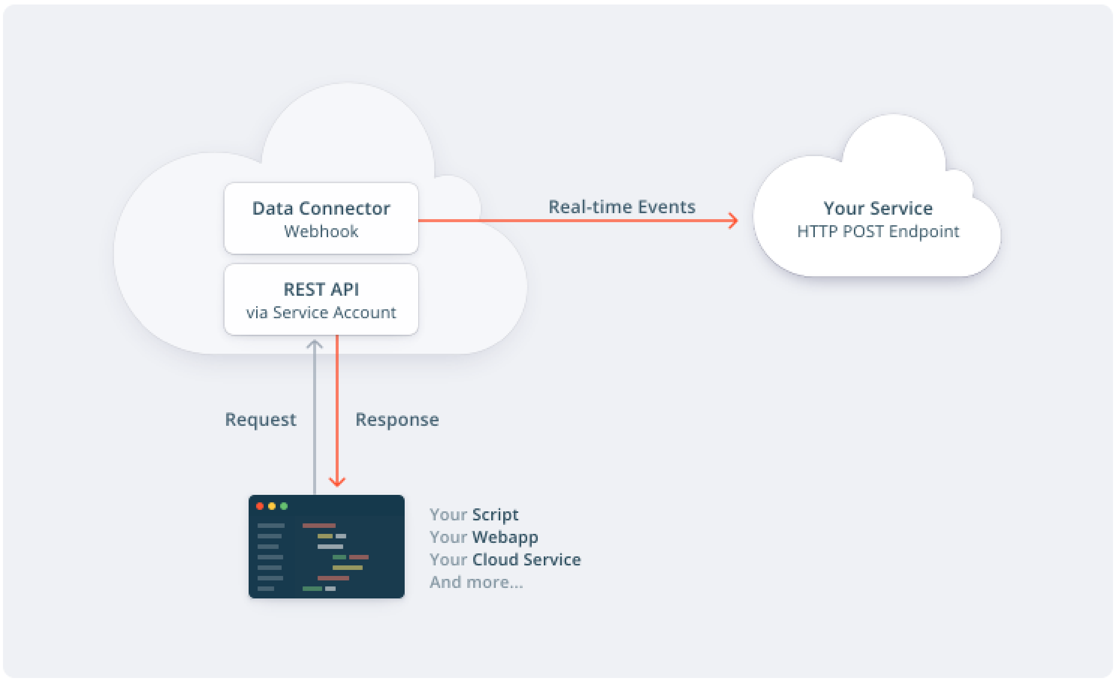


Figure 6.1: Two possible communication paths from DT [61]

The events received depend on how we choose to communicate with DT Studio. Either communicating via their REST API or set up a downwards stream where data ingestation happens at the endpoint, and DT pushes any relevant data onto the stream. Both versions are shown in Figure ??, and were built before deciding what version should be used in the pipeline.

Stream

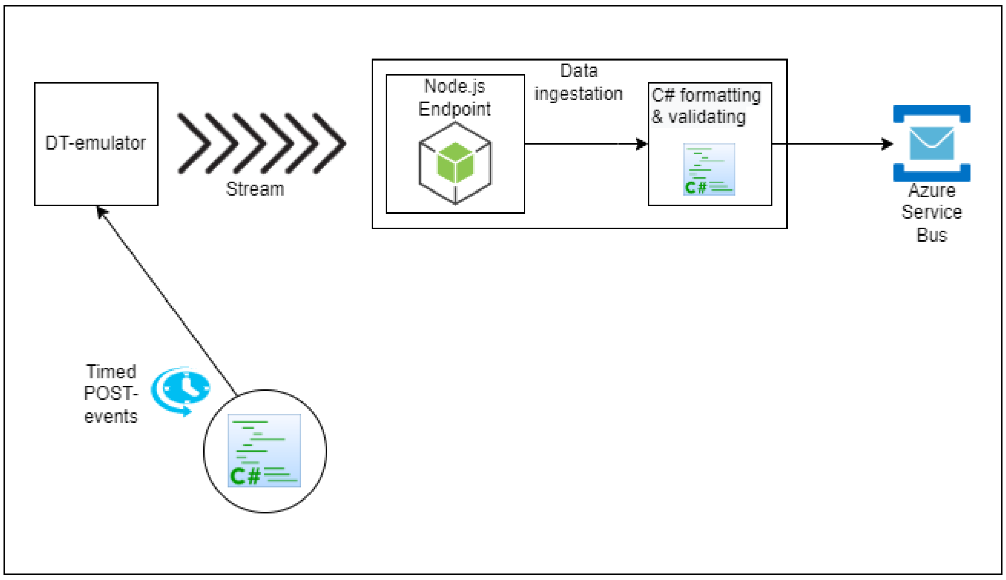


Figure 6.2: Integrate DT to pipeline via stream

To build the stream version, DT's Node.js library was utilized to access a threading package. This package is used to continuously stream events in a separate thread, independent of the main program. With this package customization of the messages are made possible. Every time a new event appears in the stream the data ingestation step can ingest it into the message broker.

In this scenario DT Studio operates as a forwarding mechanism from the IoT Sensors. Any data recorded in DT's Cloud Connectors are sent to DT Cloud where they are forwarded via stream to our endpoint, illustrated in Figure 6.2 as *Node.js Endpoint*. This is quite fast and would also follow Guideline 6.1: *Data transfer should happen with event streaming or similar technology, avoiding ETL. This allows for fast and reliable data transfer.*

Miles Connect technical limitation

After the events have arrived at the Node.js endpoint formatting and validation is needed to take place using Miles Connect. The choice of Miles Connect is discussed in Section 5.3.

Miles Connect operates as an advanced task scheduler, like Hangfire [62], with added logging, validation and formatting capabilities. A task scheduler is based around tasks starting, executing and stopping, and therefore not applicable to a pure event stream pipeline. Thus, one can not stream events in Miles Connect. This means that the pipeline will breach Guideline 6.1 at this step. Making the tasks pick up data from the Node.js endpoint continuously would make this process operate almost like a stream. Thereby allowing for capabilities that are quite similar to an actual stream.

To have a full event stream pipeline not breaching Guideline 6.1 the stream itself would need to be injected directly into the service bus without a task scheduler step in between. We perform tests in Section 7.3.1 to see how this would limit our pipeline.

REST API

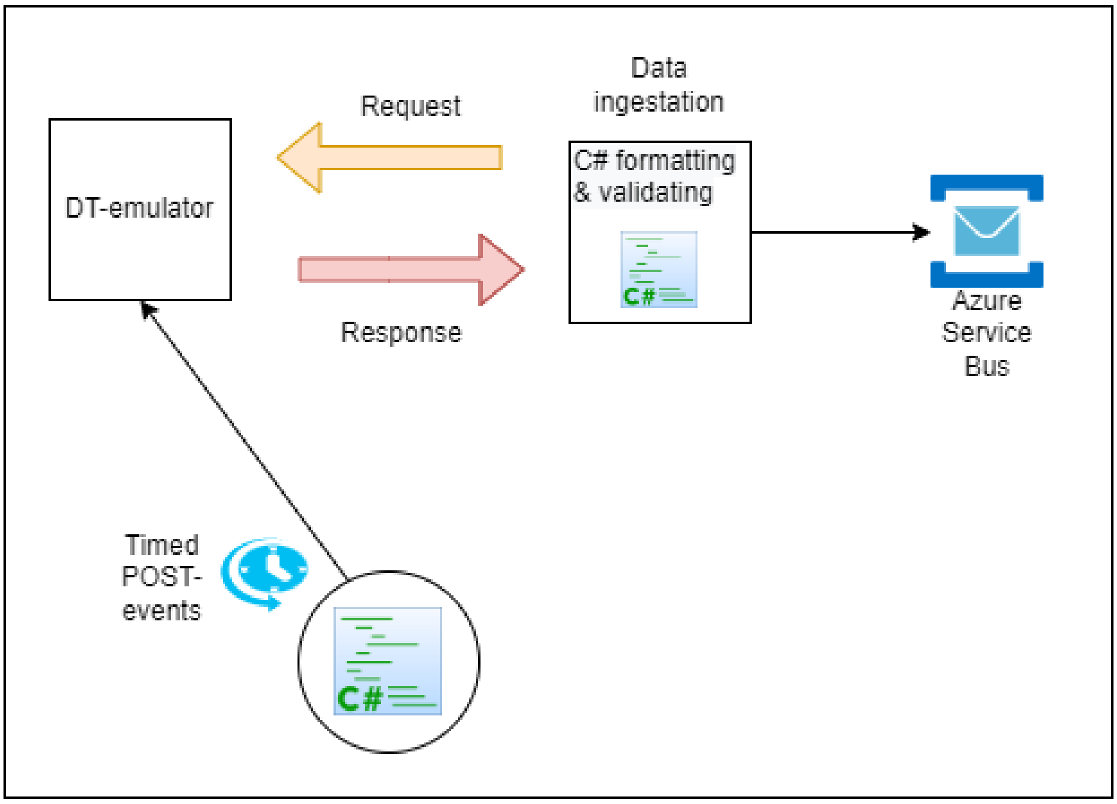


Figure 6.3: Integrate DT to pipeline via REST API

A REST API operates on a request-response basis, illustrated in Figure ??. This means that using a REST API would automatically breach Guideline 6.1. In addition to this a drawback with the REST API is that it takes about 10 seconds longer before any events are available in the API, then when pushed to the stream. The REST API however, grants us larger control of what type of data the ingestation step should receive, in addition to when it should be received. It is also possible to get historical data stored in DT Studio. Thus, the events received and the format it is received on depends highly on what is requested in the request. In addition to control sensors etc. via the REST API, allowing for two-way communication.

Miles Connect is built with regards to REST APIs, gaining synergy from Miles Connect, not needing to set up the extra Node.js endpoint step shown in Figure 6.2. In addition to not being limited by Miles Connect's technical limitations explained in Section 6.1.3 either, as the REST API works well with task schedulers.

After working with and building these two solutions, we trust that the benefits of the REST API outweigh the 10 second delay the pipeline must endure, and choose to move forwards by connecting to DT Studio via the REST API, even though this is a breach of Guideline 6.1. This will, however, be tested in the next chapter to examine how much this breach *costs* the pipeline in terms of performance.

Because the events from a sensor are historically saved within the DT Studio, one can access recorded temperatures over a longer period of time. With an already established connection to a sensor, it is quite easy to access the sensors' event history and make use of the data. For example using the matplotlib package to plot these recordings, as shown in Figure 6.4. This type of plot could be used to make it easier for humans to detect anomalies, or visualize temperature over a period.

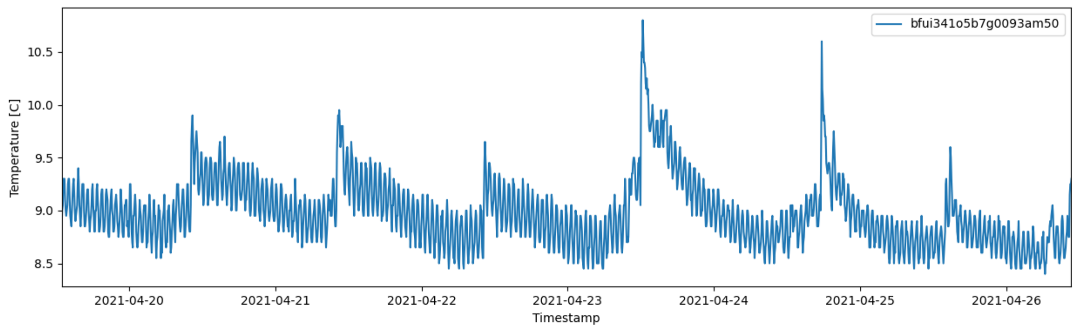


Figure 6.4: Event history sensor. Example image from DT [63].

6.2 Connecting to the Message Broker

To set up the message broker we reached out to Miles and set up a service bus on their license. Azure Service Bus is an enterprise message broker that uses queues and publish-subscribe topics [59]. It decouples applications and services, which allows for load balanced work processes, and can transfer data across applications. Miles Connect is built on the .NET environment, and it is therefore easy to implement service bus by Azure which is supported by .NET.

6.2.1 Miles Connect functionality

Miles Connect has a core functionality where it simplifies receiving data from a *provider*, validating it in a *validator*, formatting it in a *converter* and sending it onward with a *consumer*. In addition to having complete control to configure all of these steps.

Provider The provider communicates with the DT REST API, sending a GET request containing the authentication explained in Section 6.1.2, in addition to an URI explaining what the pipeline wants the REST API to return. In this pipeline a list of all sensors present in our project and their current status is requested. The provider then splits this list up into separate messages, where each message consists of one sensor and its current status. Each message is then sent separately to the validator. The reasoning behind this is that a message containing a temperature sensor might be sent to another application than a message containing a proximity sensor.

Validator When the validator receives a message it validates it before it sends it through to the converter. The validation in question can remove sensors with null values, or otherwise if the message itself is corrupt.

Converter The message format at this point is still the same as it was in the list that was in the response from the REST API. However, it may not be necessary to send all this data further through to the pipeline. The converter converts this format to a predetermined format that is used as a standard on the message broker. One can either add a message ID or remove unwanted parameters. After the conversion is finished the data can be sent through a validator again to make sure that the conversion did not break the message.

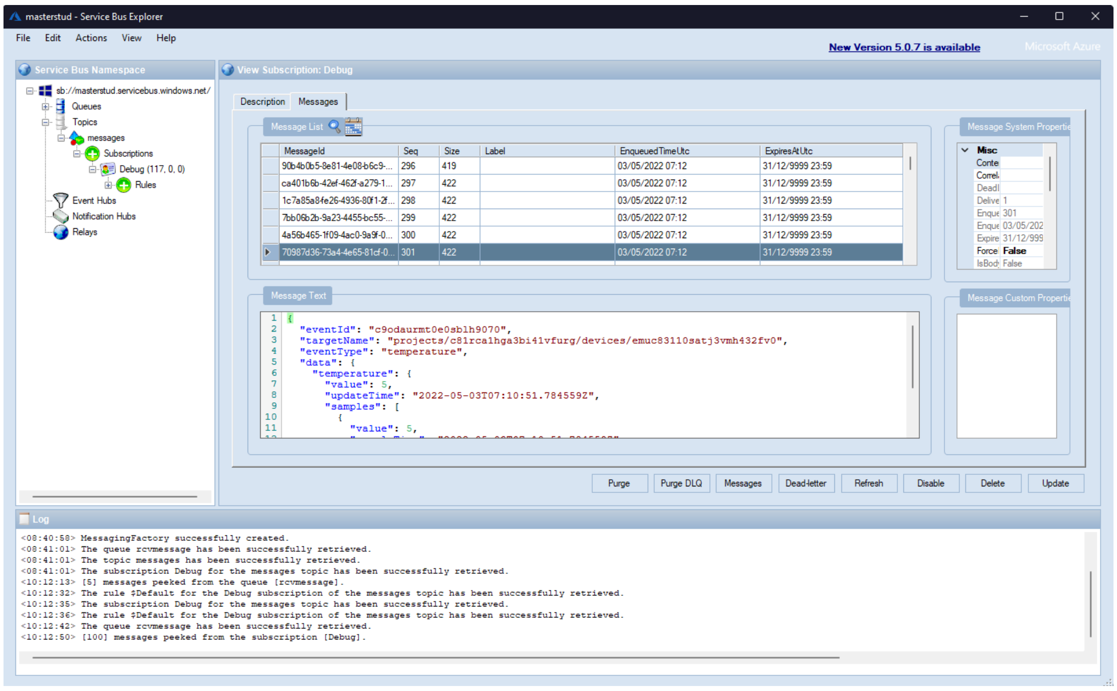
Consumer The consumer is the final step in Miles Connect. This is where the converted message is sent to its new destination. In this case this is the service bus. Connecting the Miles Connect application to the service bus is done by inserting the service bus connection string and topic name into the .NET functionality *ServiceBusMessage*, sending it with the task *ServicebusSender.SendMessageAsync(msg)*. When using the .NET packages created by Azure and Miles, synergy is gained as functions for simpler implementation are enabled.

6.2.2 Azure Service Bus Explorer

In order to control the behavior of this service bus the program *Azure Service Bus Explorer* [64] was used. The service bus explorer lets us administrate messages with ease. The app also contains features for import/export, testing of topics, subscriptions and event hub.

The service bus explorer application is recommended when working with a service bus or event hub as it is open-source, well documented and frequently updated. The tool makes the use of topics, subscriptions and event hubs more intuitive and efficient. It is directly connected to our Azure account, and can generate messages for testing as well.

Figure 6.5 shows what the application looks like.



	MessageId	Seq	Size	Label	EnqueuedTimeUtc	ExpiresAtUtc
	90b4b0b5-8e81-4e08-b6c9-...	296	419		03/05/2022 07:12	31/12/9999 23:59
	ca401b6b-42ef-462f-a279-1...	297	422		03/05/2022 07:12	31/12/9999 23:59
	1c7a85a8-fe26-4936-80f1-2f...	298	422		03/05/2022 07:12	31/12/9999 23:59
	7bb06b2b-9a23-4455-bc55-...	299	422		03/05/2022 07:12	31/12/9999 23:59
	4a56b465-1f09-4ac0-9a9f-0...	300	422		03/05/2022 07:12	31/12/9999 23:59
▶	70987d36-73a4-4e65-81cf-0...	301	422		03/05/2022 07:12	31/12/9999 23:59

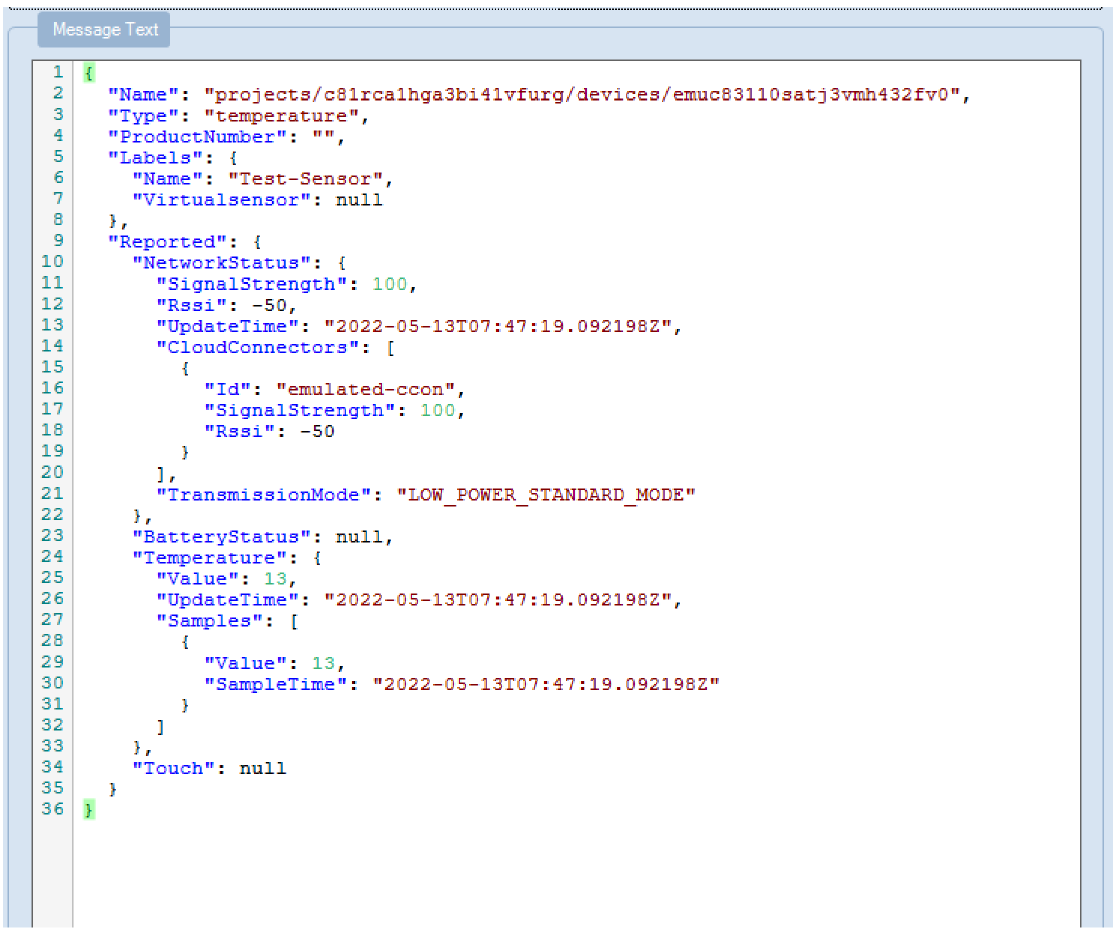
```
{
  "eventId": "c9odaurmt0e0sblh9070",
  "targetName": "projects/c81rca1hga3bi41vfurg/devices/emuc83110satj3vmh432fv0",
  "eventType": "temperature",
  "data": {
    "temperature": {
      "value": 5,
      "updateTime": "2022-05-03T07:10:51.784559Z",
      "samples": [
        {
          "value": 5,
```

Misc	
Conte	
Correl	
DeadI	
Delive	1
Enque	301
Enque	03/05/202
Expire	31/12/999
Force	False
IsBod	False

```
<08:40:58> MessagingFactory successfully created.
<08:41:01> The queue rcvmessage has been successfully retrieved.
<08:41:01> The topic messages has been successfully retrieved.
<08:41:01> The subscription Debug for the messages topic has been successfully retrieved.
<10:12:13> [5] messages peeked from the queue [rcvmessage].
<10:12:32> The rule $Default for the Debug subscription of the messages topic has been successfully retrieved.
<10:12:35> The subscription Debug for the messages topic has been successfully retrieved.
<10:12:36> The rule $Default for the Debug subscription of the messages topic has been successfully retrieved.
<10:12:42> The queue rcvmessage has been successfully retrieved.
<10:12:50> [100] messages peeked from the subscription [Debug].
```

Figure 6.5: Service Bus Explorer app

An example of the data sent is shown in Figure 6.6.


Message Text

```
{
  "Name": "projects/c81rca1hga3bi41vfurg/devices/emuc83110satj3vmh432fv0",
  "Type": "temperature",
  "ProductNumber": "",
  "Labels": {
    "Name": "Test-Sensor",
    "Virtualsensor": null
  },
  "Reported": {
    "NetworkStatus": {
      "SignalStrength": 100,
      "Rssi": -50,
      "UpdateTime": "2022-05-13T07:47:19.092198Z",
      "CloudConnectors": [
        {
          "Id": "emulated-ccon",
          "SignalStrength": 100,
          "Rssi": -50
        }
      ],
      "TransmissionMode": "LOW_POWER_STANDARD_MODE"
    },
    "BatteryStatus": null,
    "Temperature": {
      "Value": 13,
      "UpdateTime": "2022-05-13T07:47:19.092198Z",
      "Samples": [
        {
          "Value": 13,
          "SampleTime": "2022-05-13T07:47:19.092198Z"
        }
      ]
    },
    "Touch": null
  }
}
```

Figure 6.6: Service Bus Explorer message

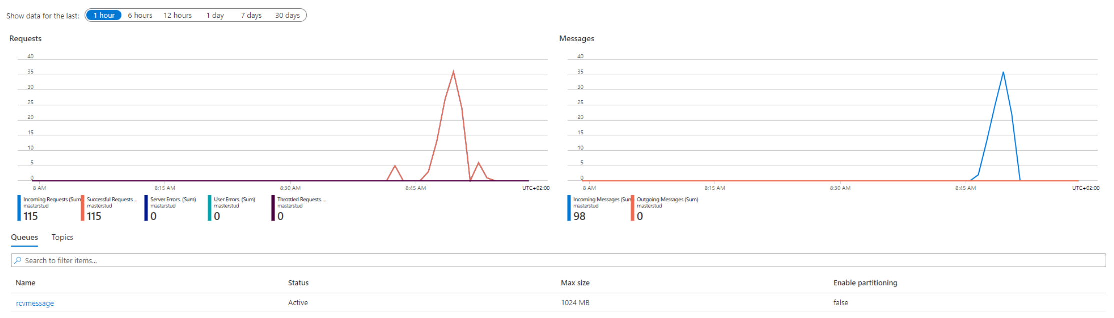
6.3 Constructing the Data Load

When the data is located on the service bus, any subscriber to the topic name can access this data. By using the same packages from Azure and Miles as described in Section 6.2.1, a listener was created with our subscription. This listener is located within the Miles Connect provider. After accessing the data stored on the service bus, one can further format and validate any message if necessary. This process can be performed on a local server if the data is going to be saved for backup by a local application. Otherwise it can run in the cloud on a virtual machine, in order to rapidly load data into cloud applications.

