2018•2019
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Developing a generic IoT solution to prepare production lines for Industry 4.0

PROMOTOR :
Prof. dr. ir. Ronald THOELEN

PROMOTOR :
Dhr. Ruan HOLSTEYNS

BEGELEIDER :
Dhr. Bram PIETERS

Chiel Dommange, Pieter Polmans

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

2018•2019
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Developing a generic IoT solution to prepare production lines for Industry 4.0

PROMOTOR :
Prof. dr. ir. Ronald THOELEN

PROMOTOR :
Dhr. Ruan HOLSTEYNS

BEGELEIDER :
Dhr. Bram PIETERS

Chiel Dommange, Pieter Polmans

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Preface

After we struggled with our respective theses, we decided to start a new thesis when realizing a cooperation between students from campus Group T and campus UHasselt was possible.With newfound vigor we went looking for interesting subject to write about. After finding a multitude of potential subjects we decided to work on this topic. This thesis has not only granted us new insights, but also helped us grow professionally as well as personally.

As expected, we encountered some setbacks along the way, some of which were minor and some of which were more severe. But, in the end, failure is always an opportunity to learn.

We would like to thank our supervisors: Bram Pieters, Prof. Dr. Ir. Ronald Thoelen and Ruan Holsteyns for their insights and constructive feedback. Additionally, we would like to thank our families, friends and loved ones, without whose unending support and patience we would not have the persistence and confidence to finish this dissertation. Also, we would like to thank the entirety of Maeccenas group for providing a relaxed and supportive work environment.

And finally, we would like to thank you, the reader, for taking the time to read this dissertation; We sincerely hope our work will spark your interest and grant you new insights.

Contents

List of Figures

Glossary

Apache Kafka: A robust and scalable messaging system.

API: Short for Application Programming Interface: a set of functions and procedures allowing the creation of applications that access the features or data of another service.

Chronograf: A data visualisation tool.

Consumer application: An application that implements the Apache Kafka Consumer API.

ERP: Enterprise Resource Planning.

Free and open source software: Commonly abbreviated to FOSS. Free and open source software is software that is free to use and of which the source code is available to anyone interested.

Grafana: A data visualisation tool.

Industry 4.0: The fourth industrial revolution.

InfluxDB: A TSDB which can be queried using InfluxQL.

InfluxQL: Querying language used by InfluxDB.

IoT: Internet of Things: A system of interconnected devices consisting of mechanical and digital devices.

JSON: JavaScript Object Notation.

MAC-Address: Media Access Control Address: Unique identifier of a machine in a network.

Persistent storage: Storing data such that it is retained over reboots of the program or system.

Producer application: An application that implements the Apache Kafka Producer API.

Querying language: A language created for accessing, modifying and inserting data in a database.

Raspberry Pi: A small single-board computer.

Record: A piece of data stored in a topic in Apache Kafka.

Sensor node: Any piece of hardware that gathers aggregates the data of one or more sensors.

Server Pi: A Raspberry Pi that functions as a server.

SLA: Service Level Agreement: An agreement between a service provider and client as a guarantee for quality of service.

Streams application: An application that implements the Apache Kafka Streams API.

Time series data: A series of data in chronological order.

Topic: A collection of records in Apache Kafka.

TSDB: Time Series DataBase: A database optimised for storing time series data.

Unix nanosecond timestamp: The number of nanoseconds that have passed since 1 January 1970.

Victhorious platform: A cloud hosting platform.

Zookeeper: A centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. (https://zookeeper.apache.org)

Abstract

This project revolves around developing a generic *IoT* solution to prepare for *Industry 4.0*. The primary objective was to achieve the first two steps of the Acatech Industry 4.0 maturity study development track[1]. Should this objective be reached, the secondary objectives were to implement as many Industry 4.0 development track stages as possible, in order.

The project was realized by first creating an introductory proof of concept, and later implementing a stack of applications on a cloud platform, based on the insights gained. The application stack runs on a virtualized container of Ubuntu Linux hosted on the *Victhorious platform* [2]. In order to attain the desired functionality, *Apache Kafka* was used to create a robust messaging system capable of aggregating sensor data [3]. *InfluxDB* was used as a scalable way of storing IoT data aggregated by Apache Kafka [4]. Finally, the data was visualized by using *Grafana* [5]. Although these applications could be hosted on distributed platforms, they were developed on the same platform for this project in order to minimalize resource costs.