After the validator and converter has done their job, the consumer establishes a connection to our SQL server. This database can be exchanged and replaced with another application if needed. However, for our basic use case it was decided to use a standard database as the client application. The storing of data is completed by creating a SQL transaction, querying the database to store our message.

6.4 Storing Data in the Cloud

The data being stored in an SQL database is proof that the data was successfully traversing every step of the pipeline and arrived at the correct place in a proper manner. As mentioned in the previous section, a private database had to be created, and an additional table within the database for data storage. For our initial setup phase, only a basic version of a database service was needed to ensure that our data was not lost or corrupted during transfer. Figure 6.7 shows messages received during a test run.



Incoming Requests (Sum) masterstud 115
Successful Requests ... masterstud 115
Server Errors. (Sum) masterstud 0
User Errors. (Sum) masterstud 0
Throttled Requests. ... masterstud 0

Incoming Messages (Sum) masterstud 98
Outgoing Messages (Sum) masterstud 0

Name	Status	Max size	Enable partitioning
rcvmessage	Active	1024 MB	false

Figure 6.7: Messages received on the cloud storage

Chapter 7

Verifying the Guidelines

In this chapter, the pipeline built is examined in regards to each guideline mentioned in Chapter 4. The aim is to verify if our pipeline follows these guidelines, performing tests where applicable. Additionally, questions explicitly relevant about DT's sensors are given to DT to see if their devices follow our guidelines. The questionnaire sent to them is located in Appendix B, along with the answers received. The discussion on whether the guidelines themselves are useful is located in Section 8.3.3.

7.1 Guidelines for IoT Device

7.1.1 Guideline 1.1

The IoT devices in use should be created by a trusted source.

Setting up a quality IoT pipeline requires that you can trust the hardware on which the data collection is based around. As seen in Section 4.2.1 this is not always the case.

DT claims to prioritize security and privacy in all parts of the design and development process. Particularly for their chip design, sensor design, radio protocol design, cloud services and APIs.

We reached out to DT to find out how they create and keep their equipment safe and which security measures are present on the sensors. Validation of Guideline 1.1 was done by the following question, answered by a representative of DT.

How is the IoT devices produced by a trusted source?

"In order to ensure that the sensors produced are in fact not tampered with and that the manufacturing process is controlled, DT owns its own manufacturing line in Germany with strict restriction on access."

Owning their own manufacturing line ensures that DT is not dependent on a third party for making their sensors. This results in more control over the production line and better overview over the production as a whole. Having control on what type of machinery is used for manufacturing the sensors and what kind of quality the machines producing the sensors have is also an added benefit.

Any further exploration on the manufacturing of DT's IoT sensors, would go beyond the scope of this thesis.

7.1.2 Guideline 1.2 and 1.3

Guideline 1.2: *Users of IoT devices should be authenticated with modern standards.*

Guideline 1.3: *Common IoT security measures such as encryption should be implemented on the IoT device.*

These guidelines are validated with the following question sent to DT.

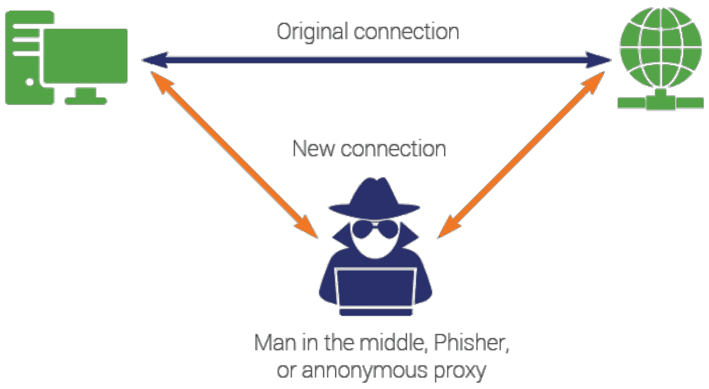
> *How are security measures like encryption and authentication implemented in Disruptive Technologies IoT devices?*

DT's response referred to an article [65] with information around the security of their sensors as well as the SecureDataShot (SDS) protocol.

DT's IoT sensors encrypt all data within the sensors themselves. The data is encrypted while transmitted until it reaches DT's cloud. While using encrypted protocols the data is forwarded to customers applications. This results in end-to-end encryption. Normal encryption ensures only encryption while in transit, but not while located on the device.

SecureDataShot SDS is a high security, low energy protocol. The secure part comes as a result of sensors being paired with the user instead of the gateway. Usually gateways collect encrypted data, decrypt it, perform processing operations on it, and then encrypt it again before sending it onward.

With SDS the gateways, called Cloud Connectors, collect encrypted data from the sensors, but are unable to decrypt them. The Cloud Connector routes packages without seeing what is inside, much like a cellular base station. Because the Cloud Connector forwards data that has been encrypted by the sensor and does not decrypt it, this reduces the risk of a man-in-the-middle (MITM) attack.



A MITM attack is when a perpetrator places itself between a conversation of two parties in order to eavesdrop, alter messages etc. In IoT, a favorable target is the gateway. This is partly because gateways that exist now weren't designed with security in mind, in addition to being in an unsecured environment. This results in many attack vectors against the gateway [66]. In addition, DT claims the following:

"With fewer layers and a single vendor managing end-to-end encryption across sensors, connectors and storage, management is simple and risk is decreased" [67]

As the data is encrypted when SDS is used, the perpetrator can not gain access to the data even if they are able to successfully execute a MITM attack.

Encryption and operational security in DT's sensors At the time of manufacturing, each sensor is assigned a unique 256 bit asymmetric encryption key, generated by FIPS 140-2 Level three hardware security modules.

Asymmetric encryption is an encryption technique where the data is encrypted and decrypted with two separate, but mathematically linked keys, called a public and a private key. The opposite being symmetric encryption where you encrypt and decrypt information using the same key.

FIPS stands for the Federal Information Processing Standard and is a collection of IT standards which are expected to be used in non-government agencies. FIPS 140-2 describes security requirements for cryptographic modules. This standard is divided into four levels where level three is the second highest, and is most commonly used for high security purposes [68].

The public part of these keys are installed in a physically secured facility with access control, in addition to encrypted backups on multiple secure locations in case of data loss. Using these keys the sensor and the cloud can authenticate each other, creating secure sessions. This is how they establish a tamper-proof end-to-end communication channel.

Third-party verification DT has completed two independent security reviews, conducted by a global safety and certification consulting firm called UL. In addition, the company Praetorian has assessed the components DT delivers to be within the top 5–10% of Praetorian's client base in regards to Existing Vulnerability Measure. Existing Vulnerability Measure is used to quantify the collective risk of all findings identified during an assessment, comparing it to the rest of Preatorian's client base [69].

The requirement for encryption and authentication on the IoT devices is verified based on the documentation provided by DT, and additional research on their communication protocol.

7.1.3 Guideline 1.4

To ensure that each IoT device has the latest software which patches security flaws, software used in IoT devices should have the newest version available.

For this guideline, the following question was presented to DT:

> *Is it possible to update Disruptive Technologies' sensors with software updates or security patches after they have been installed at a location? How is this done?*

The response referenced the technology described in Section 7.1.2, DT can provide over-the-air updates to change firmware or configurations. DT operates a fleet management system to facilitate this. These updates are fully automatic and require no involvement from the customer.

Any further investigation beyond this is deemed to be out of scope for our thesis.

7.1.4 Guideline 1.5

The access control of the IoT devices and data produced, should be limited to the owners or certified operators.

Data produced by DT's sensors is owned by the customer of DT. In addition DT's developers and staff is unable to access production data without the customers approval. Any access to the production data is logged, and rules of what they can do is specified in data protection policies [65].

Technical access rights From the technical documentation presented by DT on their website regarding *Managing access rights* they inform how their permission hierarchy operates. As mentioned in Chapter 6, DT practices access to their sensors through service accounts.

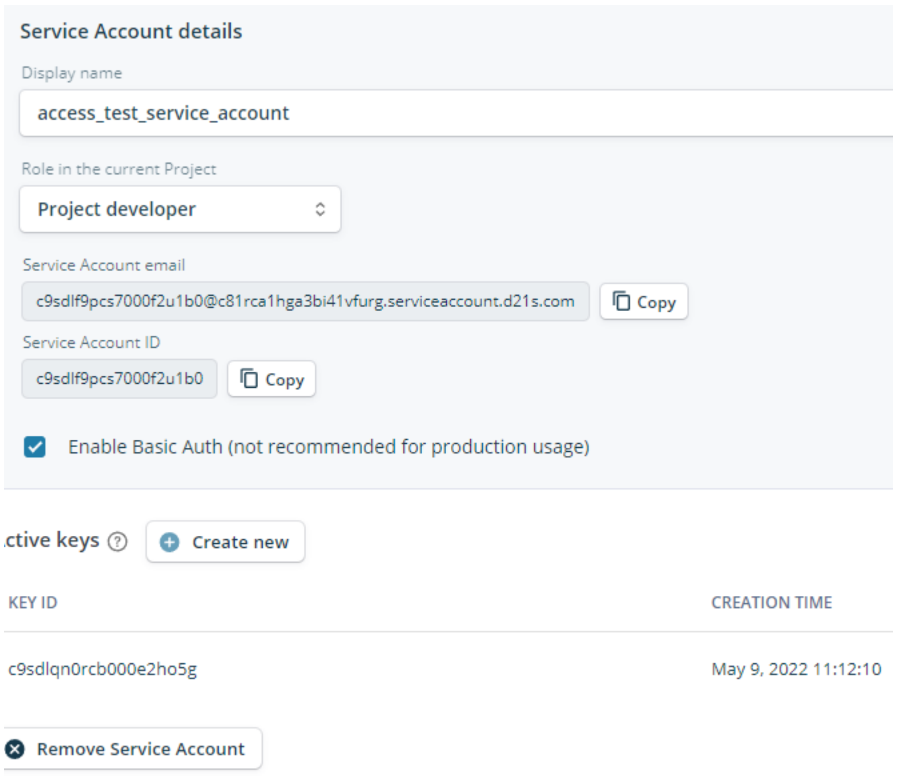
A service account is given a role, which contains certain permissions. A service account can be a member of multiple projects and organizations, and is assigned a role for each project/organization. Even though a service account is created, it needs to be added to a project to be given a role, which accompanies permissions.

A common project user will have access to read data, a developer can also create and update configurations, and administrators can additionally update, create or delete service accounts for a project. On top of that, an organization administrator can handle project management [70]. This allows us to follow the standard security advice of granting least privilege, or granting only the permissions required to perform a task.

To perform a thorough testing of whether or not the pipeline configurations follow Guideline 1.5, we would need assistance from technicians that specialize within the field of cyber security. This is deemed to be beyond the scope of this project. However, to ensure that the basic authentication prevents illicit users from accessing or altering the data, a small test is conducted.

Test: Verify the limitation of access control

To perform this test DT Studio is used, where a project with multiple service accounts was created.



Service Account details

Display name: access_test_service_account

Role in the current Project: Project developer

Service Account email: c9sdlf9pcs7000f2u1b0@c81rca1hga3bi41vfurg.serviceaccount.d21s.com

Service Account ID: c9sdlf9pcs7000f2u1b0

✓ Enable Basic Auth (not recommended for production usage)

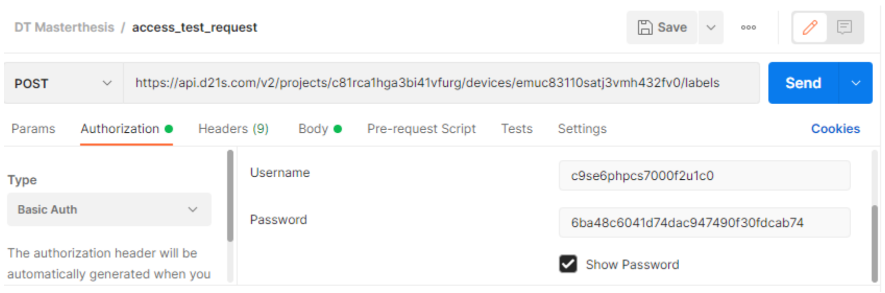
ctive keys

KEY ID	CREATION TIME
c9sdlqn0rcb000e2ho5g	May 9, 2022 11:12:10

Figure 7.1: DT Service account

A new service account called *access_test_service_account* is created and granted *project developer* role, as can be seen in Figure 7.1. This role should allow the user to see and change sensors and settings.

We use Postman to be able to efficiently send requests to the REST API. Postman is an API platform used for building and testing APIs [71]. This chapter utilizes Postman extensively to simplify tests as one can easily mock servers, server responses and REST API calls.



DT Masterthesis / access_test_request

POST https://api.d21s.com/v2/projects/c81rca1hga3bi41vfurg/devices/emuc83110satj3vmh432fv0/labels

Type: Basic Auth

The authorization header will be automatically generated when you

Username: c9se6phpcs7000f2u1c0

Password: 6ba48c6041d74dac947490f30fdcab74

✓ Show Password

Figure 7.2: Postman POST request

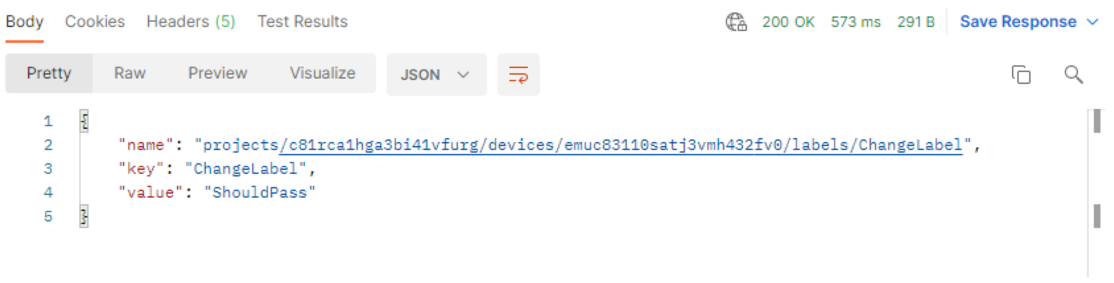
As can be seen in Figure 7.2, a POST request is created. If the request is successful, DT Studio should access the label of the device with ID: *emuc83110satj3vmh432fv0*. Then

DT studio should add a new label which is presented in the POST request. The body of the request is shown in code Listing 7.1 below.

```
{
    "key":"ChangeLabel",
    "value": "ShouldPass"
}
```

Listing 7.1: POST request body

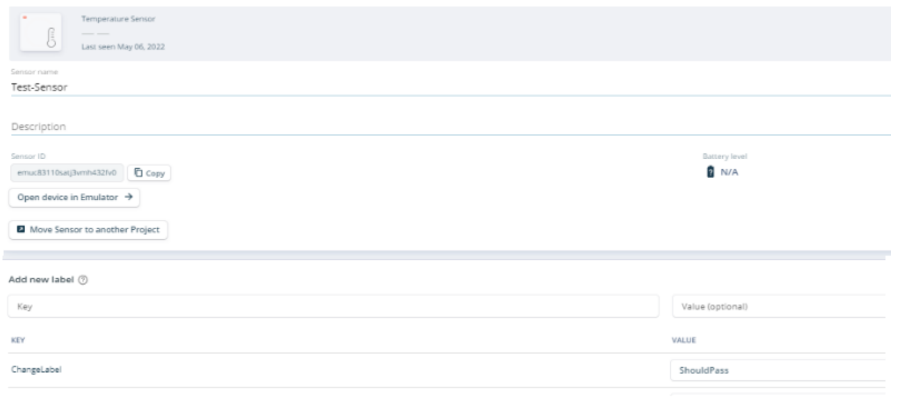
After sending the request a *200 OK* response is received with the values that has been added to the device, as shown in Figure 7.3.



```
{
    "name": "projects/c81rca1hga3bi41vfurg/devices/emuc83110satj3vmh432fv0/labels/ChangeLabel",
    "key": "ChangeLabel",
    "value": "ShouldPass"
}
```

Figure 7.3: Postman Response

This is verified by logging into DT Studio and see that under labels, a new label has been added with the correct key/value pair, as shown in Figure 7.4.



Last seen May 06, 2022

Sensor name
Test-Sensor

Description

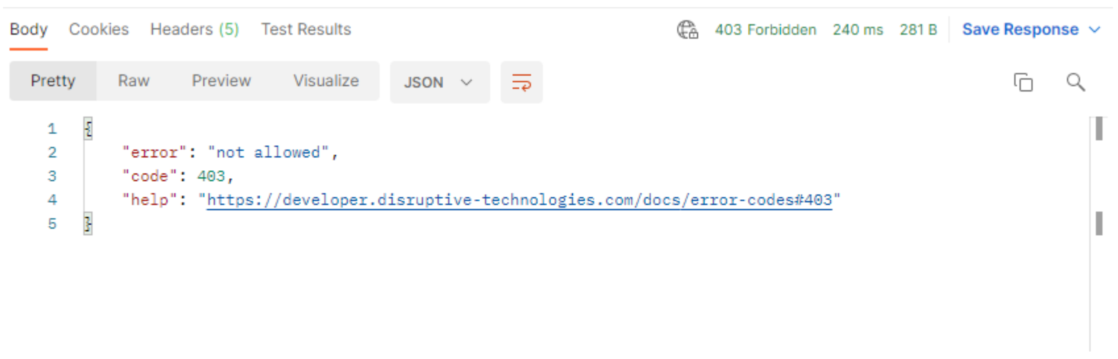
Sensor ID
emuc83110satj3vmh432fv0

Battery level
N/A

KEY	VALUE
ChangeLabel	ShouldPass

Figure 7.4: DT studio screenshot

The privileges are then changed for the service account to *project user*. Afterwards the same response is sent again. A project user should not have the possibility of editing sensors as a project developer has, according to DT's documentation. They should only be allowed to view data.



```
{
    "error": "not allowed",
    "code": 403,
    "help": "https://developer.disruptive-technologies.com/docs/error-codes#403"
}
```

Figure 7.5: DT studio response

Running the same POST request shown in Figure 7.2, returns a *403 Not allowed* as can be seen in Figure 7.5. A *403 Not allowed* response means that this user does not have access to edit the label at this time.

The test user is then removed from the project and the request is sent once more. This time also resulting in a *403 Not allowed.* As any other more advanced way of testing the security of the system is deemed out of scope we conclude our tests for this guideline at this point.

Discussion: Is basic authentication enough of a security measure?

In the DT developer documentation [61] it is recommended to use the OAuth authentication protocol in production settings. OAuth is an industry standard protocol for authorization [72]. However DT does not enforce this recommendation. This means that a developer could use basic authentication while setting up the pipeline. Forgetting to implement OAuth prior to launching the pipeline in production, can leave the pipeline vulnerable. This is because the basic authentication operates as a username-password combination, and any illicit person gaining this combination would easily cause major damage to the pipeline. This would allow a user to be authorized as the service account linked to this username-password combination, and make use of the privilege this service account has.

In Section 7.1.4, the levels of privilege are shown. A malicious actor with project user access can read data, which will breach the confidentiality of the data, as described in Section 2.1. Should a developer service account be breached, this would result in a breach of data integrity as the data could be accessed and altered by the perpetrator. Illicit use of administrator privileges would result in a breach of integrity for the entire pipeline, as the perpetrator would have full leeway to edit or delete anything in the project.

In response to this, OAuth should be mandatory before a data pipeline starting with DT could go operational. We suggest that DT enforce this as a requirement, not a choice.

7.1.5 Guideline 2.1

Ensure that the IoT device and gateway are able to handle poor signal strength.

Transmitting data is an energy-consuming activity, and particularly so for devices with portable and small batteries, which often is the case for IoT sensors. By having poor signal strength the sensors have to spend more time transmitting data, meaning more energy will be spent for that purpose. This can dramatically decrease the lifetime of a small battery [41].

As poor signal strength may affect battery time it is important to choose sensors that have large batteries if operating in poor signal areas. DT's sensor uses Varta CR1216 coin cell for battery power which claims 27 mAh capacity in room temperature. This is roughly 74-111 times less than the average AA battery [41]. DT uses an energy-optimized protocol to limit the transmit time, which in turn limits the power usage.

The range from the sensor to the gateway compares to standard WiFi in normal buildings. Thus, in poor signal strength zones the range and efficiency decreases. WiFi signals, as any radio frequency signal, degrade the further they travel and the more obstacles they have to pass through. Signal from the sensor must travel to the closest gateway, and if that is through walls, floors or around corners the signal strength will suffer. Contrasting this to LoRaWAN, which can have a range of up to 300 meters in deep basements [73], but comes at a cost of low data transfer rate.

Discussion: How well does the IoT sensors fit in the basement use case?

In this scenario, IoT sensors from DT may not be the best match as they are extremely resource constrained, and poor signal strength may remove years off the battery lifetime. Furthermore, the temperature and humidity of the basement may degrade the sensors' performance. The WiFi properties of DT's sensor are effective in normal operations but may be more problematic in areas with poor signal strength zones. As a result, for this use case, where we anticipate poor signal strength we would most likely choose LoRaWAN technology, or another provider for the IoT devices.

7.1.6 Guideline 3.1

Employ a battery-saving scheme that ensures sufficiently long battery life for the devices.

As mentioned in Section 4.2.3, IoT sensors require batteries that should not have to be replaced frequently. As DT claims battery life for at least ten years for their sensors, this question was asked to validate Guideline 3.1:

> *What type of battery-saving scheme does Disruptive Technology sensors use since they have a battery life of 10+ years?*

Their response referred to an article [41] about their sensors' energy consumption.

DT's sensor is powered by coin cell batteries with a 27mAh capacity compared to an AA battery with a capacity of 2000-3000 mAh. This means that the sensor itself has to be extremely energy efficient in order to operate for 10+ years. The average current in the sensor must be less than 175nA to achieve this goal [41]. This average must take into account power for wireless communication, sensing, and anticipated leakage.

As the battery is very small, it is unable to deliver much current. However, the wireless microcontroller needs quite a lot of current in order for it to send data to the gateway. Wireless communication is amongst the most energy expensive operations the sensor must execute.