The primary objective was reached as well as the visualization stage of the Industry 4.0 development track. The transparency, predictive capability and adaptability stage can be built based on this research in further development.

Dutch Abstract

Dit project draait rond het ontwikkelen van een generieke IoT oplossing om op Industry 4.0 voor te bereiden. Het hoofddoel was heb bereiken van de eerste twee stappen van de Acatech Industry 4.0 maturity study development track [1]. Indien dit doel behaald wordt, zijn de secundaire doelen om zo veel mogelijk Industry 4.0 stages in volgorde te implementeren.

Het project werd gerealiseerd door eerst een inleidend *proof of concept* te maken en later een *application stack* op een *cloud platform* gebaseerd is op de behaalde inzichten. De application stack draait op een gevirtualiseerde Ubuntu Linux container die *gehost* wordt op het Victhorious platform [2]. Om de beoogde functionaliteit te behalen werd Apache Kafka gebruikt om een robuust *messaging system* te maken dat in staat is om sensor data te aggregeren [3]. InlfuxDB werd gebruikt als een schaalbare manier om IoT data, door Apache Kafka geaggregeerd, op te slaan [4]. Ten slotte werd de data gevisualiseerd met Grafana [5]. Hoewel deze applicaties ook op een gedistribueerd platform gehost kan worden, werden ze op hetzelfde platform ontwikkeld om de kosten te drukken.

Het hoofddoel was behaald samen met de *visualisation* stage van de Industry 4.0 development track. De *transparency, predictive capability* en *adaptability* stage kunnen verder ontwikkeld worden gebaseerd op dit onderzoek.

1. Introduction

1.1 Motivation for thesis

"Industry 4.0", as defined by Dr. Heiner Lasi in 2014, is a term used to describe the fourth Industrial revolution *ex ante*. Dr. Lasi envisioned a paradigm shift within the manufacturing industry as a result of recent developments in computing technology. This revolution will be characterized by a strong technology push pertaining automation, digitalization, Internet technologies and computer miniaturization. Predicted developments are: Smart Factories, Cyber-Physical systems, self-organized manufacturing- and distribution systems and individualized procurement[1] [2].

According to the Acatech Industry 4.0 maturity study, the Industry 4.0 development path is described in 6 stages. The first two stages, which are considered digitalisation instead of part of the Industry 4.0 development path itself, are computerisation and connectivity. Although Industry 4.0 is predicted to rely heavily on these technologies and their recent developments, the Acatech maturity study regards digitalisation as a preliminary requirement for Industry 4.0 [1].

Industry facilities already saw an introduction of computerization to some of their machines after the third Industrial revolution[2] in the form of Programmable Logic Controllers (PLCs), which is why this stage is not considered an Industry 4.0 capability stage. Conversely, network connectivity is a relatively new tool with regards to the latest Industrial revolution. But seeing as a network connectivity doesn't provide additional features by itself but rather provides a platform for other concepts to do so, this step too is regarded as a preliminary requirement.

The last 4 steps in the development path pertain to more concrete Industry 4.0 applications. These final stages are respectively called visibility, transparency, predictive capacity and adaptability. Each of these stages provide a company with a set of useful tools as well as a cumulatively growing capability to react in a more agile manner to events that disturb facility workflow. Throughout this thesis, the stages described by the Acatech maturity study will serve as a guiding principle for the objectives.

The final stage of the maturity model can potentially provide the exceptional capability to predict malfunctions and avoid them before they pose a significant problem. This stage would be especially beneficial, since preventing defects by proactively maintaining equipment would be far less costly in both time and resources compared to doing so reactively.

Without proactive maintenance, facility managers are often unaware of impending malfunctions, which can significantly slow down or even completely halt production workflow when they arise. Scheduled maintenance may alleviate this problem, but will interrupt facility workflow on a regular basis. These scheduled interruptions are sometimes not required and thus unnecessary. Being able to provide technical personnel with detailed information of current or predicted defects also allows for a more efficient approach to problem solving.

However, the Acatech study also mentions a common reluctancy in adopting Industry 4.0. This is especially true for Belgium [3], where some manufactories are equipped with older machines that lack digitalization

[1] also known as 'batch size one'

[2] The third Industrial Revolution is stated to have started in 1969

capabilities due to a lack of modern sensors but are otherwise perfectly functional and thus scheduled to remain in use for several more years.