One of the competitors that have created an energy-efficient wireless microcontroller with competitive numbers, reports a usage of 550nA just to be in standby mode [41]. This leads to a problem, as the average current out of DT's battery needs to be less than 175nA to achieve a battery life of 15 years.

The solution comes from a collection of energy storage capacitors. These can be drained quickly for energy intensive tasks, and can be slowly recharged by the current from the battery. This way the battery can deliver a steady current of 175nA, while the wireless microcontroller can drain the conductors the few milliseconds it needs to be online. This can be seen in Figure 7.6.

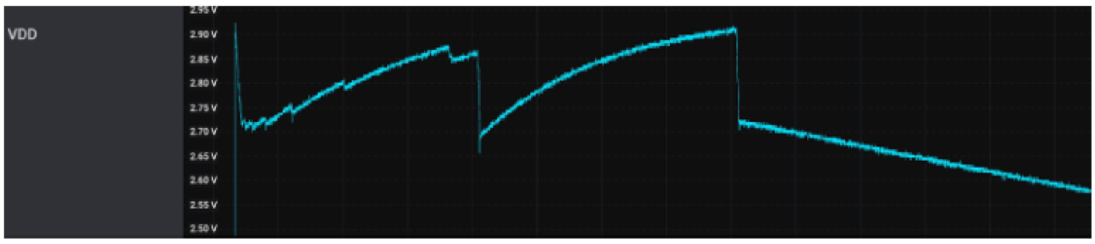


Figure 7.6: Sensor voltage when wireless microcontroller is active [41]

In addition to this, the ASIC *DT Silicon* is used in their sensors. DT Silicon is designed with care towards extremely low power consumption. The ASIC is responsible for keeping the wireless communication completely powered down most of the time to prevent it from draining energy, only powering it up if a sensor detects anything to send, or a timer timeout.

The ASIC's code is written in an assembly language which ensures optimal processing operations, as the language does not have to be compiled further down into machine code, which is the case for other high level programming languages.

SDS

As a wireless microcontroller is a heavy drainer on energy, the amount of time the microcontroller is spending on transmission has a severe impact on the battery. The SDS protocol mentioned in Section 7.1.2 is designed in a way to ensure that the minimum amounts of bits are needed to send a secure message, hence reducing the radio transmission time. A temperature sensor sending an encrypted message to the cloud with 30 temperature samples demands 17 bytes of packet payload [41].

Competing technologies

After reading relevant articles [74–76], which research the possibility of 10 years of battery life for wireless sensors, they deemed it difficult, but achievable. One of the sources claim that their sensor can last up to 10 years if using Bluetooth Low Energy protocol, and having the sensor in sleep mode for 99.86% of the time.

They state that low power consumption while in sleep mode is the key factor to achieve 10 years of battery life for communication devices. Depending on the time it takes the sensor to transmit a message, the interval between measurements, and the average current consumed while in sleep mode, one could aim for 10 years of battery life.

Discussion: How well does this battery-saving scheme fit our use case?

As mentioned DT manages to keep a battery life of minimum 10 years by employing highly energy-efficient batteries and microcontrollers. This alongside their SDS protocol makes sensors communicate at a energy-efficient level.

From the information provided by DT, it can be seen that they do achieve low energy consumption during communication. However, we must compare them against existing competitors to see if their battery-saving scheme is the best suited for our use case.

As stated in Section 7.1.6, an IoT sensor might achieve 10 years of battery life. The fact that DT's sensors achieve a minimum of 10 years in normal operations, verifies this guideline.

Keep in mind, after deciding on a sensor provider, one must consider the transmission strength, duration, and off/on cycle for the power saving mode for the given use case. For our three given use cases presented in Section 5.5.3, we might have different needs for each of them. This would greatly affect the performance of the battery life.

7.1.7 Guideline 3.2

Control that the IoT device load and bandwidth is not exhausted, and scale resources if necessary.

In order to test that the device load and bandwidth is not exhausted, a rogue sensor that will send unnecessary amounts of data through the pipeline is simulated. A temperature sensor from DT will send the temperature it has measured in intervals of 15, or 5.5 minutes, depending on if its a first, or second generation sensor [77].

Therefore a temperature sensor spamming 75 messages each minute would probably be rogue. The reason 75 messages is used is because that is the maximum one can test for in a developer setting in DT Studio.

The fact that this sensor is malfunctioning should not exhaust the bandwidth and device load of the rest of the pipeline.

Test: Rogue sensor

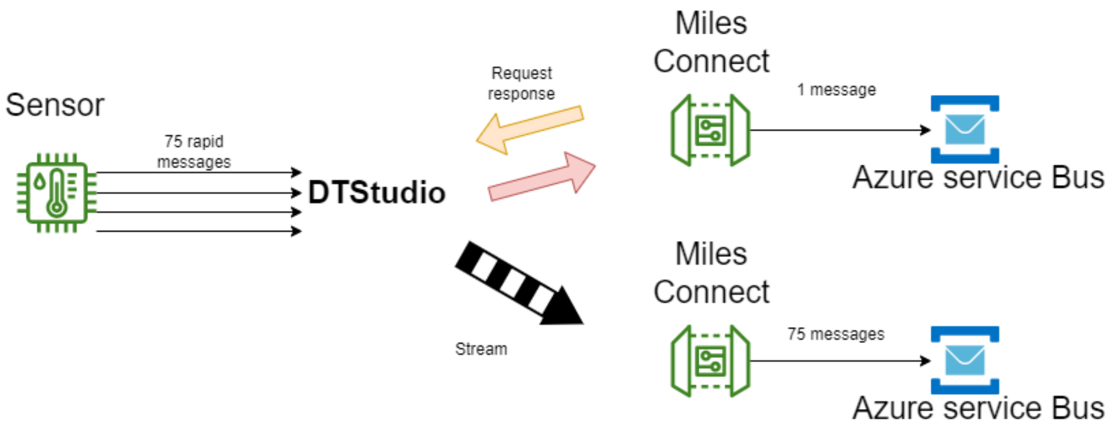
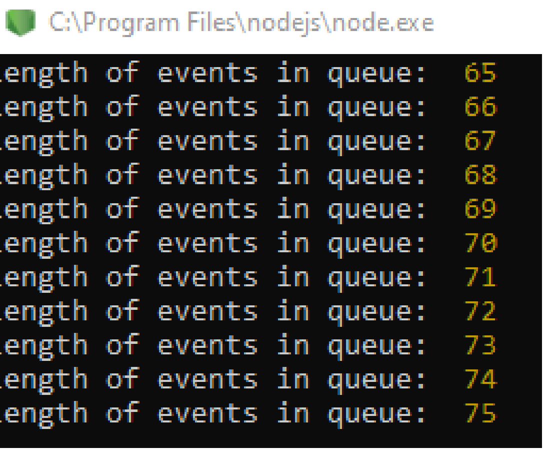


Figure 7.7: Rogue sensor handling, Request-response vs STREAM

We start our request-response pipeline, and set up one alternative stream pipeline as shown in Figure 7.7. Then the emulator in DT Studio is used to mock a rouge sensor that spams data points.

As our pipeline is operating in a request-response fashion, this rogue sensor should not have any noticeable effect on the pipeline. The request will ask for the latest update and gain the latest update from DT Studio. Thus, we expect to get one message on the service bus each minute. In the stream pipeline however, any messages received will be fast forwarded through the pipeline and we expect to get all these messages through the pipeline. This will waste bandwidth and device load.

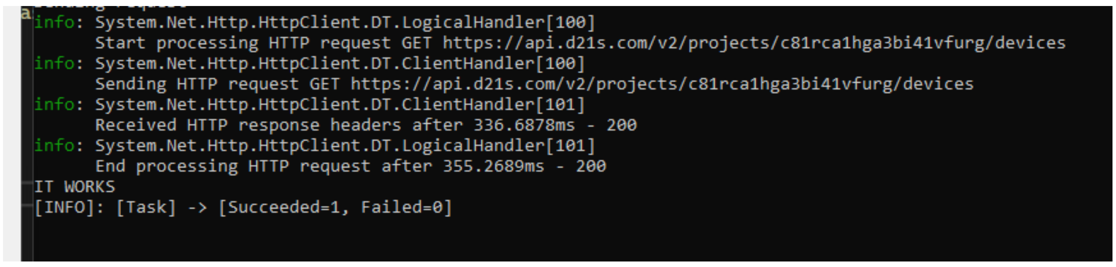
Checking the number of messages through the stream is done by counting all messages that arrive at the stream endpoint.



```
C:\Program Files\nodejs\node.exe
Length of events in queue:  65
Length of events in queue:  66
Length of events in queue:  67
Length of events in queue:  68
Length of events in queue:  69
Length of events in queue:  70
Length of events in queue:  71
Length of events in queue:  72
Length of events in queue:  73
Length of events in queue:  74
Length of events in queue:  75
```

Figure 7.8: Console screenshot, Stream pipeline

As can be seen in Figure 7.8 the total number of messages received at the stream endpoint is 75, meanwhile a single message is passed through the request-response pipeline as shown in Figure 7.9.



```
info: System.Net.Http.HttpClient.DT.LogicalHandler[100]
      Start processing HTTP request GET https://api.d21s.com/v2/projects/c81rca1hga3bi41vfurg/devices
info: System.Net.Http.HttpClient.DT.ClientHandler[100]
      Sending HTTP request GET https://api.d21s.com/v2/projects/c81rca1hga3bi41vfurg/devices
info: System.Net.Http.HttpClient.DT.ClientHandler[101]
      Received HTTP response headers after 336.6878ms - 200
info: System.Net.Http.HttpClient.DT.LogicalHandler[101]
      End processing HTTP request after 355.2689ms - 200
IT WORKS
[INFO]: [Task] -> [Succeeded=1, Failed=0]
```

Figure 7.9: Console screenshot,Request-response pipeline

This verifies that the bandwidth is not exhausted on sending irrelevant messages.

7.2 Guidelines for Data Ingestion

7.2.1 Guideline 4.1

The gateway should scale gracefully with a growing number of devices in the architecture.

To test that the gateway is able to handle this, a working environment where some IoT sensors are already sending data to the pipeline is simulated. By using POST requests in Postman, more sensors can be added to the operation. When the newly added sensors establish connection to the gateway, they will start transferring their own data. When we can observe that all sensors are transmitting data as expected, the newest addition of sensors from the pipeline is removed. Ideally, the pipeline will handle this fine and will increase message throughput while the sensors are online, and decrease it when they are disconnected, without the need for a restart or complete failure. When messages enter the service bus queue, they are logged and can be viewed in Azure Portal.

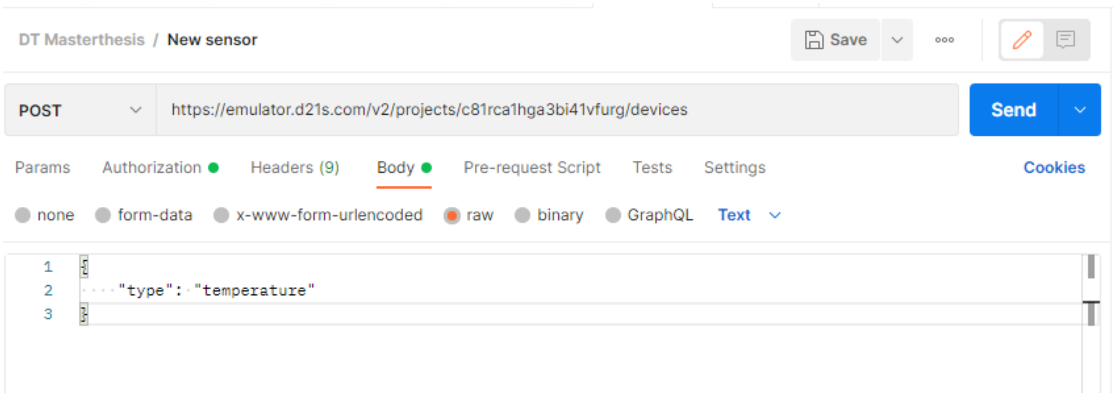
Test: Add and remove sensors while online

```
while (true)
{
    var request = new HttpRequestMessage(HttpMethod.Post, emulatorLink);
    HttpContent content = new StringContent(
    "{\"temperature\": {\"value\": " + integer + "}}");
    request.Content = content;
    System.Threading.Thread.Sleep(5000);
    HttpResponseMessage response = await client.SendAsync(request);
    response.EnsureSuccessStatusCode();
}
```

Listing 7.2: C# script to simulate working environment

The C# script as shown in Listing 7.2 uses a while-loop to command the emulated sensors to send the value every fifth second. This is done by sending a POST message to DT's REST API where the StringContent parameter includes what the sensor should emulate. For this test it is sufficient that it measures a temperature of 0°Celsius every fifth second. In this test, only the number of messages are interesting, not the data each message carries.

After the script has run for a time, the POST request shown in Figure 7.10 is sent from Postman, informing DT Studio that a new emulated temperature sensor should be added.

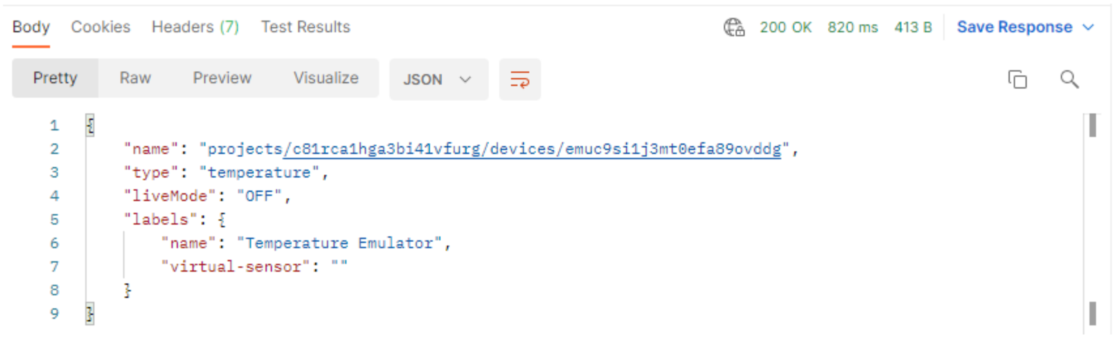


```
https://emulator.d21s.com/v2/projects/c81rca1hga3bi41vfurg/devices
```

```
{
    "type": "temperature"
}
```

Figure 7.10: Create new sensor Request

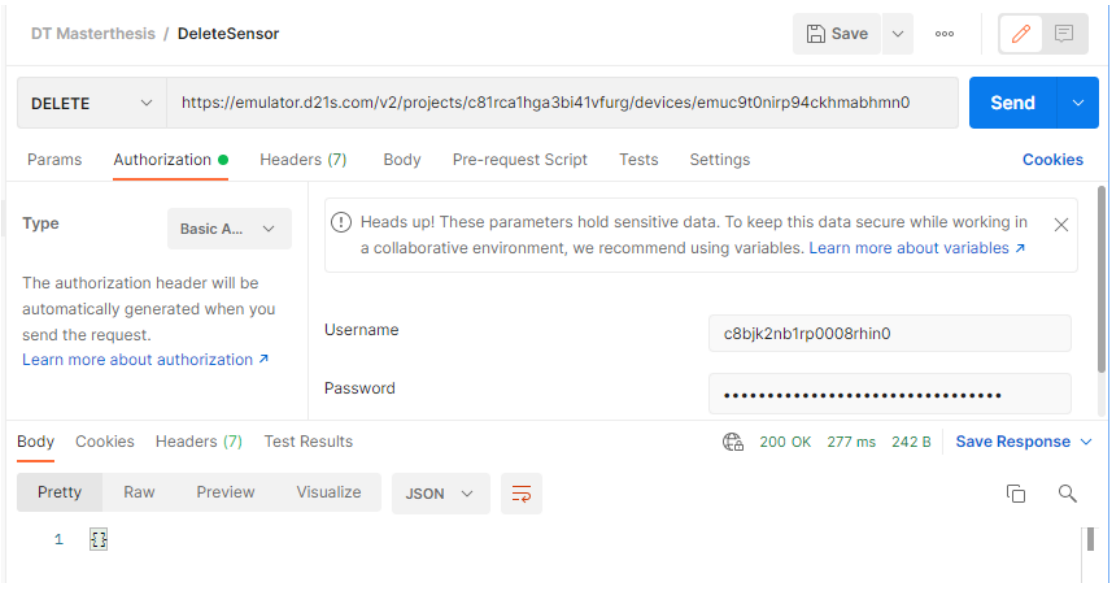
The response from the server is shown in Figure 7.11. We can see that the response received includes the ID of the new sensor, as well as the type and name.



```
{
    "name": "projects/c81rca1hga3bi41vfurg/devices/emuc9si1j3mt0efa89ovddg",
    "type": "temperature",
    "liveMode": "OFF",
    "labels": {
        "name": "Temperature Emulator",
        "virtual-sensor": ""
    }
}
```

Figure 7.11: Create new sensor Request

This sensor will not send any data to the pipeline until an additional script that tells it to emulate data is started. Fortunately, this is just a duplicate of the script in Listing 7.2 with a different ID. As a result, a copy of the script with the new ID was started. After a while it was turned off. Finally, removing this new sensor from the operation. Removing the sensor is completed by sending a DELETE request with the ID of the sensor that is to be deleted, as shown in Figure 7.12.



DT Masterthesis / DeleteSensor

DELETE https://emulator.d21s.com/v2/projects/c81rca1hga3bi41vfurg/devices/emuc9t0nirp94ckhmabhmn0

Params | Authorization ● | Headers (7) | Body | Pre-request Script | Tests | Settings | Cookies

Type: Basic A...

The authorization header will be automatically generated when you send the request.
Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗

Username: c8bjk2nb1rp0008rhin0

Password: ••••••••••••••••••••••••••••••••

Body | Cookies | Headers (7) | Test Results — 200 OK 277 ms 242 B — Save Response

Pretty | Raw | Preview | Visualize | JSON

```
1 {}
```

Figure 7.12: Delete sensor Request

And as can be seen in Figure 7.12, a *200 OK* response was received from the server, informing us that the sensor is deleted.

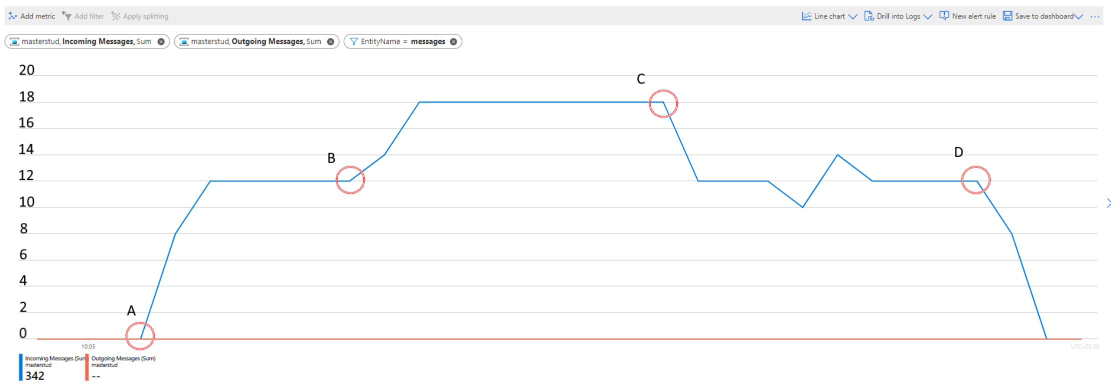


Figure 7.13: Azure Service Bus Result

The test ran over the course of half an hour with a total of 342 messages sent. Each sensor sent their current status every tenth second. In Figure 7.13 the status of incoming messages to the service bus is visualized. After starting two of the sensors at *Point* ***A***, we observe that the number of incoming messages in the service bus stabilized at around 12 messages per minute. A third sensor is introduced in *Point* ***B*** and then one can see that the number increases to 18 messages per minute.

At *Point* ***C*** the third sensor is removed and the number of messages stabilizes back to 12 messages per minute again. After the last two sensors are turned off in *Point* ***D***, there are no more messaging being sent and therefore the service bus now receives zero messages per minute.

Discussion: How well can the gateway accept additional sensors while running?

From Figure 7.13, we did observe that the number of messages increased after adding the third sensor. This is assumed to verify that our gateway is capable of accepting new incoming sensor data.

After the disconnection of the third sensor, there is an unstable moment before reaching *Point **D***. Only 10 messages were received at one minute interval, two of the missing messages being sent at the next given minute, thus, having two additional messages during the next one-minute interval. This is assumed to occur due to Internet connection or software related execution time and thus has no impact on the resulting evaluation.

As the pipeline operates as expected, Guideline 4.1 is deemed verified.

7.2.2 Guideline 4.2

The gateway in the IoT pipeline should be able to connect to other gateways to expand the load of data, working in parallel.

In order to find out how load balancing was done with the gateways, or Cloud Connectors as they are called within DT's system, the following question was presented to the representative of DT:

> *How does Disruptive Technology gateways/data connectors scale if the density number of IoT devices in a small area is higher than the supported capability?*

The representative refer to the fact that each Cloud Connector supports more than 10.000 sensors. And thus, they do not expect the number of devices to be a limiting factor for a long time. In addition, the cloud services have load balancing features that scale with the number of data points to process.

This guideline is verified for up to 10.000 sensors in a small area.

7.2.3 Guideline 4.3

The gateway should be capable of performing edge computing tasks such as pre-processing, cleansing, and filtering.

As mentioned in Section 7.1.6, DT manages to keep a battery life of minimum 10 years by employing highly energy-efficient batteries and microcontrollers. This, alongside their programming code, makes sensors communicate at a energy-efficient level. However, using these sensors in a scenario where we would like to perform edge processing is not practical.

As the design from DT's perspective is to send messages with a minimal package size at given intervals, without any early processing altering the data, any quality assurance of the data would not be possible. The goal of DT is to save battery life, not perform operations on the data, which would be costly in regards to energy consumption.

Also because of the technology used on DT sensors regarding encryption explained in Section 7.1.2, the Cloud Connector is unable to decrypt the data, and thus can not support edge computing. This is a result of the sensor measurements being end-to-end encrypted from the sensor to the cloud, where DT saves their sensor data. This results in the fact that the gateway can not perform edge computing.

The idea behind edge computing is that operations should happen as close as possible to the data source. In our pipeline for this thesis, the operations are performed at the first available location, in the data ingestation step with Miles Connect.

The guideline itself may not be fulfilled, but the pipeline performs the necessary operations at our data ingestation step instead. Which is acceptable for our pipeline which focuses on completing data processing operations on the data before, and after, the message broker step.

7.2.4 Guideline 5.1

The data ingestation step should ensure that the format of the data is correct so that the data transfer will work as expected.

This guideline is validated by sending multiple variations of data formats, where one of them should pass through the pipeline without problem, using the correct format. The other invalid formats should be detected and denied. The pipeline should stop data in an invalid format as early as possible to prevent unnecessary use of the pipeline's resources. The pipeline will forward the data from the IoT devices, but it will stop the data messages at the data ingestation section, if it fails validation.

Inconsistent data Inconsistent data in the system often comes from human errors or missing protocols for data handling. Examples of inconsistent data include similar data in different formats, naming convention mistakes, or absence of data constraints. These errors can lead to inconsistency, which may affect the processing of the data [46]. Before an analysis can be performed, the observed errors require to be corrected. Therefore, it is desirable to ensure the quality of the data at an early stage.

Test: Sending incorrectly formatted messages

In this test there are two sensors that measure temperature, in addition to one sensor that detects motion, and another sensor that detects touch. This is shown in Figure 7.14 The motion and touch sensor will play the part of sending incorrectly formatted messages. The pipeline is configured to only handle incoming data from the temperature sensors, rejecting all other formats of data. The data ingestation step is configured to include a validation process which examines the label *"type"* of the data object.

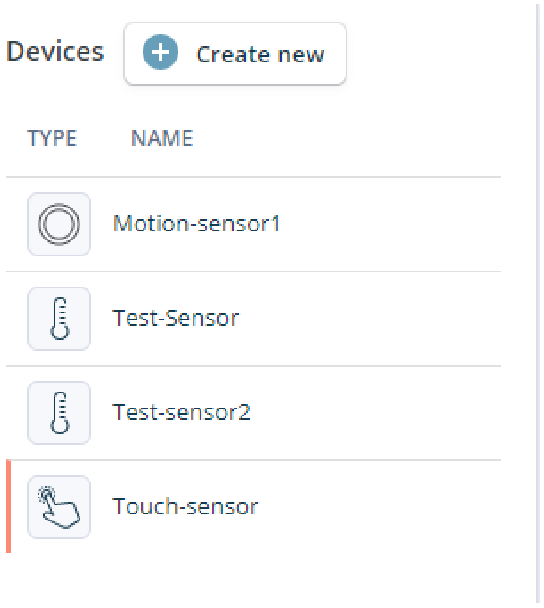
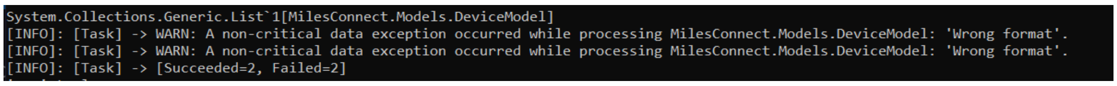


Figure 7.14: Different types of sensors

The test starts by asking each sensor for a status update every 30th second, giving us two responses per minute, per sensor. These responses are set to be handled and routed through the pipeline. To ensure that our resources are only used on the correct messages, a filter is set up to check which type of sensor response is being received. The program should throw a *ValidationException*, which informs the user that the format of the object is incorrect. The program will however still continue to process any incoming data, only rejecting the ones that are incorrect. The errors are displayed in the console, but will not have any affect on the rest of the system.



```
System.Collections.Generic.List`1[MilesConnect.Models.DeviceModel]
[INFO]: [Task] -> WARN: A non-critical data exception occurred while processing MilesConnect.Models.DeviceModel: 'Wrong format'.
[INFO]: [Task] -> WARN: A non-critical data exception occurred while processing MilesConnect.Models.DeviceModel: 'Wrong format'.
[INFO]: [Task] -> [Succeeded=2, Failed=2]
```

Figure 7.15: Expected error in format

In Figure 7.15 one can observe that for each request to DT Studio, four sensor data updates are received, but two of them are failed. This is because two of the four sensors did not pass the format validation test. This can be confirmed with the service bus, where only accepted messages are located. The program will over the test period ask DT Studio for any updates on sensors, and upload valid objects to our service bus. With a frequency of 30 seconds one should expect eight updates per minute, if all messages are valid.

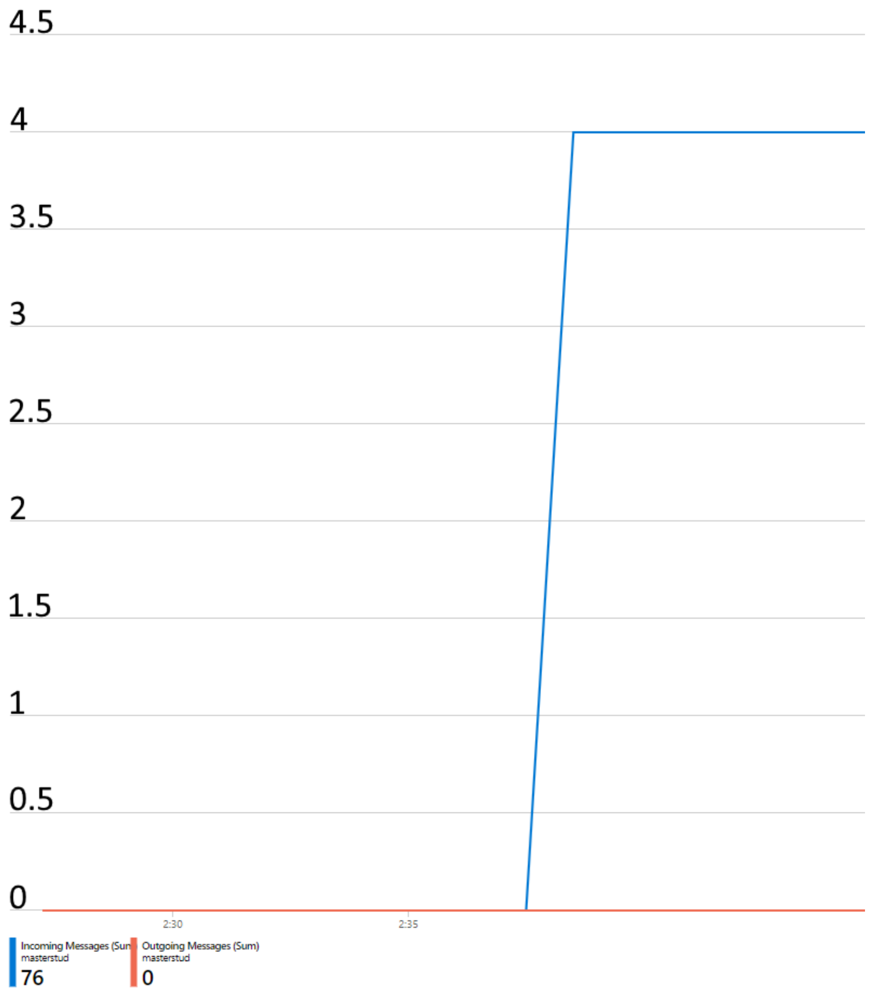


Figure 7.16: Correct formatted messages

However, in Figure 7.16 the incoming messages detected at our service bus are shown. We can see that it stabilizes at four messages per minute, as seen in the y-axis on the left-hand side. This plot tells us that half of the messages sent from DT Studio into the pipeline are not accepted, this because of their invalid format.

These results acts as proof that this guideline is verified.

7.2.5 Guideline 5.2

The data ingestation step should validate that the incoming values are correct and usable, removing any invalid data.

Missing data This guideline concerns any inaccurate or worthless data that might be entering our system. The lack of information within the data create gaps that will affect our final analysis in a bad manner, and should be fixed. These sightings of missing data could be a cause of system downtime or physical errors from the sensors. In the early stage validation it would be sufficient to ensure that the three most important values are present, which is critical for any further processing. These include the device ID, time of recording, and the measured value. If working with temperature values, one would demand that the temperature value exists. If any of these values do not exist, it will cause invalid data to be stored in the database or make a processing function crash at a later point.

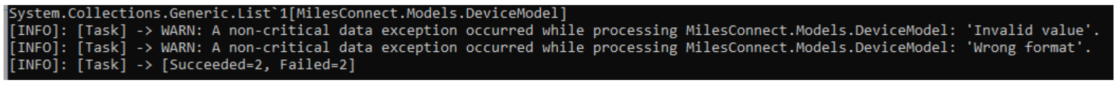
To ensure that our messages do not contain any missing data, the validator is extended to perform operations on the data. The test will deliberately insert invalid data to ensure that the validator can detect these errors.

Test: Sending invalid values

Using the setup from the previous test performed in Section 7.2.4 one of the invalid sensor formats is swapped out to a regular temperature one. However, this time the measured temperature is altered to be incorrect. In this test scenario there will be three temperature sensors, in addition to one touch sensor. It is expected to see one of the sensors fail because of its format, and another one because of its invalid value.

The DT Studio emulator is not built to send invalid values, but rather a range of correct values one might expect in a real-life application. Postman is used instead to create a mock server with an invalid response. By following the documentation of how to set up a mock server, a mock response as well as a unique URI is created. The mock response is created to be identical to the one that would be received from DT Studio, but with invalid values. To achieve the same processing in the pipeline as if using DT Studio, the source URI from DT Studio is replaced with the Postman URI, while all other aspects remain the same.

In the validator a few extra checks has been added that ensure that any message that are further delivered to the pipeline do not contain any 'null' values in the temperature, name, or date variable. If the validator detects any incorrect values, it will throw a similar warning as seen in the testing section for Guideline 7.2.4.



```
System.Collections.Generic.List`1[MilesConnect.Models.DeviceModel]
[INFO]: [Task] -> WARN: A non-critical data exception occurred while processing MilesConnect.Models.DeviceModel: 'Invalid value'.
[INFO]: [Task] -> WARN: A non-critical data exception occurred while processing MilesConnect.Models.DeviceModel: 'Wrong format'.
[INFO]: [Task] -> [Succeeded=2, Failed=2]
```

Figure 7.17: Invalid value

This can be seen in Figure 7.17 where the validator is displaying a warning about one sensor with incorrect value, and another with incorrect format.

Similar to previous validation, the arrival of correct messages is confirmed by looking at the metrics for the connected service bus.

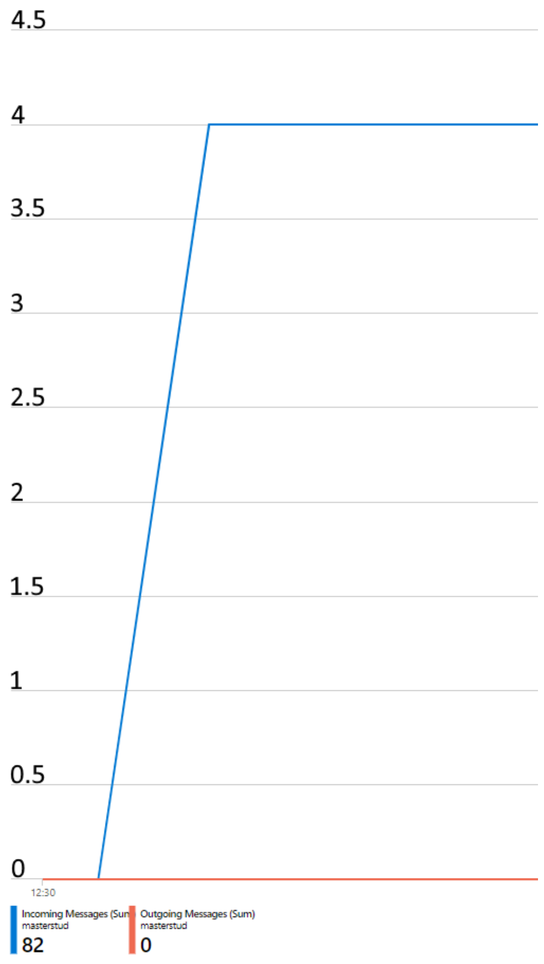
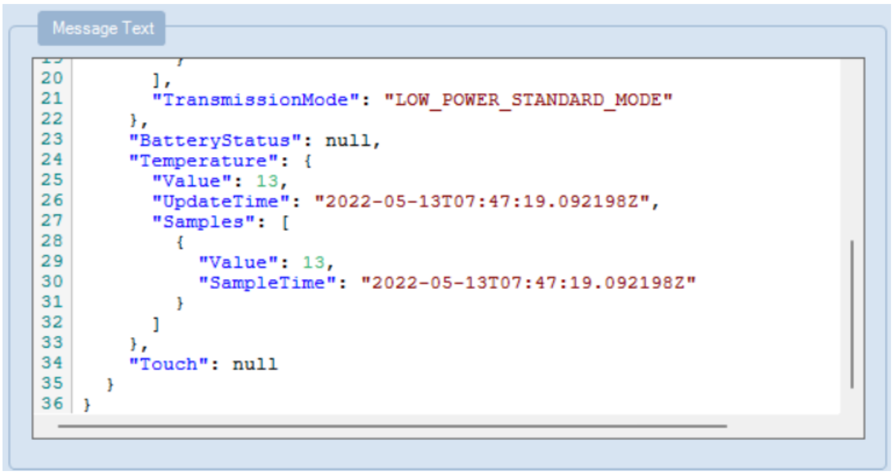


Figure 7.18: Valid value

In Figure 7.18, we see that the number of maximum received values is four per minute, which equals two approved sensors sending one message each, every 30. second.



```
      ],
      "TransmissionMode": "LOW_POWER_STANDARD_MODE"
    },
    "BatteryStatus": null,
    "Temperature": {
      "Value": 13,
      "UpdateTime": "2022-05-13T07:47:19.092198Z",
      "Samples": [
        {
          "Value": 13,
          "SampleTime": "2022-05-13T07:47:19.092198Z"
        }
      ]
    },
    "Touch": null
  }
}
```

Figure 7.19: Approved temperature value

This can also be confirmed by investigating the messages on the service bus to see that they do indeed contain expected correct values, as shown in Figure 7.19.

7.3 Guidelines for Data Transfer

7.3.1 Guideline 6.1

Data transfer should happen with event streaming or similar technology, avoiding ETL. This allows for fast and reliable data transfer.

As addressed in Section 6.1.3, the pipeline is in breach of this guideline as it is using ETL instead of event-driven architecture. This test wants to find out how much this breach costs the pipeline in terms of performance. This is done by maximizing the load to see how well the system can handle huge quantities of data, then comparing it to a test run on Kafka, which is an event-driven architecture.

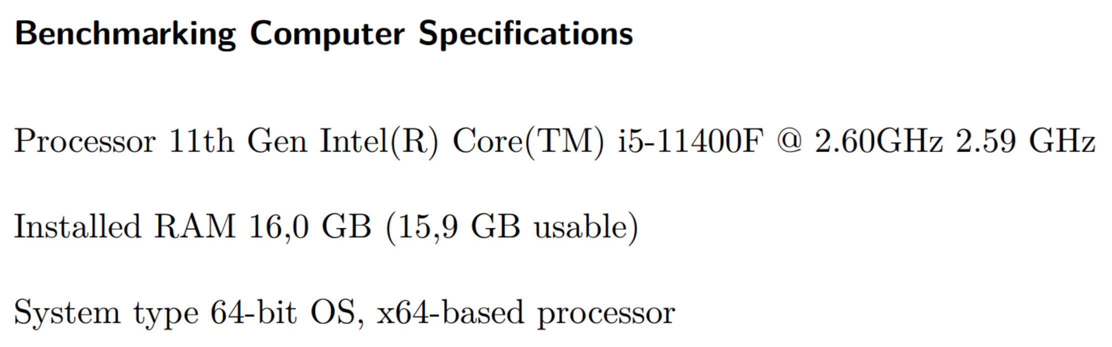
As it is Miles Connect that breaches this guideline, Miles Connect is the focus of the performance test.

There are a few key aspects that are important to be aware of when designing this test.

- **Latency** - Latency between DT's servers, Miles Connect and Azure will have a significant impact on performance.
- **Data size** - The data size of the message which is retrieved may impact performance.
- **Number of messages** - Number of messages collected on each iteration may impact performance.

- **Waiting period** - Miles Connect as a task scheduler have the option to wait a set time before starting a new task. This waiting period may impact performance.
- **Environment** - Running the experiment on slow equipment may impact performance.

To avoid latency from DT Studio being a problem, the DT response is spoofed from a local Postman mock server. That way, the latency is kept to a minimum and the number of messages sent and its data size can be easily altered. This includes the amount of messages Miles Connect will retrieve on each iteration. As Miles Connect is a task scheduler, one must choose how often the iterations should run. By using a time-specific operation command, *cron*, the iteration time is set to the minimum possible with cron, one second. Additionally, the number of sensors Miles Connect should retrieve messages from is altered to see how well Miles Connect scales.



Benchmarking Computer Specifications

Processor 11th Gen Intel(R) Core(TM) i5-11400F @ 2.60GHz 2.59 GHz

Installed RAM 16,0 GB (15,9 GB usable)

System type 64-bit OS, x64-based processor

Figure 7.20: Environment Specifications

The environment this test ran on is a 2021 computer with specifications provided in Figure 7.20.

To test the different configurations between Miles Connect and Kafka, Azure Service Bus was used. Their logging reports the number of messages ingested per minute. Thus the results equals the number of messages sent every minute the test was online. In these test results, startup and shutdown procedure times are not included.

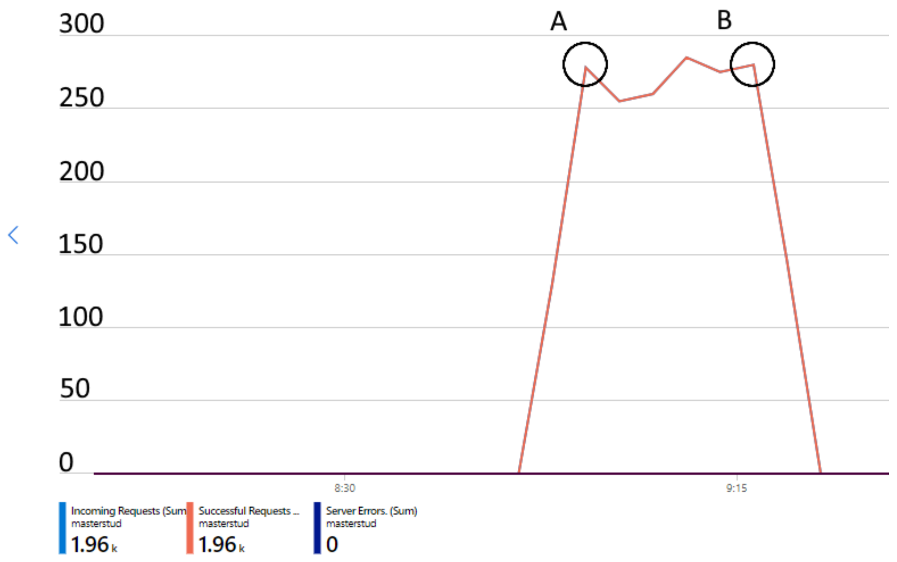
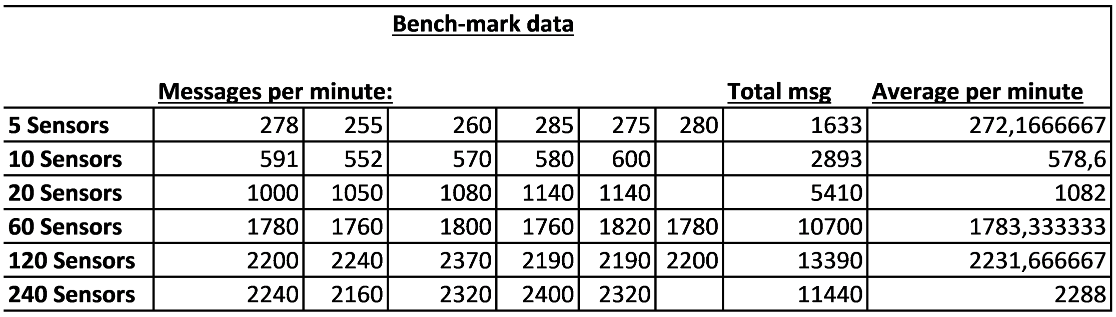


Figure 7.21: One second delay, five sensor messages

Shown in Figure 7.21 is the result of benchmarking Miles Connect with a scheduling period of one second and five sensors connected. The start time for our benchmarking starts at point A, and ends at point B. Throughout this interval the sensors sent 1633 messages. For each of the six minutes the test ran, the system averaged roughly *1633/6 = 272* messages per minute. The total messages collected in the service bus is a bit higher at 1960 messages, which comes from messages sent before and after the benchmarking interval.



Bench-mark data

	Messages per minute:						Total msg	Average per minute
5 Sensors	278	255	260	285	275	280	1633	272,1666667
10 Sensors	591	552	570	580	600		2893	578,6
20 Sensors	1000	1050	1080	1140	1140		5410	1082
60 Sensors	1780	1760	1800	1760	1820	1780	10700	1783,333333
120 Sensors	2200	2240	2370	2190	2190	2200	13390	2231,666667
240 Sensors	2240	2160	2320	2400	2320		11440	2288

Figure 7.22: Bench-mark data

A similar test is executed with a set of 5, 10, 20, 60, 120 and 240 sensors. The results of the tests are shown in Figure 7.22.

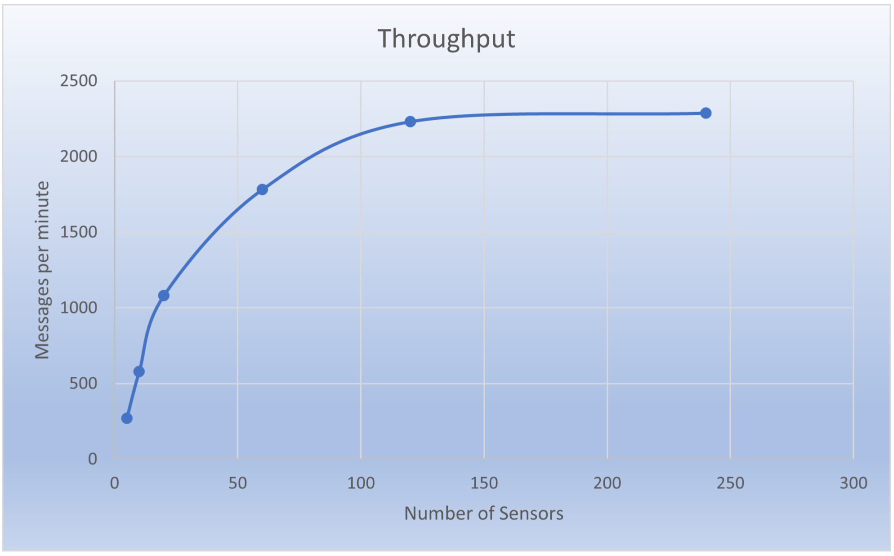


Figure 7.23: Throughput of Miles Connect

The plot displayed in Figure 7.23 shows the throughput of Miles Connect. The figure shows that the amount of messages created and processed, rapidly increase between 5–20 sensors, before it gradually lowers its efficiency and flattens out at 120 sensors.

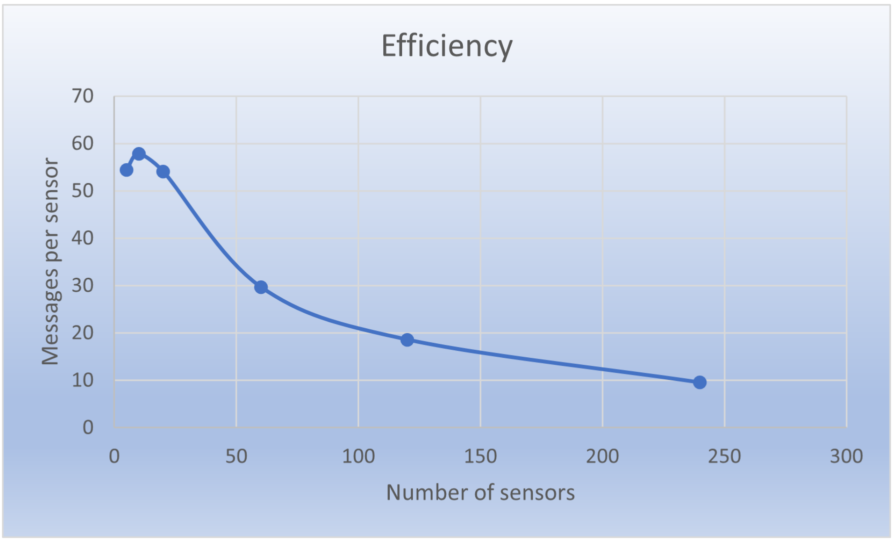


Figure 7.24: Efficiency of Miles Connect

How many messages per minute each sensor on average got through to the service bus, is shown in Figure 7.24.