Other facilities do possess modernized machines capable of detecting a suite of sensor values but oftentimes their data is only available in a closed or proprietary protocol. Furthermore, even when sensor data is available, this is not commonly integrated into facility workflow by medium of an IoT system, so the information remains in a closed circuit within the machine. In conclusion, before a facility is ready to adopt Industry 4.0, the computerization and network connectivity stages both need to be addressed.

Without a universal IoT standard, becoming Industry 4.0 ready proves to be a daunting task for companies. The lack of standardization causes a lengthy and arduous conversion process, which is usually paired with high costs. Due to these high costs, most manufactories deem the investment too steep compared to the prospective benefits.

The main goal of this thesis is to develop a generic IoT solution to prepare for Industry 4.0. This preparation should help lower the cost of entry to industry 4.0 by providing a generified platform on which the Industry 4.0 stages from the Acatech maturity study can be built. Building the Industry 4.0 capability stages is outside of the scope of this thesis, but the researchers have elected to attempt to develop as many of these stages as the given timeframe permits.

2. Background

2.1 What is Industry 4.0

When discussing prospective developments in the manufacturing industry, Industry 4.0 quickly comes to mind. Topics frequently related to Industry 4.0 are: smart factories, factory of things, agent-based factories and cyber-physical systems. According to Dr. Lasi et al., Industry 4.0 is characterized first and foremost by a strong application-pull and technology-push [2].

2.1.1 Technology-push

The technology-push pertains to recent and future technological advancements that significantly impact human activities. Specifically, some related technologies are miniaturized computing, interconnectivity and rapid prototyping.

Miniaturized computing is a result of the constant reduction in size of semiconductors and their applications in accordance to Moore's Law [4]. This continuous size reduction in semiconductor technology resulted in the emergence of technologies such as smartphones, performant microcontrollers, etc.

Since the dawn of the 21st century remarkable advancements have also been made in communications technologies. Notable advances were: the introduction of Bluetooth, LoRaWAN, 5Ghz WiFi and mobile cellular internet, among others. These new internet technologies enable an unprecedented interconnectivity of digital systems, providing the basis for new topologies like the Internet of Things.

Innovations in the fabrication technology sector also serve as a strong technological push for Industry 4.0. For example, computer-aided design and machinery (CAD/CAM) and additive manufacturing have drastically reduced the time from design to prototype. As a matter of fact, 3D-printing technologies have become so effective they are being considered as a replacement for conventional production techniques in some instances [5].

2.1.2 Application-pull

The introduction of industry 4.0 is also believed to be a result of a strong application-pull. This means that not only will Industry 4.0 come forth from new technological opportunities, but also from societal, economic and political changes. These changes will force the manufacturing industry to adopt a new attitude towards the production process. According to Dr. Lasi et al., this application-pull would mainly revolve around a number of concepts: "time to market", "batch size one", flexibility, decentralization and sustainability.

An ever-growing success factor for enterprises is reducing their products' time to market to get and stay ahead of their competition. Aside from a business standpoint, this evolution also allows manufacturers to provide their customers with their latest innovations as quick as possible.

Another factor for staying competitive is to be able to respond to customers' increasingly individualized demands in a flexible and economical manner. For this reason, value production chains should be designed so that they can rapidly produce batches as small as one unit from varying products as opposed to large quantities of a single product.

In order to streamline the decision-making process, it should be decentralized to reduce the latency introduced by conventional hierarchies. Paired with a focus on human decision-making in near-fully automated facilities, this paradigm shift should ensure more agile production facilities.

Finally, as the finite nature of the earth's resources becomes ever more apparent, the demand for sustainable production techniques increases constantly. Evolutions in juridical environments as well as a societal push for ecological awareness demand a more resource-efficient approach. Because of these influences, enterprises must adapt their manufactories for sustainability.

2.2 What is Apache Kafka

Apache Kafka is a distributed streaming platform that can be used as a message broker. It can be seen as a hybrid between the message queuing and publish-subscribe models that are used in other messaging systems.

In a queuing messaging system, messages are sequentially added to a queue. Consumers in a pool of subscribers then each take a message from the queue and handle that message. The main issue with queuing messaging systems is that once a consumer has read a *record* from the queue, another consumer can no longer get that message from the queue.