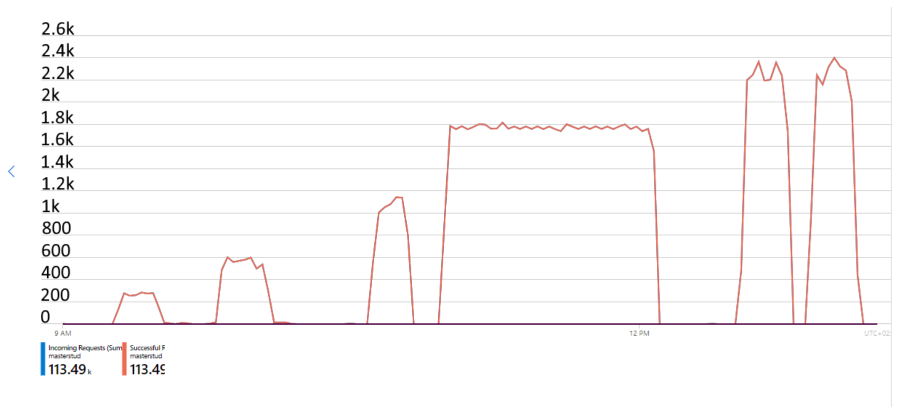


Figure 7.25: All benchmarking tests. From the left: 5, 10, 20, 60, 120 and 240 sensors

Figure 7.25 shows all the benchmarking tests completed, side by side, in chronological order. Combined 113 490 messages were sent and received.

Comparison to Kafka

The authors of the article [31] referenced in Section 3.2.1, use Docker images for easy installation and replication. Docker is an open platform for app development, shipping, and execution [78]. Docker allows one to decouple applications from infrastructure, allowing the developer to deliver software more quickly. This also allows for time synchronization between different servers. They use three identical virtual machines, specifications given in Appendix B.

The Kafka tests are solely focused on the data ingestation step, and are executed on a specialized setup and test environment.

"Apache Kafka producer batches messages to lower the number of requests, thereby increasing throughput" [31]. Threading is used periodically in the insertion, with a batch size of 16KB. Each execution only sends one message to Kafka. These executions have a delay of 10,000 nanoseconds. As 10k nanosecond is 1×10^{-5} seconds, this results in 1×10^{5} operations per second.

When using a *read-in-ram* configuration for their setup, they are able to achieve approximately 220k messages per second.

Discussion: How well did the system perform?

Throughput and efficiency From Figure 7.23 we observed that the throughput increased rapidly when adjusting from 5–20 sensors, also an significant increase from

20 to 60. After increasing to 60 sensors, we notice that the throughput factor does not increase as steadily as it has for the first tests. This is assumed to be a symptom of the system's inability to process more messages at a steadily increasing rate. When further increasing the number of sensors, the throughput converges to a limit at a point between 2000–2500 messages per minute, or between 33–41 messages per second.

Likewise in Figure 7.24, we see that we receive most efficiency when dealing with 5–20 sensors. Even a peak at 10 sensors, as it can handle 60 messages for 10 sensors (600) better than 55 messages for 20 sensors (1100).

It is worth noting that the test was run on a PC with specification provided in Appendix B. Thus with a dedicated server, the tests' upper limit would likely rise.

In a use case, such as our office building one, we might expect a number of 10–100 sensors to send their temperature measurements. As mentioned in Section 7.1.7, DT's sensors send temperature readings every 15 minutes for the first generation sensor and 5.5 minutes for the second generation sensor. Thus, 100 sensors in a office building would send on average $100/15 = 6.6$ messages per minute through the pipeline for the first generation sensor, and $100/5.5 = 18.2$ messages per minute for the second generation. This is well under the maximum throughput.

If one would need more sensors, and assume an average throughput of 2000 messages per minute, which is under the theoretical limit, we can calculate the number of sensors supported. Using only first generation temperature sensors, each sending a message every 15th minute, the total sensors that would be supported is:

$$\begin{aligned} &15 \text{ 1st generation sensors} \times 2000 \text{ messages per minute} \\ &= 30,000 \text{ sensors sending every 15th minutes} \end{aligned} \tag{7.1}$$

$$\begin{aligned} &5.5 \text{ 2nd generation sensors} \times 2000 \text{ messages per minute} \\ &= 11{,}000 \text{ sensors sending every 5.5 minutes} \end{aligned} \tag{7.2}$$

Thus, the ingestation step can handle between 11 000 – 30 000 sensors, at the current settings. Which should be sufficient for most use cases. Increasing the amount of sensors supported can be done by running several data ingestation programs in parallel, or running the data ingestation on a dedicated server.

In comparison to Kafka however, these ingestation rates look rather bleak. In order to efficiently compare these two tests one would have to remove the formatting, validation and converting from Miles Connect, in addition to running it on the same server as Kafka. We believe that Miles Connect still would not reach the same heights as Kafka, but for most relevant IoT use cases we deem Miles Connect to be sufficient.

7.3.2 Guideline 7.1

Message brokers must ensure that the consumers will receive its messages by employing backup strategies. These strategies include external storage for message retrieval, and delivery protocols such as 'at-least-once'

When using Azure Service Bus as a message broker one can access features such as queues and topics. Queues operate in a one-to-one model, where any sent message is stored in the queue until the consumer is ready to accept the incoming message. With this, each queue acts as a distinct broker, storing the message until the recipient is able to retrieve it.

The queue enables a pull delivery mode, where the consumer must ask for a certain message to receive it [59]. The queue also operated with a first-in-first-out principle, so that the messages are delivered in order.

A publish–subscribe model with topic also has the feature of storing messages until the consumer is ready to accept the message. In a topic, each consumer will have their own queue they are subscribed to. The delivery method can be set to either use at-least-once or at-most-once delivery.

If a consumer's application crashes during the process of a message, at-least-once will deliver the message when the consumer is back online. This approach ensures that every message is processed at least once. At-most-once ensures that every message that is successfully processed and delivered, will not be received by the consumer again. This reduces the risk of duplicate data being sent to the consumer [79].

In an event where the message is not successfully delivered to a consumer, the service bus for queues and topics also operates with a dead-letter queue. This queue has an endpoint where the consumers may retrieve their messages at a later time.

Therefore using Azure Service Bus ensures that Guideline 7.1 is verified.

7.3.3 Guideline 7.2

Message brokers should have the ability to perform asynchronous processing.

As stated in Section 6.2, a message broker which utilizes asynchronous processing decouples the applications of providers and consumers, and therefore make the communicating parties independent of each other. In order to verify Guideline 7.2 research is conducted to see if the message broker incorporated in the pipeline has these features.

By using Azure Service Bus and the accompanying queues the possibilities to take use of Azure's asynchronous features is enabled. Messages sent on the service bus use a store-and-forward mechanism, which makes asynchronous processing possible [59].

Using asynchronous messaging opens up for a multitude of communication scenarios such as decoupled workload, load balancing, load leveling, cross-platform integration, reliable messaging and asynchronous workflow [50].

In this thesis the practice of decoupled workload happens when the message broker pushes its messages to a specific queue based on sensor input. The consumer of these messages can access them whenever they like, without any connection directly to the provider. Should sudden bursts of messages from the sender happen, load leveling in the service bus handles this.

As seen in testing of Guideline 6.1 and visualized in Figure 7.25, a large volume of messages over a small period of time was generated. In this scenario the system might be overwhelmed if receiving applications had a long processing time and had to fetch all these messages at once. Instead, the queue serves as a buffer, and the receivers will gradually deplete it at their own tempo. The load leveling characteristic is used for service throttling and prevents resource exhaustion.

As these features are implemented directly into the service bus utilized in this thesis, the verification of Guideline 7.2 is concluded.

7.3.4 Guideline 7.3

Message brokers should allow different applications to communicate regardless of the programming language.

Azure Service Bus decouples applications and services which improve reliability and scalability. As mentioned in Guideline 7.2 using service bus also enables the feature for cross-platform integration. This opens up for the possibility to use programs on different platforms, which may be built using different programming languages and technologies [50].

The technology which enables a cross-platform integration is called Advanced Message Queue Protocol (AMQP), which Azure Service Bus uses. This is a message protocol created by Microsoft and competing vendors and customers. The goal of this protocol is to have an open-standard messaging protocol that is independent of programming language, framework and operating systems. The development of this messaging protocol was motivated by the rapid development of new programming languages and application frameworks appearing in the sector [80].

Message broker vendors discovered that not all existing applications supported features of new platforms being introduced. The introduction of AMQP opened the possibility for using cross-vendor applications, however, it is tricky and often requires integration at application level.

After the development of AMQP it has become an international standard supported by ISO and IEC. It was created with a collaboration of twenty world wide companies within technology and end-user experience. An example which takes advantage of the service bus using AMQP is illustrated below in Figure 7.26.

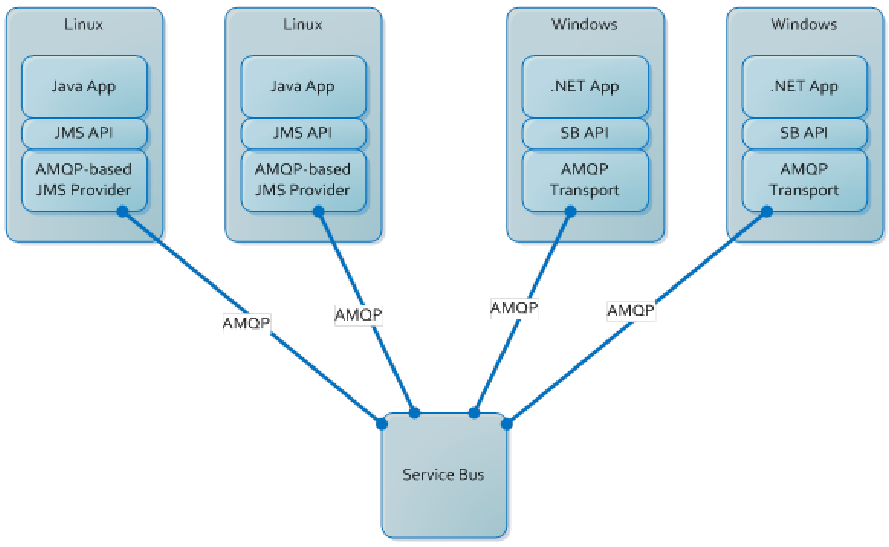


Figure 7.26: AMQP [80]

The image illustrates how an application running with Java language on a Linux operating system can send its message with a connected service bus to a .Net application running on a Windows operating system. The accessibility of AMQP on service buses can be used with programming languages such as Java, C, Python, Go, PHP and Ruby. This ensures that this guideline is verified.

7.4 Guidelines for Data Load

7.4.1 Guideline 8.1

The data load step should perform small data processing validations to increase data quality.

In Section 7.2.5 initial validation was added to remove any values that are invalid to minimize the amount of measurements that do not bring value to our application. In this guideline the pipeline will further extend the cleaning of the data and introduce a new validator. This validator seeks out to ensure that our values are within our range of accepted values. An example of this is to only store values that are between -40°Celsius and +85°Celsius as this is DT's sensor's minimum and maximum. This would remove any unrealistic spikes the sensors detect, known as noise.

Noisy data Any rare data spikes in the data set with meaningless input, are labelled as noisy values [46]. They create an unrealistic view of the data gathered, often created by human mistakes, occur by rare exceptions, or other issues during the harvesting of data. If one would observe the temperature of a living room over a longer period of time, and suddenly detect a few seconds of negative fifty degrees Celsius, before it recovers to its normal temperature level, this would be classified as noisy data.

Test: Remove noisy data

This test uses four sensors where all four have the correct format, and no invalid values. However, two of them will contain values that are noisy. One of the sensors will detect values of 125°Celsius, and another of -100°Celsius, and the final two sensors have expected values. To achieve this, generated data is sent from Postman to the service bus, to fill up the queue of our topic. This procedure is similar to the one seen in the test in Section 7.2.4.

In the next step the data is fetched from the service bus. After fetching the data the new validator is next, which checks for the values within the messages.

The validation check is to see whether or not the temperature values are in the range of accepted values. If a temperature message is not within our range, it will be discarded and no longer processed in our pipeline. The accepted messages will be forwarded and stored in a SQL database service on Azure Portal. To connect to this database, a connection string referencing our database is needed, in addition to writing SQL commands in our C# application.

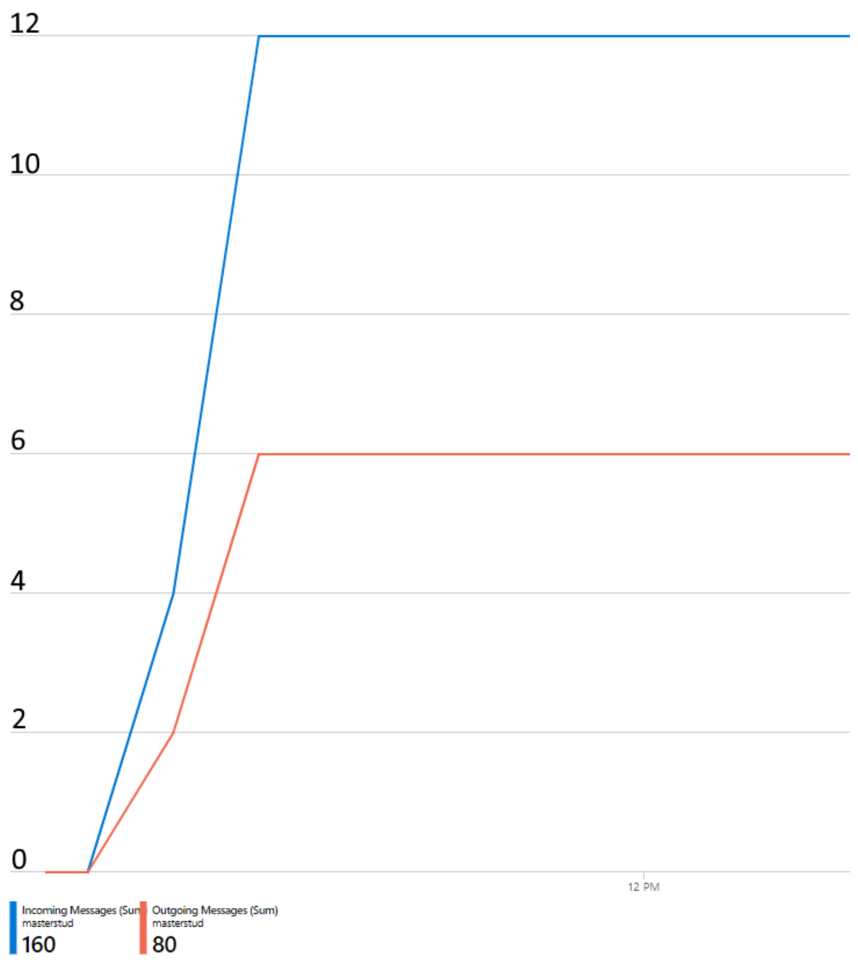
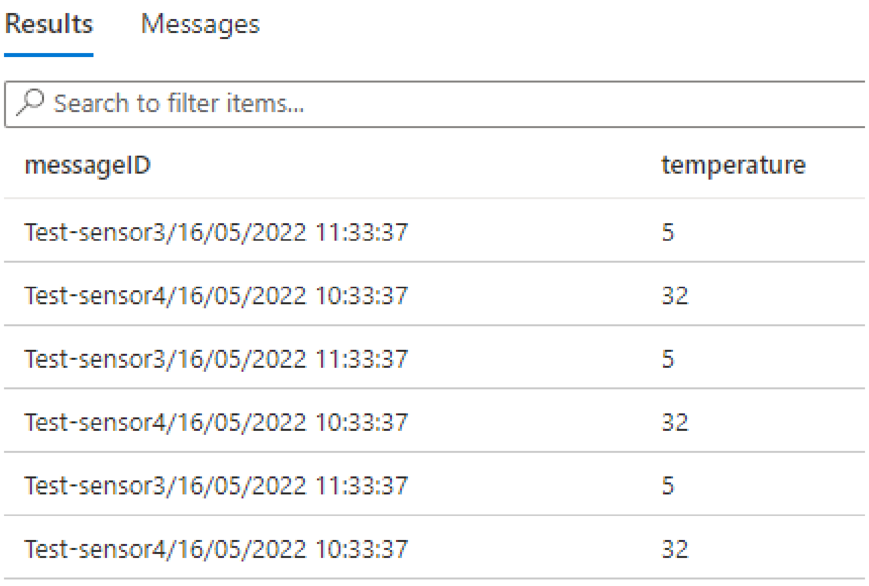


Figure 7.27: Incoming vs Outgoing messages

From Figure 7.27 the incoming (blue) and outgoing (orange) messages from the service bus are displayed. This test scenario are storing four messages every 20. second, which equals 12 messages per minute. In total, 160 messages were sent in this test. Our validator is set to remove two out of four sensors, based on their temperature value. As can be observed from this test the outgoing messages, the ones fetched from service bus, are only reaching six messages per minute. Hence, half of the sensor data messages are not accepted.



messageID	temperature
Test-sensor3/16/05/2022 11:33:37	5
Test-sensor4/16/05/2022 10:33:37	32
Test-sensor3/16/05/2022 11:33:37	5
Test-sensor4/16/05/2022 10:33:37	32
Test-sensor3/16/05/2022 11:33:37	5
Test-sensor4/16/05/2022 10:33:37	32

Figure 7.28: Accepted values in SQL

To confirm that our stored values are within the range, Figure 7.28 shows that data from sensor3 and sensor4 are placed in the SQL database. This confirms the correctness of our validator, and that the pipeline operates as expected, verifying our guideline.

7.4.2 Guideline 8.2

The data load step should conduct application-specific operations before delivery is complete.

To verify Guideline 8.2 application-specific operations on the data received from the message broker is conducted. As discussed in Section 4.5 and visualized in Figure 4.5 unique data load steps for each given application available is performed. This implementation aids the separation of various types of data that are intended for different forms of analysis.

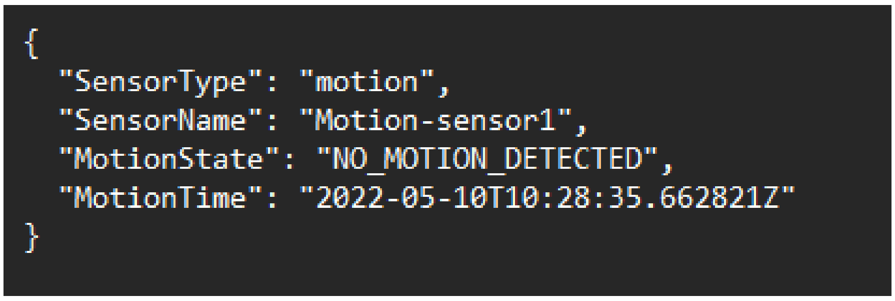
In this scenario a new application service is created in the Azure Portal. This time using a storage account with a data storage container, to distinguish this endpoint from the already existing SQL database. As documented by Azure in their documentation *"Azure Storage offers highly available, massively scalable, durable and secure storage for a variety of data objects in the cloud"* [81]. More on the security of this service application will be discussed in Section 7.5.1.

With this storage account in use, blob type data can be stored, which support streaming scenarios and can be used for big data analytics. The blob storage can also be used for video, images and audio, which makes it a versatile data type. It is also recommended for data backup and recovery.

Test: Application-specific operations

The test starts with configuring the consumer application in Miles Connect to handle incoming data from motion sensors differently than temperature sensors. As the procedure of storing data in the Azure SQL database with C# code is different than for storage account, modifications to the codebase was necessary.

To allow our existing application to insert data into our storage account a unique connection string must be used. For this test a new JSON type object that will contain motion sensor data is created. This can be seen in Figure 7.29.

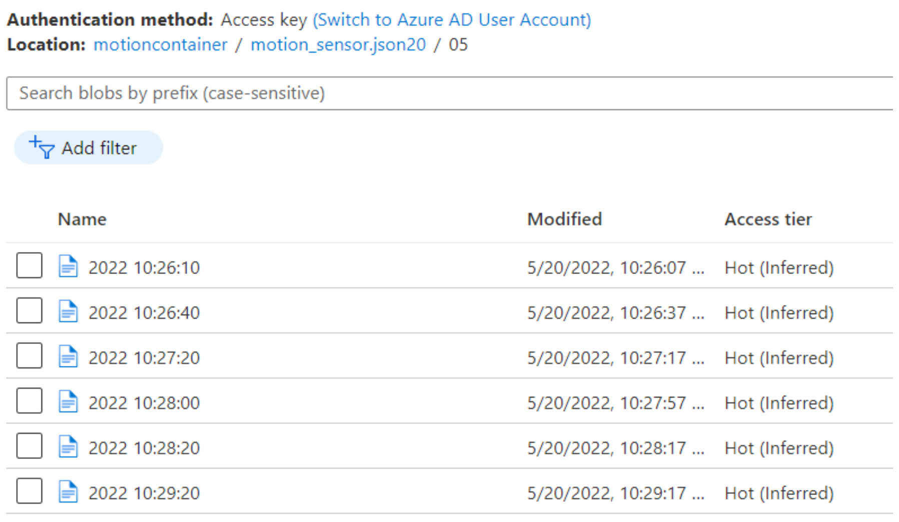


```
{
  "SensorType": "motion",
  "SensorName": "Motion-sensor1",
  "MotionState": "NO_MOTION_DETECTED",
  "MotionTime": "2022-05-10T10:28:35.662821Z"
}
```

Figure 7.29: Motion sensor data

In this scenario two temperature sensors are used, and one motion sensor. Application-specific operations are conducted on the sensors, before storing them in different services. If the data type is motion, it is stored in the container, otherwise it is stored in the SQL database.

By using the log-time as part of the name on the data being stored, a structured log is created in the storage account container. The files in this test are stored in a folder path of month/day, with the year and timestamp as unique filename for each data set, as seen in Figure 7.30.



Authentication method: Access key (Switch to Azure AD User Account)
Location: motioncontainer / motion_sensor.json20 / 05

Search blobs by prefix (case-sensitive)

Add filter

Name	Modified	Access tier
2022 10:26:10	5/20/2022, 10:26:07 ...	Hot (Inferred)
2022 10:26:40	5/20/2022, 10:26:37 ...	Hot (Inferred)
2022 10:27:20	5/20/2022, 10:27:17 ...	Hot (Inferred)
2022 10:28:00	5/20/2022, 10:27:57 ...	Hot (Inferred)
2022 10:28:20	5/20/2022, 10:28:17 ...	Hot (Inferred)
2022 10:29:20	5/20/2022, 10:29:17 ...	Hot (Inferred)

Figure 7.30: Storage account container

Now with this setup one can log the motion data and temperature data separately. Additionally, an extra storage container could operate as a backup for the SQL database. As these application-specific operations worked as expected, this guideline is verified.

7.5 Guidelines for Applications

7.5.1 Guideline 9.1

The applications in a pipeline should be verified as secure.

The pipeline built primarily uses Azure SQL Database as an application step in order to have a functioning pipeline, but Azure Storage Account must also be checked as it was used during testing. This means that to verify Guideline 9.1 research of three topics is required, how secure is Azure SQL Database, Azure Storage Account and how secure is Azure?

Azure security

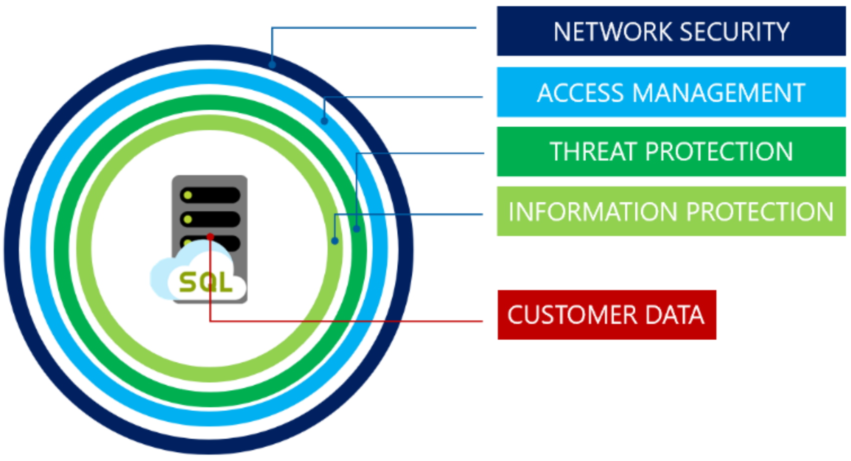
As the applications used run on Azure, the pipeline is dependent on Azure itself being secure. Microsoft Azure used the Secure Development Lifecycle approach when being developed. The principle of this approach being that one should assume a breach.

This means that you should place limited trust in external and internal networks, services and identities. The designer of Azure and Azure's applications have to assume that these are not secure and could be compromised from the start. This means that the design should be prepared for this and in theory should make any future breaches ineffective. Writing security requirements, and guidance for developers on this topic, should be prepared before any code is written. During the development, threat modeling is used to uncover risks in the application, by using a security checklist published by Microsoft. After development, testing with static, random and invalid data is conducted.

Microsoft Azure follows ISO 27001 which is a standard which maintains hundreds of specifications which enables infrastructure to be safe. In addition to FEDRAMP, which is a specification for secure data storage in the US [82].

Azure SQL Database security

As securing an SQL database is a common requirement for many business needs, Azure provides many ways of securing the database to fulfill common security standards.



For network security, firewalls by default prevent access to the database unless explicitly told not to, based on IP-address or virtual traffic origin. For access management, authentication with either SQL authentication or Azure AD is required to prove that they should have access to the database. After the user has been confirmed he can only perform operations on the database based on permissions that user has. This can be either database or database row specific.

Azure SQL database tracks database activities, logging them to an audit log, which can be used to monitor ongoing activities in the database. Or using the logs for historic purposes, locating who did what, at what time. Using Advanced Threat Protection, alerts are issued if SQL-injection, brute force or any other known threatening operation is performed on the database.

Data in transit in the database is encrypted by Transport Layer Security, while data at rest is encrypted with an AES encryption algorithm to prevent unauthorized access to offline backups or discarded hard drives not wiped clean [83].

Azure Storage Account

In the process of validating Guideline 8.2 a storage account on Azure was created, and thus a research on the security of this application is required. According to Microsoft, Azure Storage uses server-side encryption to automatically encrypt the data that enters the cloud. The encryption protects the stored data and meets organizational security and compliance commitments [84]. The data stored within a Azure Storage are encrypted using 256-bit AES encryption, *"one of the strongest block ciphers available"* [84].

The service is also FIPS 140-2 compliant. Microsoft Azure also states that *"Because your data is secured by default, you don't need to modify your code or applications to take advantage of Azure Storage encryption"* [84].

There is also an option to enable encryption at the infrastructure level for those who require a higher level of assurance that their data is secure. This enables double encryption on the data with two separate algorithms, and two separate keys.

Discussion: Application security

In regards to Guideline 9.1 concerning the security of applications in a pipeline, the validation process could be massive, either because of the sheer number of applications, or the amount of detail the security verification each application requires. New applications are launched daily so this is a never-ending job.

In our case, we luckily only had two applications we needed to verify. As it was produced by Azure, a lot of thorough documentation on these applications' own security is published. This however, does not mean that every application which is connected to the pipeline will have the same level of documentation and testing as Azure SQL Database or Azure Storage Account.

A way of dealing with the applications could be to have a pre-approved list of verified applications or vendors. This however, often results in low engagement rates as discussed in Section 4.6. A firm security culture where employees are trusted to make this decision themselves could work for smaller operations, while this would probably result in security breaches in huge enterprise operations.

7.5.2 Guideline 9.2

Applications in use should generate valuable insight from the data they receive, in order to make data analysis easier for the user.

For Guideline 9.2, a scenario where this project could be used in a real use case to provide insight is presented. As we did not have time to set up and perform a test of our system with a real client, a fictional example was created.

Thus, the fictional client named Mr.Freezer, who owns a shop is introduced. In this shop he sells any type of products, and some of them are stored in a freezer room. To ensure that the products are safe to consume, the temperature in the freezer room must not be above 0°Celsius for a longer period of time.

To monitor this, Mr. Freezer would like to use a temperature sensor and store the current temperature every hour. This data should be stored in a safe place, and be visualized in a suitable format for easy detection of abnormalities.

We present our IoT pipeline, and decide to help Mr.Freezer with monitoring the temperature of his freezer room. The pipeline contains technologies discussed throughout this thesis in detail in Chapter 6.

The solutions includes sensors from DT, using Miles Connect to handle the data ingestation and export, Azure Service Bus for data transfer and storing the in Azure SQL database.

After running the sensor and streaming the data points through our system and storing it safely in our database, Mr. Freezer would like to visualize the collected data. When visualizing the data it is important to keep in mind that one would want to keep it simplistic so that Mr.Freezer easily can observe the temperature of his freezer room over a period of time.

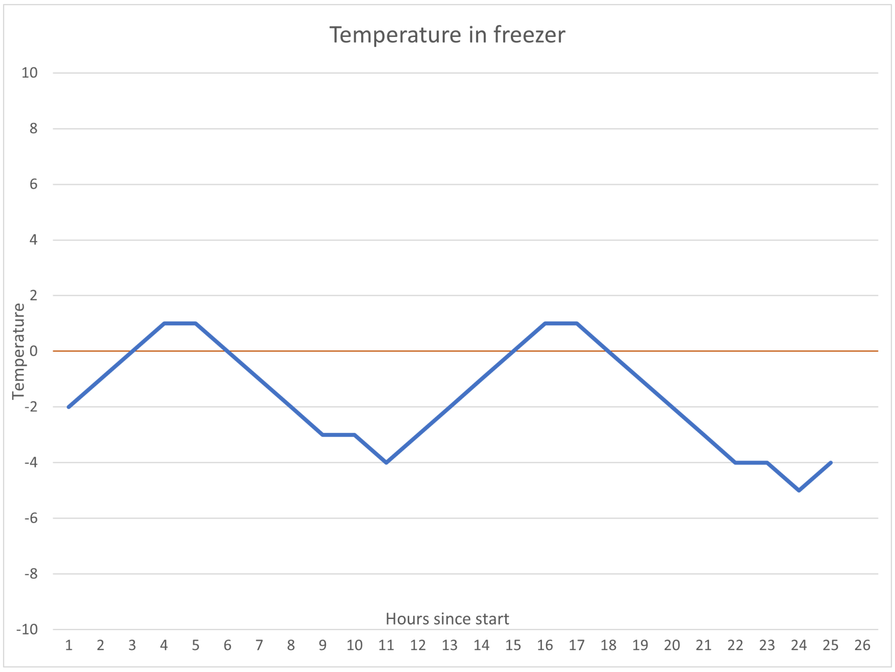


Figure 7.31: Temperature of freezer over time

After 24 hours have passed it can be observed that the temperature is rising and sinking multiple times. And what can be seen from Figure 7.31 is at hours (4) and (16) the temperature is above 0°Celsius, which is above the set danger point. Keeping the temperature at such levels over a longer period of time could damage the products in the freezer room. Hence, Mr.Freezer should quality check the mechanical settings of his freezer room to figure out why such abnormalities occur within 24 hours.

For more critical use cases simply monitoring and visualizing the data would not be sufficient. Connecting to an application where custom alerts can be created should be the other alternative. With certain cloud services one could enable email/phone notification when some values surpass a limit.

For this scenario, the pipeline and visualization of the data is deemed sufficient enough to fulfill the guideline of generating valuable insight and making data analysis easier for the user. The data is safely stored, and the data is visualized in a simple-to-understand manner, which tells the client whether or not the conditions for his products are good.

7.5.3 Guideline 9.3

The selection of applications in an IoT pipeline should benefit the company in a cost-efficient manner.

What is regarded as a cost-efficient manner can vary greatly from what the company is hoping to achieve. For this pipeline, one would want to collect, transmit, store and analyze IoT data. Thus, the relevant costs for each of these steps must be examined.

Disruptive technologies cost

DT operates in a purchase and subscribe manner, where you buy the sensors you need, and subscribe to the access and data you receive from the sensors. The purchase price covers the cost of hardware and one year of subscription. After one year the subscription cost is 140.- NOK for each sensor and 320.- NOK per Cloud Connector. The subscription fee covers updates to the sensors, sensor data reception, database hosting, cloud forwarding, cloud API applications, in addition to customer support.

Azure costs

To calculate the costs of Azure, the pricing calculator from Microsoft Azure was used [85].

Using Azure Service bus at the basic tier would cost 0.05$ per million operations. This would be sufficient for our use case, as a normal DT temperature sensor would send one message each 5-15 minute. If one increases the tier to standard or premium the costs go up to 10$ dollars per month or 677$ per month, respectively.

For the SQL database we have the option of choosing a serverless database or a provisioned one. A serverless is only active when in use and goes into standby mode when not used, while a provisioned one is always online. It is a substantial difference in costs based on the demand needed. Storing 32 GB in a serverless database costs us approximately 7$ per month, while the same storage in a provisioned one costs approximately 1988$. Thus, unless specific demands require a provisioned database, a serverless database would be preferred, and would be the option in our use case.

Based on the services used in the pipeline, and the fact that the minimum tiers of each service was used, a relatively cheap cloud-based solution is created. Increasing the tier of our services could be necessary if a heavier load of messages were to pass through our pipeline, or if it was deemed necessary to use a provisioned database, would prove more costly. In this more expensive case, one might not find the price for the solution cost-efficient.

Additionally, setting up such a pipeline with these services for a person without any knowledge of cloud services and their prices, might make a misjudged decision when choosing applications and tiers, and cost their company a lot of unnecessary expenses.

Discussion: Cost efficiency

Even though all services seem inexpensive at first, it appears to be quite easy to choose expensive options. Even though they may not be necessary. It seems like it is a steep scaling from the basic tiers where one pays very little, to the premium tiers. If however, one manages with the cheap tiers, we believe that this pipeline would be quite cost-efficient. Thus, this guideline proves valuable, assuming the developer following said guideline actually checks the price of the products to be integrated.

Chapter 8

Discussion

In this chapter a general discussion of important topics in the thesis is presented, as well as discussing how the software chosen worked for each dedicated step. The pipeline steps and guidelines are also discussed, to figure out if they indeed are good additions to enhance the quality of a pipeline. Relevant issues and point of views from the discussions in Chapter 7 are included. The discussion chapter finishes by evaluating if the guidelines are a helpful resource when building an IoT data pipeline, in addition to relevant feedback from the industry.

8.1 Discussing the Pipeline

The result of this thesis is a fully functioning, agnostic, and modular IoT data pipeline built based on 22 guidelines. For the three distinct use cases presented; office building, basement, and freezer, the pipeline was deemed fit for two of them. The basement scenario was uncertain due to the fact that other technologies could offer superb distance communication in poor signal areas, which a basement often is. The basement scenario could be solved with DT's sensors as well, just with particular care towards where the sensors and Cloud Connector are placed.

Adding more sensors or more applications to the system, making the pipeline do a multitude of tasks, would require few modifications. Mainly in the data ingestation and data load step. Changing any step to a different provider should prove an easy task due to its modularity.

8.1.1 Are the pipeline steps correctly defined?

When building the separate steps from Section 2.4, we found that most steps adhered to the theory of these five general steps. And thus it was easy to follow guidelines related to each step. There was, however, a slight problem with the second step, data ingestation.

Data ingestation, Miles Connect, and gateways

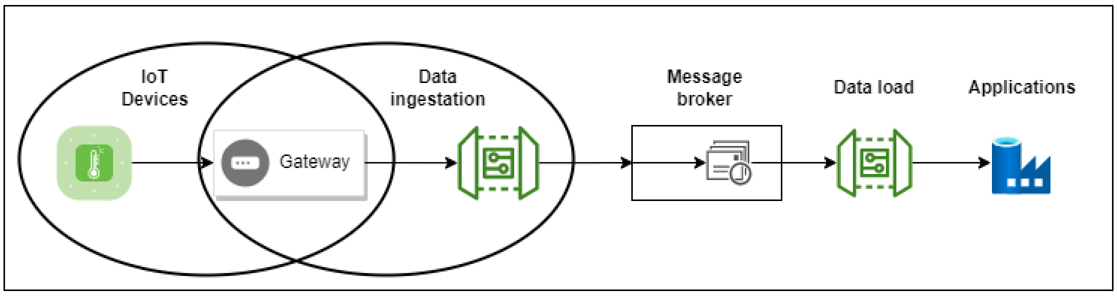


Figure 8.1: Uncertainty of gateway location

The main component in the pipeline that we had problems fitting in with the actual reality, was the gateway in the data ingestation step. As stated in Section 2.4.2, the data ingestation step should retrieve data from the sensors and insert it into the message broker.

In this thesis, the data ingestation step presented in Section 2.4.2 included the gateway. But when building the pipeline, using DT's sensors and Cloud Connectors, this was not the case. Cloud Connectors and gateways in general are physical devices for IoT. Thus, it may be more correct for the gateway to occur in the IoT devices step. This issue is illustrated in Figure 8.1.

In our pipeline, data produced by IoT sensors went through Cloud Connectors to a separate instance of Miles Connect, before being put on the message broker. Thus, the Cloud Connector and Miles Connect are distinctly different from each other as one is a physical gateway located close to the sensors, and the other is a task scheduler hosted locally, or in the cloud. This creates the uncertainty of which step the gateway should be in, to be able to distinctly separate all steps. This is illustrated in Figure 8.1.

Possible Solutions As of now the gateway became partly in the IoT device step and partly in the data ingestation step. It may seem that there should exist a sixth step between the IoT device and the data ingestation step, which would be the gateway. Another solution is to fully incorporate the gateway into the IoT device step.

8.1.2 Is the pipeline actually agnostic?

In the terminology of this thesis, Section 2.1, the term agnostic is defined to include a system that is independent of a specific provider and interoperable among various systems.

In our thesis the message broker is agnostic as it simply transfers the data received by data ingestation to the data load step. It does not perform any computations or modifications on the messages it receives.

The data ingestation and data load steps are not agnostic however, as they are modified to the given IoT provider or application. When the guidelines for data load were introduced, it is stated that a converting step is needed to make the message broker agnostic. The data load step will receive the messages in the same format, but it needs to convert the messages depending on which application needs it. This led to Guideline 8.2, *The data load step should conduct application-specific operations before delivery is complete.*

The same principle could be applied for data ingestation, as different types of IoT vendors or sensors will operate with different types of data formats. Therefore, one can expect to modify the data ingestation step depending on the IoT vendor, converting the data to the format data loaders expect coming out of the message broker.

In our pipeline data ingestation and data load is incorporated to remove any invalid values, messages, and formats depending on the use case. This will lead to increased data quality, but at the cost of not having a fully agnostic pipeline. As we have focused and practiced a modular approach for our steps, the agnostic approach might not be possible. This, however, would depend on the use of the term agnostic. If one would assume that for the entire pipeline to be agnostic, all steps in said pipeline must be agnostic, a pipeline for these use cases might not be achievable at all. This is due to the fact that at one point, one part of the system has to know what it is receiving and what format the receiving data is on.

8.1.3 Is the pipeline too complicated?

Arguments can be made that our pipeline for our use cases is too complicated and unnecessarily complex. This comes down to the modular, agnostic, performance, and scaling features introduced.

For a simple use case where a developer wants to check the temperature inside the building they are working from, this may very well be the case. It is reasonable to assume there would be simpler methods of gaining access to the data and examining it.

In a scenario where someone is running a huge operation with many IoT sensor providers, and many applications on the end of the pipeline, the features presented are critical for successful implementation and growth of the pipeline. These features are extremely important to prevent a complete redesign and rebuilding of the pipeline. In such a use case, one is dependent on easy integration of multiple unknown parameters, formats and applications. This is ensured by the modular and agnostic traits in this pipeline. If the project gains traction and scales from one office building to a hundred, the scaling trait becomes important. And if many different applications and use cases should make effective use of the data, efficiency is key.

Thus, for building a robust pipeline that can meet future needs, the pipeline is just as complicated as need be to ensure robustness.

8.1.4 General Security in the Pipeline

The guidelines in the thesis focuses on security in the IoT devices and applications. IoT sensors and the gateway is important to secure as they are physically located in an unsafe environment, like an office building. The applications are also important to secure as the sheer number of applications means that some of them might have security flaws. Two guidelines were therefore created, one for IoT devices, and another for the applications.

There exist no guidelines in this thesis for the general security of an IoT pipeline. It is however possible to verify that the protocols for data transportation are secure. This is because the pipeline uses HTTPS when communicating between softwares. It also use .NET packages with the latest version, which incorporate SAS. Therefore, it is assumed that this is deemed safe and not a topic of discussion in this thesis. As mentioned in Section 2.5.6, both HTTPS and SAS are considered secure for usage in our pipeline.

8.2 Discussing Our Providers

In this thesis, frameworks and tools from various providers are utilized, and a discussion on how these providers fit into our thesis follows.

8.2.1 Discussing Miles and Miles Connect

Before starting this thesis, we were already in talks with Miles regarding a joint effort to see if the Miles Connect framework could work in an IoT scenario. The main goal being to set up a pipeline and use their framework where applicable, executing tests to see how well Miles Connect performed. When discussing with Miles, they wanted us to investigate if Miles Connect could be used in a pipeline scenario, therefore, a point of view is provided in regards to how well Miles Connect fit the pipeline.

Miles Connect

As discussed in Section 5.3, Miles Connect gave us the control needed to perform validation and conversion on the data passing through. Additionally, it made the creation of a pipeline intuitive with the logging and error handling features.

One of the research questions that Miles had was if it was possible to reduce the data size per message for optimal storage without losing any value. This means that for Miles, performing operations on the data is prioritized over raw throughput. The pipeline performs operations on the data before and after data is sent to the service bus. In these operations, data size can be reduced to only contain the necessary data points needed to gain insight at a later stage.

To further extend the reduction of messages stored in the pipeline, it is possible to collect data over a smaller period of time and calculate median data values. These calculated values give an overview of what the actual measurements were, but not the full picture. In the use case presented regarding a freezer with heat-sensitive products, or the basement with risk of flooding, a median calculated value might not suffice. In such scenarios any single value above a certain threshold should be processed, stored, and sent to an operator immediately.

In the third use case of this thesis, the temperature measurements in an office building, such an approach might be possible. In an office, one does not expect the temperature to drop or rise at an alarming rate, and a small rise or decline in temperature would not lead to any critical consequences. Therefore, to save resources, one might consider to calculate a median value over a defined period of time and only store this value. This should be done as soon as possible for optimal resource saving. In our thesis, this would first be possible in the data ingestation step.

In this thesis we did not have enough time to set up and test this research question, but from our experiences of the other tests executed, we deem it possible and that it would have reduced storage usage.

8.2.2 Discussion Disruptive Technologies

In one of the early dialogues with Miles regarding this thesis, we were recommended to investigate if DT had the resources that could be used. As the starting point was to collect data from IoT sensors, DT did indeed have the tools needed to play the IoT device part of a pipeline.

The fact that DT could provide access to their emulator quickly accelerated the start of the project. This gave us a solid foundation for collecting data from emulated IoT sensors. Furthermore, our emulated sensors could be replaced with real IoT sensors quite easily, as was one of our goals for this thesis. However, we were unable to conduct a test using real sensors due to time constraints. These constraints include misjudged time management for how long it would take to build and test our pipeline, as well as the actual delivery time of IoT sensors.

Manufacturing in DT

As DT has their own manufacturing line in Germany where they have complete control, a client only needs to trust DT and not all of DT's suppliers. This should increase trust in their IoT devices. This may result in higher IoT sensor costs, as it is often more expensive to produce in Europe than to outsource to other continents.

How has DT affected the project?

Using DT resources, the IoT parts of the pipeline were simple to set up, and the support team helped with the project when struggling at first. Additionally, the answers received from DT were a fine addition to the project, as an industry expert evaluated the guidelines.

In regards to edge processing, DT's SDS encryption technology makes it impossible to conduct edge processing on the Cloud Connectors. The limited size of DT's ASIC, DT Silicon, makes edge processing at the IoT sensor impossible as well. Even if the pipeline built would have included physical sensors, they would still stream data to DT Cloud with SDS. Both with, and without, physical sensors, the first available location for edge processing would have been data ingestation. Therefore, a use case with focus on edge processing may not be possible with DT's sensors and their encryption scheme.

For our base case, DT has proved to be a sufficient supplier of IoT sensors. For other more complex use cases such as edge processing and the basement scenario, another IoT provider could be considered.

Alternative IoT providers

It is relevant to examine if another IoT sensor provider could be used. At the project start, no preference regarding where the IoT data was coming from, was present. Therefore, at the beginning, *any* IoT provider would suffice, as the pipeline would have adapted to the provider's requirements.

Even though our pipeline is created using DT as our sensor provider, the pipeline should be able to handle data from any provider, due to its modularity.

8.2.3 Discussing Azure

Section 5.4 is when Azure Service Bus as the message broker for this thesis is introduced. In the thesis, no particular focus has been given on which vendor was chosen for the message broker part. A study of existing enterprise service buses was explored, where Azure Service Bus was one of the services that scored the best.

The selection of Azure was in part based on this, and in part based on the authors, and external supervisors, experiences with this technology.

The purpose of this thesis was to build a general pipeline that was independent of cloud service providers. This is also why, rather than selecting a single cloud service provider and using all of their components, multiple providers for the steps in our pipeline were used.

Due to the modularity of the pipeline there should be no big issue to transfer the message broker step from Microsoft's Azure to AWS, Google Cloud, or any other provider if that turns out to be the better option.

8.3 Discussing the Guidelines

This thesis has resulted in 22 distinct guidelines aimed at helping a developer build a secure and reliable IoT data pipeline. But did these 22 guidelines actually improve pipeline quality more than if no guidelines existed?

8.3.1 A secure and reliable pipeline

In Chapter 2, the QoS requirements for a pipeline being deemed secure and reliable is introduced. A secure pipeline should not experience breaches in the domains of confidentiality, integrity and authenticity. A reliable pipeline should not have a breach in availability and have sufficient fault tolerance.