In a publish-subscribe messaging system, every message is sent to every subscriber. The difficulty of using a publish-subscribe system lies in its scalability. Because every message gets sent to every subscriber such a system typically doesn't scale well vertically.

Apache Kafka adopts concepts from both of these models; hence it is considered a hybrid system. It relies on the concept of consumer groups. In a consumer group the processing of the messages can be distributed among the consumers in that group. This works like a queuing system. Similarly, Apache Kafka allows broadcasting to multiple consumer groups akin to publish-subscribe system[3].

Apache Kafka provides four core *API*s that allow applications to publish and subscribe to streams of records that are called records. These four APIs are the Producer API, the Consumer API, the Streams API and the Connector API.

The Producer API is used by applications to publish a stream of records to one or multiple *topics*. In our study this API was used by our *sensor node*s as well as the other applications that gathered data from various sources[4].

The Consumer API is used by applications to subscribe to one or more topics and receive a stream of records from them. These applications can then process these streams of records. Consumers can be used to write data to an external database for example.

The Streams API subscribes to one or more topics and publishes to one or more topics. Applications implementing the Streams API consume the streams they receive from the topics they are subscribed to, perform transformations and then produce a new and transformed stream that is sent to the relevant output topics.

The relation between the aforementioned core APIs and Kafka topics is illustrated in figure 1.

The Connector API is used to build reusable consumers and producers to connect Apache Kafka to existing external applications. The connector API was not used in this research. [6]

[3] For a more detailed explanation, refer to the Apache Kafka documentation and introduction page.

[4] These other applications are described in a later section

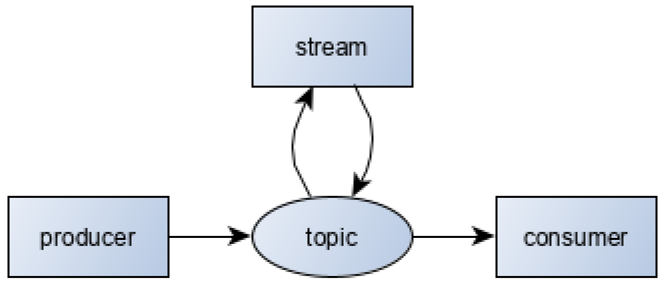

figure 1: Simplified Apache Kafka messaging Topology

The applications in which the Producer, Consumer and Streams API were used are described in 4.2.3 Behaviour.

2.2.1 Motivation for using Apache Kafka

In this research, the end result should be able to handle a vast amount of data, collected by a range of sensor nodes. These sensor nodes would collect a diversity of data types at various intervals. Because of this, a message broker was necessary to coordinate a correct data flow.

The messaging platform had to be horizontally and vertically scalable to handle a large volume of data from a heterogenous pool of data sources. Additionally, the messaging system had to be highly performant to provide near real-time data availability for a vast number of data sources.

At the recommendation of their mentor, the researchers have considered Apache Kafka for this purpose, among other messaging systems like RabbitMQ. Ultimately, Apache Kafka was deemed as the best fit for this project.

2.3 What is InfluxDB

In IoT solutions, *time series databases (TSDB)* are widely used to store and represent continuous data. These databases, optimized for collecting and storing *time series data*, are critical to analyse data trends. The Analysis of these data trends may vary from a simple graph to multi-feature machine learning algorithms which can then later be used for prediction models.

Time series data is any data that is collected over time and for which the time at which the data is collected is stored along with the data. The application of time series databases for data analysis and decision-making is not a new concept either, take for example the financial trends in typical stock market buildings, medical monitoring or any other graph which represents continuous data over a given time frame.

As Illustrated in figure 2, an instance of InfluxDB can actually consist of multiple databases. Each of these databases holds one or more tables called measurements. Measurements are a collection of points, which serve as data records. [7]