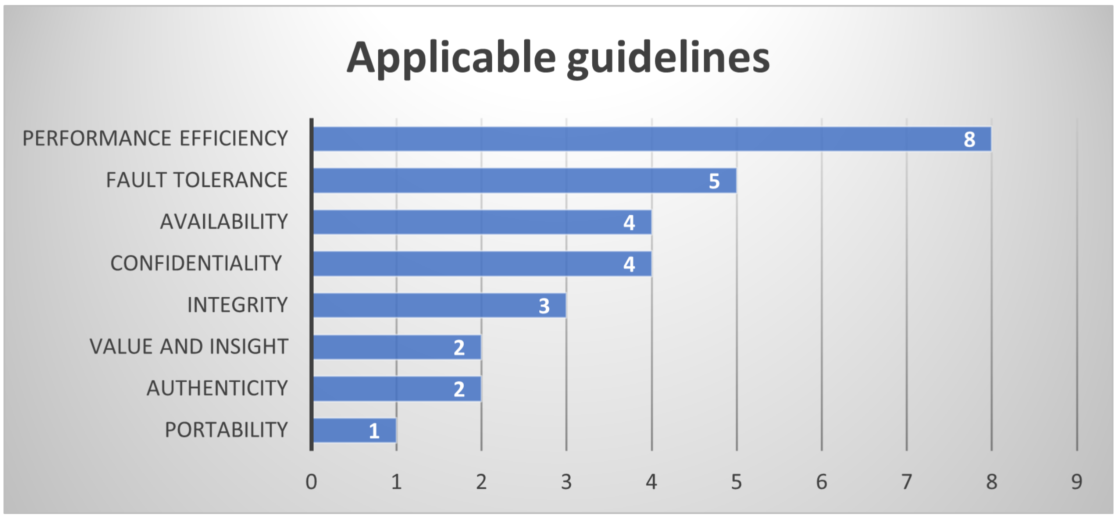
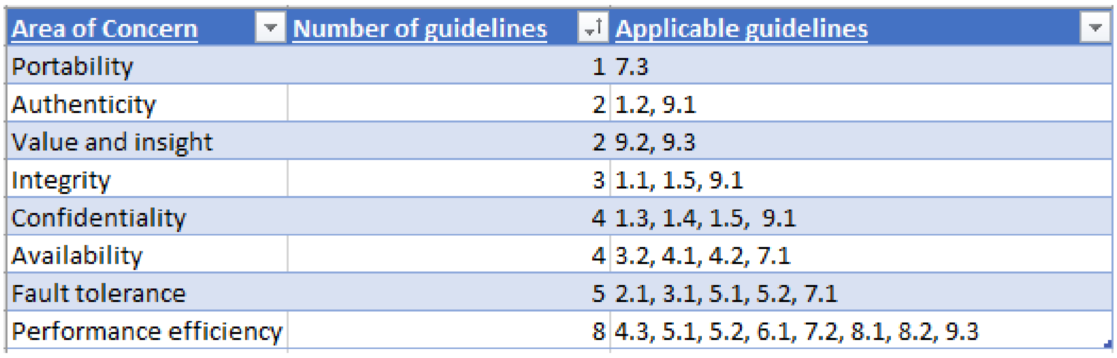


Figure 8.2: Guidelines related to each terminology from Section 2

In Figure 8.2, the QoS requirements are listed with the number of guidelines deemed applicable to them. The source for these numbers is a classification done by the authors, and is shown in Figure 8.3.



Area of Concern	Number of guidelines	Applicable guidelines
Portability	1	7.3
Authenticity	2	1.2, 9.1
Value and insight	2	9.2, 9.3
Integrity	3	1.1, 1.5, 9.1
Confidentiality	4	1.3, 1.4, 1.5, 9.1
Availability	4	3.2, 4.1, 4.2, 7.1
Fault tolerance	5	2.1, 3.1, 5.1, 5.2, 7.1
Performance efficiency	8	4.3, 5.1, 5.2, 6.1, 7.2, 8.1, 8.2, 9.3

Figure 8.3: Applicable guidelines table

As can be seen, the QoS requirement with the most applicable guidelines is performance efficiency, with 8 related guidelines. Followed by fault tolerance and availability. These are the two components that make up the reliable trait.

Confidentiality, integrity, and authenticity make up the three components of the secure trait. This means that both the reliable and the secure trait have 9 applicable guidelines each. Value and insight, and the portability trait, has two and one related guideline, respectively.

The portability trait regards the terms agnostic and modularity. This trait is related to Guideline 7.3, which references the message broker. The message broker is the key central part where data is exchanged, thus, one guideline should be enough to ensure an agnostic system. Even though no specific guideline is set up for the modular trait. The fact that the pipeline is divided into 5 distinct steps ensures modularity.

The two guidelines regarding value and insight, Guidelines 9.2 and 9.3, might seem too limited, and too wide. This is however, due to the fact that value and insight are two very broad terms. Thus, specifying these guidelines further would most certainly make them irrelevant for many use cases and potential pipelines. As the goal was a general pipeline with general guidelines, specifying the guidelines further would work against the goal of this thesis.

Overall the guidelines are evenly distributed between performance efficiency, security, and reliability. These three traits are the biggest three focuses of the guidelines, which fit quite well with the intended goal.

8.3.2 Following the guidelines

Particularly helpful guidelines

It can be difficult to evaluate the helpfulness of the guidelines. Speaking out of our own experiences, some guidelines were more helpful than others. The security guidelines for IoT devices (Guidelines 1.1 – 1.5) meant that additional research was conducted to ensure that the IoT devices were safe. This is most certainly something most developers take for granted, and thus, these act as a reminder of ensuring safety. Therefore these are deemed particularly helpful. The same can be said for Guideline 2.1, where the poor signal strength environment is probably something that could go unnoticed until after the sensors are set up. As emulators were used, we did not experience this, but the reminder this guideline presents is helpful.

The security of the applications from Guideline 9.1, is deemed helpful as more often than not, users will use applications without vetting them. Thus, this guideline ensures that a user or developer should be certain that the applications are safe before implementing them.

Less helpful guidelines

The guideline regarding battery-scheme (3.1), is an interesting one. The person setting up an IoT data pipeline most likely do not have any knowledge towards battery consumption on IoT devices, thus battery specifications might not mean much to the developer. This means that the guideline is too specific to offer any help for this particular person.

If the developers have this deep knowledge regarding batteries, they probably already know of the importance of battery life, thus this guideline becomes too general to offer any help. Therefore, this guideline can be seen as less helpful.

The guidelines for the message broker (7.1 – 7.3), can be seen as less helpful due to the fact that any developer most likely chooses a message broker from one of the existing ones. This means that these guidelines more often than not are fulfilled from the start.

The guidelines regarding benefiting the company with cost effectiveness and valuable insight (9.2, 9.3), may be unnecessary as this is often the point with setting up an IoT data pipeline. It is worth mentioning that one should have a keen eye towards how much one actually pays and how much value one actually gets. But the generality of these pipelines makes us deem them less helpful.

Use case specific guidelines

The guidelines regarding scaling (3.2, 4.1, 4.2) depends on the specific use case. DT offer a solution where each gateway can support up to 10 000 devices. Thus, a developer using DT most likely would have no need for these guidelines. The guidelines are very useful if one has more control of the IoT devices and gateways, and are able to program them. These guidelines can then ensure that the pipeline can be scaled for future needs without replacing important parts.

Guideline 4.3, regarding edge processing is also use case dependent. For our use cases, this was solved at a later stage in the pipeline as discussed in Section 7.2.3. A use case with different sensors or different communication protocols could ensure that the sensors or the gateway can make use of edge processing. In some use cases, this is necessary for the functionality of the pipeline, often in areas with much noisy data. In other use cases, this guideline is close to irrelevant, as no edge processing is needed.

The guidelines regarding formatting and validation (5.1, 5.2), streaming vs ETL (6.1) and application-specific processing (8.1, 8.2) are also highly dependent on the use case. If one can guarantee correct formatting and correct values from the get-go, most of these become irrelevant. But when integrating towards a bunch of endpoints, this is not often the case. As a result, these guidelines might help the pipeline from constantly running into errors down the line, and thus play a part in creating a robust integration strategy.

For Guideline 6.1, our tests have shown that using ETL as a data transfer architecture works great up until a certain point. If one never reaches this point, ETL can manage just fine. In other cases where huge amounts of data is to be ingested, event streaming probably has to be used.

Breaches in the guidelines

Even though the guidelines were produced in this thesis, not all of them were followed. The pipeline breached 2 guidelines, most notably Guideline 6.1, regarding data transfer architecture. However, the pipeline seemed to manage fine without event streaming. The other breach in our pipeline was the edge processing guideline (4.3). This was due to the way DT's technology and devices are built. However, this was compensated by processing the data at the earliest convenience.

As not all guidelines were followed, one can not expect other developers to follow them all. This may be a motivation towards researching if guidelines are the best way to ensure a secure and reliable pipeline, or if alternative solutions are better.

8.3.3 Do developers need guidelines?

When setting up the IoT data pipeline, we found that guidelines, or similar solutions, for how to build a pipeline were not present. There seemed to be no best practices or standards which could be followed. This led to us researching more on the topic of IoT pipelines, and found that a set of guidelines might be useful. Not just for us to keep control of the quality of the pipeline, but also for others in a similar situation.

Starting a project with spending numerous hours on reading research papers to find out how to build a reliable pipeline, is not a suitable approach for any pipeline development. Thus, guidelines provide an easy way of ensuring that the pipeline being built follows best practices. Or at least, avoids making the same mistakes as previous existing systems.

These guidelines do not guarantee that the pipeline will operate perfectly, but they grant the developer guidance when building an IoT data pipeline.

Alternative solutions

These guidelines could be the start of a set of best practices, or even result in IoT data pipelines being standardized. This would greatly affect the quality of pipelines being built. This however, would take quite some time, and would include working with large technology corporations as well as governments. An industry standard can be seen being the most helpful and hardest to achieve. A set of best practices and know-hows, would also benefit a developer.

8.4 Feedback From the Industry

In this section relevant feedback from the companies assisting us in this thesis is discussed. They are as mentioned; Asplan Viak, Miles and Disruptive Technologies.

8.4.1 Adjusting the pipeline

After successfully setting up our pipeline we had a talk with Asplan Viak AS as a potential test case provider and got recommendations to consider using technologies that were better suited for their IoT uses. They recommended an event-streaming pipeline using Azure Event Hub and Azure Time Series Insight.

After researching these two technologies it was found that Azure Time Series Insight is to be deprecated within 2025. No further effort into investigating other possible migrations other than Azure Time Series were made, as we already had set up a functional pipeline.

Service bus vs. Event hub

Azure Event Hub is a version of Azure Service Bus specially designed for IoT devices. However, for this thesis, they did not differ greatly enough for us to notice any particular difference in our use cases. As mentioned in Section 7.3.3, the service bus offers asynchronous processing and other useful features.

Throughout this thesis, a service bus was used as the message broker, but it would also be possible to use an event hub. When first building the pipeline, research was conducted in how to transport data using a message broker. The developers of Miles use a service bus themselves when they are using the tools of Miles Connect. This rationalised the decision to use a service bus in this pipeline as well.

As mentioned Asplan Viak suggested using the event hub service as it would be useful when handling IoT data. Because it was decided not to focus on the application suggested by Asplan Viak, we deemed it unnecessary to transfer to an event hub as our message broker. During the development, both services were set up, but it was decided to proceed with the service bus as it gave us the expected results.

8.4.2 Feedback from Miles

To get feedback from the industry, the guidelines were sent to a few developers at Miles. The feedback we got is located in Appendix A.

Encryption

The first feedback from Miles was regarding Guideline 1.3 about encryption. The suggestion being that encryption might not be necessary when handling insensitive data. The example being used was temperature sensors, and that temperature values might not need encryption. While this is a fair point, we argue that not encrypting data leaves the pipeline vulnerable to a MITM attack (cf. Section ??). Any data not encrypted can be intercepted, read, and changed, thus breaking both confidentiality and integrity. In addition, the malicious actor can choose to block or delay certain messages, affecting reliability and performance efficiency as well.

Guideline wording

We received feedback for the wording in some of the guidelines. The guidelines should be general and simple enough such that most people who read them will understand them. The background for a given guideline, and its solution, should be understood with any further explanation. This was not made clear as one of the respondents from Miles did not understand the abbreviation used in Guideline 6.1, where ETL is mentioned. To ensure that such misconceptions are avoided, the guidelines should be clear and not include any abbreviations.

In addition, the wording in Guideline 7.3 was mentioned. In this guideline, the feedback was that the *applications* terminology was too wide, and should be defined more clearly.

We acknowledge this feedback, but as a pipeline should be able to handle all different applications we do not see how to specify it any further.

Adding How to's

The last feedback from Miles examined the possibility of adding *How to's* to guidelines. This would be a set of best practices as discussed in Section 8.3.3. The developer was unsure if this was the best approach or not, but that it should be taken into consideration.

8.4.3 Feedback from DT

The feedback from DT is widely used in Chapter 7 in order to verify some of the guidelines. The answers from the questionnaire are focused on the IoT device and gateway specifications, to better understand how their products fulfill the guidelines. In addition, feedback on the guidelines as a whole were positive.

Chapter 9

Conclusions

The purpose of this thesis was to identify what steps exist in an IoT data pipeline, and what the most important aspects of the pipeline is, in order to make it secure and reliable. This thesis used qualitative research to locate areas of concern, highlighting and addressing them in guidelines. Then, building an IoT pipeline and verifying with quantitative research that these areas of concern are handled by following said guidelines.

This thesis was able to identify and characterize the different parts of a typical IoT pipeline, and locate areas of concern for each part in that pipeline. It provides 22 recommendations in the form of guidelines for setting up the pipeline. During this thesis a fully functioning IoT pipeline using Miles Connect was built. This thesis was unable to test and evaluate the pipeline on real world applications. The results indicate that building a pipeline by following these guidelines, should address these concerns, and therefore should reduce issues in the future of the pipeline.

This research was conducted with three main methodologies; lessons learned, talking to the industry, and a literature study. It can be concluded that using these three methodologies together with each other gives a broader approach than using only one of the three.

The authors gained new insight in the importance of choosing a correct data transfer architecture. The differences between Kafka and Miles Connect were major, even though they operate at the same step, data ingestation. This leads to the conclusion that the data ingestation step can be a limiting factor, or a bottleneck in some use cases, if an unfit data ingestation software is used.

To summarize, the motivation for this thesis was that there were no general standards or best practices when building an IoT data pipeline. This work addresses the gaps in knowledge in this area, by researching what information exists, and validating the solutions with the help of empirical testing and industry experts.

The 22 guidelines grants practitioners some form of guidance when building an IoT pipeline, and the authors recommendation is that these guidelines should be followed.

9.1 Future Work

To better understand the implications of these results, further studies could address a deeper benchmarking of Miles Connect, and the system as a whole. In addition, evaluating the pipeline using real sensors, and digging deeper into the evaluation of the guidelines are also focus areas that can be addressed.

9.1.1 Benchmarking Miles Connect, and the system

When performing benchmarking, the system should run on a similar environment, with equal specifications, as the comparing software. This would ensure that no other external features have influence on the test in any way.

This thesis did not achieve this, and therefore a future project might benchmark Miles connect using a similar environment as Kafka. This would ensure that the comparison to Kafka would be more equal, and thus more relevant. Taking this further, to create an optimal solution, benchmarking every step of the pipeline can be done. This would also reveal any hidden bottleneck diminishing the overall performance of the system.

9.1.2 Real use case

As this thesis did not manage to achieve the goal of testing the pipeline on a real world application, this can be performed at a later stage. Even though several use cases have been introduced, and validation efforts have been made on each step of the pipeline, a real world application is best suited to observe the performance of the pipeline.

9.1.3 Evaluate guidelines as a resource

To our knowledge, this thesis presents the first set of guidelines, and first independent resource, for building IoT data pipelines. The guidelines have been created to be generally applicable in most scenarios, and aims at guiding a developer to achieve a higher quality of the pipeline.

Without access to a similar tool, evaluating if the guidelines are helpful or not, is difficult. A research where one could evaluate if another type of resource is better suited than guidelines should be conducted. As no such resource exists at the moment, the guidelines themselves are a good baseline for defining other tools that could further increase the quality of future IoT pipelines. In addition, these guidelines need not be set in stone and can be changed when new technological advances are accomplished, thus a refreshed study exploring new technology might be conducted at a relevant time.

Appendix A

In Appendix A the questionnaires and answers received are provided for any interested parties. The yellow and red text is the response from the respondents.

DISRUPTIVE TECHNOLOGIES QUESTIONNAIRE

Our Master Thesis has resulted in a few guidelines that should prove useful when designing an IoT data pipeline.

An IoT Pipeline is defined as the process from which Data is generated at a sensor, through a gateway, into a message broker, and out to a consuming component.

Following the guidelines mentioned should help create a more secure and reliable pipeline.

As Disruptive Technologies is an important part of that pipeline we want to verify if Disruptive Technologies already is following these guidelines.

We also want to take this opportunity to get feedback if there are any guidelines that are missing in your opinion, that should be an essential part of an IoT Pipeline.

Guideline: IoT devices should be created by a trusted source. The users should be authenticated with modern standards. Common IoT security measures such as encryption should be implemented on the device.

Question 1: How is security measures like encryption and authentication implemented in Disruptive Technologies IoT devices?

https://www.disruptive-technologies.com/blog/security-and-privacy-in-disruptive-technologies-sensing-solution

Question 2: How is the IoT devices produced by a trusted source?

DT owns its own manufacturing line in a factory in Germany with strict restrictions on access.

Guideline: To ensure that each device has the latest software which patches security flaws, software used in IoT devices should have the newest version.

Question 3: Is it possible to update Disruptive Technologies sensors with software updates or security patches after they have been installed at a location?
How is this done?

Yes, we can update all devices with over-the-air updates to change firmware or configurations. We have a fleet management system that facilitates this. The updates are fully automatic and require no involvement from the customer.

Guideline: IoT devices should employ a battery-saving scheme that suffices for the situation the device is used in.

Question 4: What type of battery-saving scheme does Disruptive Technology sensors use since they have an battery life of 10+ years?

https://www.disruptive-technologies.com/blog/embedded-firmware-development-for-an-extremely-resource-constrained-system

Guideline: The gateway should scale gracefully with a growing number of devices in the architecture.
Guideline: A gateway should be able to connect to other gateways to expand the load of data, working in parallel.

Question 5: How does Disruptive Technology gateways/data connectors scale if the density number of IoT devices in a small area is higher than the supported capability?

Each gateway supports more than 10.000 sensors. We do not expect the number of devices to be a limiting factor for a long time. The cloud services have load-balancing features that scale with the number of data points to process.

Guideline: Be able to perform edge computing: pre-processing, cleansing, and filtering.

Question 6: Do the IoT Devices or data connectors support edge computing: pre-processing, cleansing, and filtering of any kind?

The gateway does not support edge computing as the sensor measurements are fully end-to-end encrypted from the sensor to the cloud.

In addition, we present these guidelines:

- The pipeline should be able to control that the device load and bandwidth is not exhausted, and scale resources if necessary.

- The data transfer should use event streaming or similar technology, avoiding ETL. This allows for fast and reliable data transfer.

- Message brokers must ensure that the consumer will receive its messages.

- Message brokers should have the ability to perform asynchronous processing.

- The message broker should allow different applications to communicate regardless of the programming language.

- Perform small data processing operations to increase data quality.

- Conduct application-specific validation before delivery is complete.

- The applications in a pipeline should be verified as secure.

- An application should generate valuable insight from the data it receives.

- The selection of applications should benefit the company in a cost-
efficient manner

Question 7: Looking at all guidelines in question, are there any guidelines missing that should be essential advice in an IoT Pipeline?

Looks good :)

MILES QUESTIONNAIRE

Our Master Thesis has resulted in a few guidelines that should prove useful when designing an IoT data pipeline.

An IoT Pipeline is defined as the process from which Data is generated at a sensor, through a gateway, ingested into a message broker, and out to a consuming component.

Following the guidelines mentioned should help create a more secure and reliable pipeline.

As Miles Connect is an important part of that pipeline we want to take this opportunity to get feedback if there are any guidelines that are missing in your opinion, that should be an essential part of an IoT Pipeline.

Feel free to comment on specific guidelines or the guidelines as a whole. At the end of this paper, feel free to comment if anything substantial is missing.

Guideline: IoT devices should be created by a trusted source. The users should be authenticated with modern standards. Common IoT security measures such as encryption should be implemented on the device.

Comment(if any):

Maybe only if the data is sensitive. Might not be necessary to encrypt temperatures from a sensor?

Guideline: The data transfer should use event streaming or similar technology, avoiding ETL. This allows for fast and reliable data transfer.

Comment(if any):

ETL?

Guideline: The message broker should allow different applications to communicate regardless of the programming language.

Comment(if any):

Maybe expand a little bit on what applications it is?

General Comments(if any):

Maybe add some how’s to the guidelines or maybe guidelines shouldn’t include how’s?

Appendix B

Source code, Image Sources and Project Poster

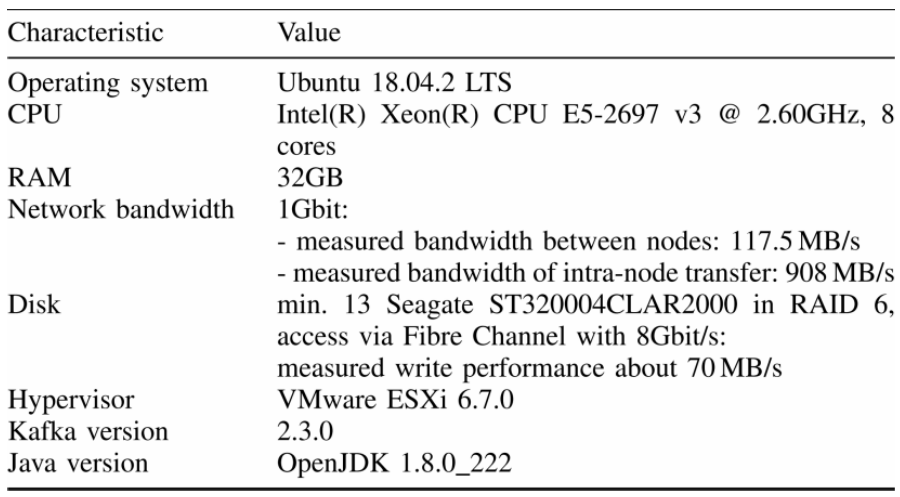
B.1 Source Code

The source code for this thesis is located at the following Google drive link: Source Code.

Appendix B.2 has the various image sources for illustrations in this project, in addition to the specifications used when Kafka was tested. Appendix B.3 includes the project poster for this thesis.

B.2 Image Sources

B.2.1 Kafka Virtual Machine specifications



Characteristic	Value
Operating system	Ubuntu 18.04.2 LTS
CPU	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz, 8 cores
RAM	32GB
Network bandwidth	1Gbit: - measured bandwidth between nodes: 117.5 MB/s - measured bandwidth of intra-node transfer: 908 MB/s
Disk	min. 13 Seagate ST320004CLAR2000 in RAID 6, access via Fibre Channel with 8Gbit/s: measured write performance about 70 MB/s
Hypervisor	VMware ESXi 6.7.0
Kafka version	2.3.0
Java version	OpenJDK 1.8.0_222

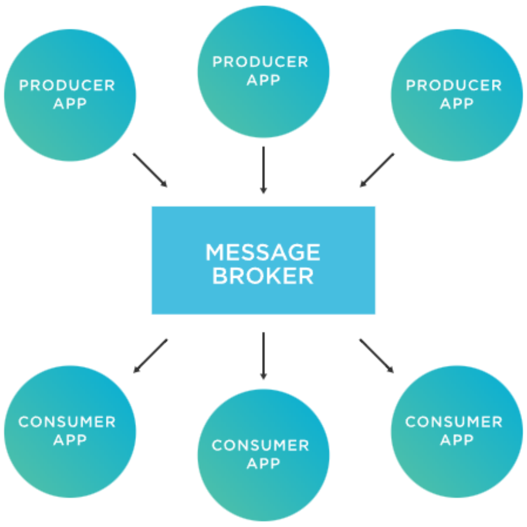
Figure B.1: Kafka Virtual Machine

In Figure B.1, we present the specifications used on their test in article [31], which we compare in Section 7.3.1.

B.2.2 Illustration Sources

IoT applications in agriculture[86]

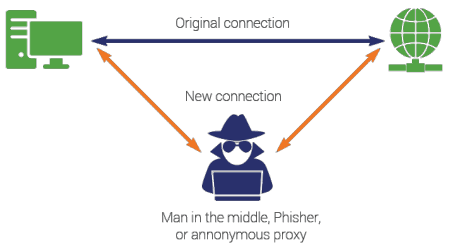
IoT devices [7]



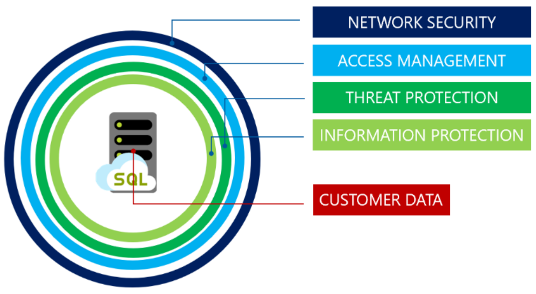
Message broker [87]

Applications [88]

People counting [89]



Man in the middle attack [66]



Azure SQL database security overview [83]

B.3 Poster

In the following attachment, the project poster which was presented at UiS 1st of June 2022, is included.

Introducing Guidelines to IoT Data Pipelines

Overview

- Introduce guidelines to improve security and reliability in the development of IoT data pipelines.
- In our thesis we present and evaluate 22 unique guidelines.

Methodology

- Our guidelines is a product of:
  Lessons learned - Security in child's watches
  Industry Experts - Miles AS, Disruptive Technology, Asplan Viak AS
  Literature study - Research the need of guidelines in the sector

Lessons Learned - Example

- In 2017 the Norwegian Consumer Council discovered significant security and privacy flaws in smart watches for kids sold from many known vendors in Norway
- Strangers could be able to take control, eavesdrop and communicate with children via their smartphone
- They could alter the clock's GPS signal misleading parents of their child's actual location
- The watch's data was not encrypted, and it was impossible for users to delete their own data

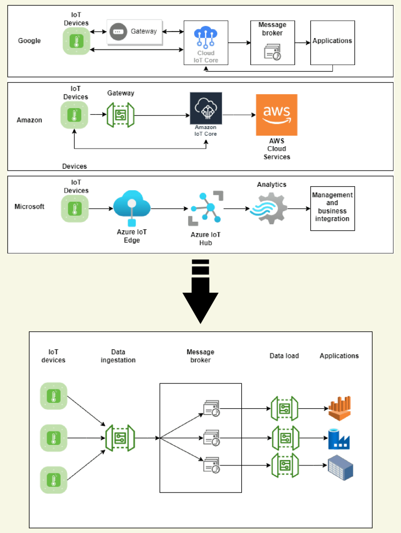
Background and Motivation

Given the vast majority of IoT pipeline vendors and lack of standardization, guidelines regarding the requirements of an IoT pipeline is needed to be able to create a secure and reliable IoT pipeline.

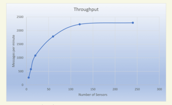
A Selection of Guidelines

- The IoT devices in use should be created by a trusted source.
- Common IoT security measures such as encryption should be implemented on the IoT device.
- The gateway should scale gracefully with a growing number of devices in the architecture.
- The data ingestion step should validate that the incoming values are correct and usable, remove any invalid data.
- Message brokers must ensure that the consumer will receive its messages.
- The data transfer should use event streaming or similar technology, avoiding ETL. This allows for fast and reliable data transfer.
- The data load step should perform small data processing validations to increase data quality.
- The applications in a pipeline should be verified as secure

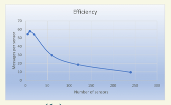
A Multitude of Pipelines Available



Results From Guideline 6.1



(a) Throughput



(b) Efficiency

Conclusion

- We presenting a list consisting of 22 guidelines divided by five defined IoT data pipeline steps.
- Each of these guidelines are verified by academic research, industry experts, vendor documentation, and tests performed in the thesis.
- Using these guidelines should support and guide anyone who seeks out to develop an IoT data pipeline.

Brynjar Steinbakk Ulriksen and Tor-Fredrik Torgersen
01/07/2022 University of Stavanger
Department of Electrical Engineering and Computer Science

Bibliography

[1] Victoria State Government. Internet of things in agriculture. *Agriculture Victoria*, 2022. URL https://agriculture.vic.gov.au/farm-management/agtech/introduction-to-agtech/internet-of-things-in-agriculture. [Accessed: 2022-05-27].

[2] V.Tudor A.Hernandez, B.Xiao. How do we solve a problem like iot data pipelines? 2020. URL https://www.ericsson.com/en/blog/2020/11/iot-data-pipelines. [Accessed: 2022-04-25].

[3] G.Onag Future IoT. Building an iot pipeline. 2021. URL https://futureiot.tech/building-an-iot-data-pipeline/. [Accessed: 2022-04-25].

[4] Mary K. Pratt TechTarget. Reach business objectives with the right iot data pipeline. 2021. URL https://www.techtarget.com/iotagenda/feature/Reach-business-objectives-with-the-right-IoT-data-pipeline. [Accessed: 2022-04-25].

[5] Gary White, Vivek Nallur, and Siobhán Clarke. Quality of service approaches in iot: A systematic mapping. *Journal of Systems and Software*, 2017. URL https://www.sciencedirect.com/science/article/pii/S016412121730105X. [Accessed: 2022-04-25].

[6] Leverege. What is iot? 2022. URL https://www.leverege.com/iot-ebook/what-is-iot. [Accessed: 2022-05-27].

[7] Zigurat. Iot device illustration. 2022. URL https://www.e-zigurat.com/innovation-school/blog/what-do-the-next-five-years-hold-for-the-iot/. [Accessed: 2022-05-02].

[8] In Lee and Kyoochun Lee. The internet of things (iot): Applications, investments, and challenges for enterprises. *Business Horizons*, 58(4):431–440, 2015. ISSN 0007-6813. doi: https://doi.org/10.1016/j.bushor.2015.03.008. URL https://

www.sciencedirect.com/science/article/pii/S0007681315000373. [Accessed: 2022-02-11].

[9] RD Statista. Internet of things-number of connected devices worldwide 2015-2025. *Statista Research Department. statista. com/statistics/471264/iot-numberof-connected-devices-worldwide*, 2019. [Accessed: 2022-02-11].

[10] Telia. Low power wide area iot. 2022. URL https://business.teliacompany.com/internet-of-things/iot-connectivity/LPWA-IoT. [Accessed: 2022-04-25].

[11] Google Cloud. Iot core. 2022. URL https://cloud.google.com/iot-core/. [Accessed: 2022-04-14].

[12] AWS. Aws iot core. 2022. URL https://aws.amazon.com/iot-core/?c=i&sec=srv. [Accessed: 2022-04-25].

[13] Azure. Azure iot hub. 2022. URL https://azure.microsoft.com/en-us/services/iot-hub/. [Accessed: 2022-04-25].

[14] Philipp Wegner IoT Analytics. The top 10 iot use cases. 2021. URL https://iot-analytics.com/top-10-iot-use-cases/. [Accessed: 2022-02-16].

[15] Rob Faludi Digi. How do iot devices communicate? 2021. URL https://www.digi.com/blog/post/how-do-iot-devices-communicate. [Accessed: 2022-02-16].

[16] Gerard Adlum. What is the impact of customer data silos and how can you fix the problem? 2022. URL https://xtremepush.com/what-is-the-impact-of-customer-data-silos-and-how-can-you-fix-the-problem/. [Accessed: 2022-04-25].

[17] Ana Neto. Why are it system integration costs so high? 2021. URL https://www.connecting-software.com/blog/infographics-why-are-it-system-integration-costs-so-high/. [Accessed: 2022-05-27].

[18] The things network. What are lora and lorawan? 2022. URL https://www.thethingsnetwork.org/docs/lorawan/what-is-lorawan/. [Accessed: 2022-02-11].

[19] Smarte Byer Norge. Hva er lorawan og hvordan kan det gi innbyggerne bedre tjenester? 2019. URL http://www.smartebyernorge.no/nyheter/2019/11/19/iot-nettverk-gir-innbyggerne-bedre-tjenester. [Accessed: 2022-02-11].

[20] Synnøve Meisingset. Nye nett kan revolusjonere kommunale tjenester:nå skal også søppelkassen på internett. 2021. URL

https://www.dn.no/teknologi/teknologi/tingenes-internett-iot/nye-nett-kan-revolusjonere-kommunale-tjenester-na-skal-ogsa-soppelkassen-pa-inte
2-1-959925. [Accessed: 2022-02-11].

[21] IBM Cloud Education. Soa (service-oriented architecture). 2021. URL https://www.ibm.com/cloud/learn/soa. [Accessed: 2022-02-11].

[22] IBM Cloud Education. Esb (enterprise service bus). 2021. URL https://www.ibm.com/cloud/learn/esb. [Accessed: 2022-02-11].

[23] Martin Potočnik and Matjaz B Juric. Integration of saas using ipaas. In *The 1st international conference on CLoud Assisted ServiceS*, page 35, 2012. [Accessed: 2022-02-11].

[24] Disruptive Technology AS. A truly disruptive technology. 2022. URL https://www.disruptive-technologies.com/technology. [Accessed: 2022-04-14].

[25] SSL.com Support Team. What is https? *SSL.com*, 2021. URL https://www.ssl.com/faqs/what-is-https/. [Accessed: 2022-06-02].

[26] Google. Secure your site with https. *Google Documentation*, 2022. URL https://developers.google.com/search/docs/advanced/security/https. [Accessed: 2022-06-02].

[27] Microsoft. Service bus access control with shared access signatures. *Microsoft Docs*, 2022. URL https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-sas. [Accessed: 2022-06-02].

[28] Morgan Geldenhuys, Lauritz Thamsen, Kain Gontarska, Felix Lorenz, and Odej Kao. Effectively testing system configurations of critical iot analytics pipelines. 02 2021. [Accessed: 2022-02-11].

[29] Kafka. Powered by. 2022. URL https://kafka.apache.org/powered-by. [Accessed: 2022-02-11].

[30] Top 10 apache kafka alternatives and competitors. 2022. URL https://www.g2.com/products/apache-kafka/competitors/alternatives. [Accessed: 2022-02-11].

[31] Guenter Hesse, Christoph Matthies, and Matthias Uflacker. How fast can we insert? an empirical performance evaluation of apache kafka. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 641–648, 2020. doi: 10.1109/ICPADS51040.2020.00089. [Accessed: 2022-02-11].

[32] Paul Le Noac'h, Alexandru Costan, and Luc Bougé. A performance evaluation of apache kafka in support of big data streaming applications. In *2017 IEEE*

International Conference on Big Data (Big Data), pages 4803–4806, 2017. doi: 10.1109/BigData.2017.8258548. [Accessed: 2022-02-11].

[33] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 69–78, 2014. doi: 10.1109/UCC.2014.15. [Accessed: 2022-03-29].

[34] Omogbai Oleghe and Konstantinos Salonitis. A framework for designing data pipelines for manufacturing systems. *Procedia CIRP*, 2020. URL https://www.sciencedirect.com/science/article/pii/S2212827120305618. [Accessed: 2022-04-25].

[35] Bruton K. et al. O'Donovan P., Leahy K. An industrial big data pipeline for data-driven analytics maintenance applications in large-scale smart manufacturing facilities. *Big Data 2, 25*, 2015. URL https://journalofbigdata.springeropen.com/articles/10.1186/s40537-015-0034-z#citeas. [Accessed: 2022-04-25].

[36] Amir Kotler iotforall. Five most common problems iot devices will encounter in 2020. 2020. URL https://www.iotforall.com/iot-problems.

[37] The norwegian consumer council. Significant security flaws in smartwatches for children. 2017. URL https://www.forbrukerradet.no/side/significant-security-flaws-in-smartwatches-for-children/. [Accessed: 2022-04-25].

[38] S. Langkamper. Essential requirements for securing iot consumer devices. *Eurofins cybersecurity*, . URL https://www.eurofins-cybersecurity.com/news/requirements-secured-iot-consumer-devices/. [Accessed: 2022-05-02].

[39] S. Langkamper. The most important security problem with iot devices. *Eurofins cybersecurity*, . URL https://www.eurofins-cybersecurity.com/news/security-problems-iot-devices/. [Accessed: 2022-05-02].

[40] WilsonPro. Why the future of iot depends on strong cell signal. 2019. URL https://www.wilsonpro.com/blog/why-the-future-of-iot-depends-on-strong-cell-signal. [Accessed: 2022-04-25].

[41] Disruptive Technology AS. Embedded firmware development for an extremely resource-constrained system. 2022. URL https://www.disruptive-technologies.com/blog/embedded-firmware-development-for-an-extremely-resource-constrained-system. [Accessed: 2022-05-22].

[42] P. Seltsikas and W.L. Currie. Evaluating the application service provider (asp) business model: the challenge of integration. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 2801–2809, 2002. doi: 10.1109/HICSS.2002.994341. [Accessed: 2022-02-11].

[43] AWS. What is aws iot? 2022. URL https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html/. [Accessed: 2022-04-14].

[44] B. Posey. Iot gateway. *IoT Agenda*, 2022. URL https://www.techtarget.com/iotagenda/definition/IoT-gateway/. [Accessed: 2022-05-02].

[45] Open Automation Software. What is an iot gateway? *OAS*. URL https://openautomationsoftware.com/open-automation-systems-blog/what-is-an-iot-gateway/. [Accessed: 2022-05-02].

[46] R. Miller. Data preprocessing: what is it and why is it important? *CEOWORLD magazine C-Suite Agenda*, 2019. URL https://ceoworld.biz/2019/12/13/data-preprocessing-what-is-it-and-why-is-important/. [Accessed: 2022-05-02].

[47] Mark Smallcombe. Data ingestion vs. etl: Differences & how to leverage both. 2020. URL https://www.integrate.io/blog/data-ingestion-vs-etl/. [Accessed: 2022-03-29].

[48] Infoq. Databases and stream processing: a future of consolidation. 2022. URL https://www.infoq.com/presentations/streaming-databases/. [Accessed: 2022-05-02].

[49] H. Subhashana. Introduction to message brokers. *Medium*, 2021. URL https://hasithas.medium.com/introduction-to-message-brokers-c4177d2a9fe3. [Accessed: 2022-05-02].

[50] Azure. Asynchronous messaging primer. *Azure*, 2015. URL https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn589781(v=pandp.10). [Accessed: 2022-05-27].

[51] Safe. What is data validation? 2022. URL https://www.safe.com/what-is/data-validation/. [Accessed: 2022-04-25].

[52] f5. Is having too many apps expanding your threat surface? 2022. URL https://www.f5.com/solutions/secure-cloud-architecture/is-having-too-many-apps-expanding-your-threat-surface. [Accessed: 2022-04-25].

[53] Roberto Torres. Enterprise app sprawl swells, with most apps outside of it control. 2021. URL https://www.ciodive.com/news/app-sprawl-saas-data-shadow-it-productiv/606872/.

[54] David Jones. Security leaders: Expect more insider data leaks, threats in 2021. 2021. URL https://www.cybersecuritydive.com/news/insider-threat-network-perimeter/597482/. [Accessed: 2022-04-25].

[55] Nordic APIs. The bezos api mandate: Amazon's manifesto for externalization. 2021. URL https://nordicapis.com/the-bezos-api-mandate-amazons-manifesto-for-externalization/. [Accessed: 2022-04-25].

[56] Redhat. What is edge architecture? 2022. URL https://www.redhat.com/en/topics/edge-computing/what-is-edge-architecture?sc_cid=7013a000002pu40AAA&gclid=CjwKCAjwgr6TBhAGEiwA3aVuISIRDvXl0eG4xt2Fzdm_oON9rPYBTfGj0wOZ0Q137vIa2hFyGk6hghoC9pAQAvD_BwE&gclsrc=aw.ds. [Accessed: 2022-05-03].

[57] Martin Fowler. Yagni. 2015. URL https://martinfowler.com/bliki/Yagni.html. [Accessed: 2022-04-25].

[58] Deviq. Separation of concerns. 2022. URL https://deviq.com/principles/separation-of-concerns. [Accessed: 2022-05-03].

[59] Microsoft Docs. What is azure service bus? 2022. URL https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview. [Accessed: 2022-02-16].

[60] Robin Singh Bhadoria, Narendra S. Chaudhari, and Geetam Singh Tomar. The performance metric for enterprise service bus (esb) in soa system: Theoretical underpinnings and empirical illustrations for information processing. *Information Systems*, 65:158–171, 2017. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2016.12.005. URL https://www.sciencedirect.com/science/article/pii/S0306437915301952. [Accessed: 2022-03-29].

[61] Disruptive Technologies. Disruptive technologies developer documentation. 2022. URL https://developer.disruptive-technologies.com/docs/. [Accessed: 2022-05-07].

[62] Hangfire. Hangfire homepage. 2022. URL https://www.hangfire.io/. [Accessed: 2022-05-03].

[63] Disruptive technologies. Plot sensor data. 2022. URL https://developer.disruptive-technologies.com/api/libraries/python/client/examples/plot_sensor_data.html. [Accessed: 2022-05-07].

[64] Paolo Salvatori. Service bus explorer. *GitHub*, 2022. URL https://github.com/paolosalvatori/ServiceBusExplorer. [Accessed: 2022-05-07].

[65] Disruptive Technology AS. Security and privacy in disruptive technologies sensing solution. 2022. URL https://www.disruptive-technologies.com/blog/security-and-privacy-in-disruptive-technologies-sensing-solution. [Accessed: 2022-05-22].

[66] Patrick Nohe. Executing a man-in-the-middle attack in just 15 minutes. 2021. URL https://www.thesslstore.com/blog/man-in-the-middle-attack-2/. [Accessed: 2022-05-13].

[67] Disruptive Technologies. How end-to-end sensor architecture addresses iot security weak points. 2020. URL https://www.disruptive-technologies.com/blog/how-end-to-end-sensor-architecture-addresses-iot-security-weak-points. [Accessed: 2022-05-31].

[68] Annabelle Lee, Miles Smid, and Stanley Snouffer. Security requirements for cryptographic modules [includes change notices as of 12/3/2002], 2001-05-25 2001. [Accessed: 2022-05-14].

[69] Praetorian Samsung. Praetorian - samsung - artik modules engagement letter. 2022. URL http://customer1st.com/wp-content/uploads/2016/10/Praetorian-Samsung-ARTIK-Modules-Engagement-Letter.pdf. [Accessed: 2022-05-22].

[70] Disruptive Technologies. Managing access rights. 2021. URL https://developer.disruptive-technologies.com/docs/service-accounts/managing-access-rights. [Accessed: 2022-05-09].

[71] Postman Inc. What is postman? *Postman*, 2022. URL https://www.postman.com/product/what-is-postman/. [Accessed: 2022-05-13].

[72] SSL.com Support Team. Oauth 2.0. 2022. URL https://oauth.net/2/. [Accessed: 2022-06-02].

[73] Atiko. What is the range of data transmission in the lorawan network in an urban environment? 2022. URL https://www.atiko.com.ua/en/blog/what-is-the-range-of-data-transmission-in-the-lorawan-network-in-an-urban-enviro
[Accessed: 2022-05-23].

[74] Jeff Shepard. Battery life analysis and maximization for wireless iot sensor nodes and wearables. *Power Electronic Tips*, 2020. URL https://www.powerelectronictips.com/battery-life-analysis-and-maximization-for-wireless-iot-sensor-nodes-and-wearables/. [Accessed: 2022-06-01].

[75] Bob Card. Achieving a 10-year battery life with bluetooth low energy and proprietary wireless protocols. *Onsemi*, 2020. URL https://www.onsemi.com/company/news-media/blog/iot/bluetooth-low-energy-wireless-protocols-battery-life. [Accessed: 2022-06-01].

[76] Evan Cornell. Achieve extremely long battery life in wireless sensor nodes. *Texas Instruments*, 2015. URL https://e2e.ti.com/blogs_/b/industrial_strength/posts/achieve-extremely-long-battery-life-in-wireless-sensor-nodes. [Accessed: 2022-06-01].

[77] Disruptive Technology AS. Wireless temperature sensor. 2022. URL https://www.disruptive-technologies.com/products/wireless-sensors/wireless-temperature-sensor. [Accessed: 2022-05-22].

[78] Docker. Docker overview. *Docker Docs*, 2022. URL https://docs.docker.com/get-started/overview/. [Accessed: 2022-06-01].

[79] Pająk. P. Quick dive into azure service bus. *Iteo*, 2020. URL https://iteo.com/blog/post/quick-dive-into-azure-service-bus/. [Accessed: 2022-05-09].

[80] Azure. Advanced message queueing protocol(amqp) 1.0 support in service bus. *Azure*, 2021. URL https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-amqp-overview. [Accessed: 2022-05-27].

[81] Azure. Storage account overview. *Azure*, 2022. URL https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview?toc=%2Fazure%2Fstorage%2Fblobs%2Ftoc.json. [Accessed: 2022-05-22].

[82] ALBERTO LUGO. How secure is microsoft azure? 2017. URL https://invidgroup.com/how-secure-is-microsoft-azure/. [Accessed: 2022-05-22].

[83] Microsoft. An overview of azure sql database and sql managed instance security capabilities. 2022. URL https://docs.microsoft.com/en-us/azure/azure-sql/database/security-overview?view=azuresql. [Accessed: 2022-05-22].

[84] Azure. Azure storage encryption for data at rest. *Azure*, 2021. URL https://docs.microsoft.com/en-us/azure/storage/common/

storage-service-encryption?toc=%2Fazure%2Fstorage%2Fblobs%2Ftoc.json. [Accessed: 2022-05-22].

[85] Microsoft Azure. Pricing calculator. 2022. URL https://azure.microsoft.com/en-us/pricing/calculator/?service=service-bus. [Accessed: 2022-05-22].

[86] IoT business news. The precision agriculture market to reach € 3.7 billion worldwide in 2025. 2021. URL https://iotbusinessnews.com/2021/03/11/47844-the-precision-agriculture-market-to-reach-e-3-7-billion-worldwide-in-2025/ [Accessed: 2022-05-27].

[87] Tibco. Message broker illustration. 2022. URL https://www.tibco.com/reference-center/what-is-a-message-broker. [Accessed: 2022-05-02].

[88] Image link. 2022. URL https://www.readingielts.com/ielts-speaking-part-1-topic-applications/. [Accessed: 2022-05-27].

[89] Xpandretail. Count illustration. 2022. URL https://xpandretail.com/logic-for-people-counting/. [Accessed: 2022-05-22].