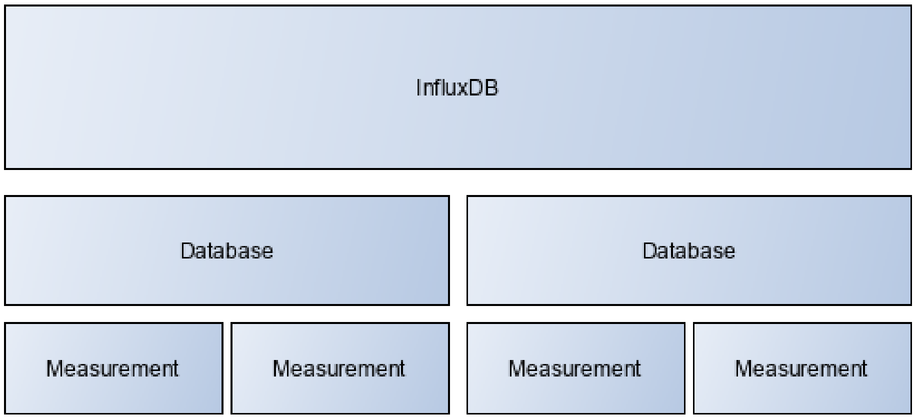

figure 2: InfluxDB structure

A point always has a timestamp, and can have multiple tags and fields. In InfluxDB, the timestamp is always saved as a *UNIX nanosecond timestamp*. Tags are key-value records where both key and value are strings. Much like the tags, fields are also key-value records with the exception that field values can be floats, integers or boolean values as well as strings.

As is customary for database software, InfluxDB can be queried to extract information in a structured manner. For this purpose, the Influx query language (*InfluxQL*) is provided. This *querying language* features SQL-like syntax, with a focus on time-series data. This syntax can then further be used by other programs that wish to exploit the data stored within InfluxDB.

In terms of querying, tags are optimized to filter data on. A good example of a tag would be the *MAC-address* of a machine. Using that tag, a query can filter data in order to only return data relevant to that specific machine. Another example of a useful tag would be the type of a sensor node. Using this tag, data of all sensor nodes from this type can be compared while ignoring data from other sensor nodes from a different type which are performing comparable measurements.

2.3.1 Motivation for choosing InfluxDB

There are multiple time series databases available, respectively: InfluxDB, Graphite, Prometheus, Amazon Timestream among others. Since 2016, InfluxDB has been the market leader with regards to time-series databases, as is illustrated by figure 3. This fact was the first motivation for choosing InfluxDB as a TSDB for this research [7].

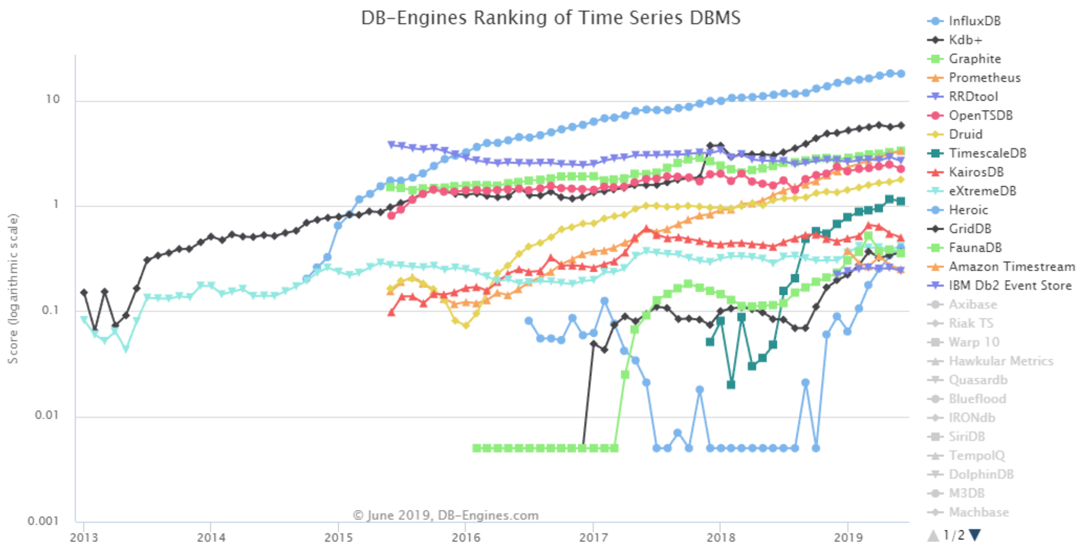

figure 3: popularity ranking of TSDBs, image courtesy of DB-engines.com

Additionally, InfluxDB is published under the open-source MIT license. This license allows anyone to freely use the software[5], as well as guarantees software transparency. Since the researchers ascribe a high value to these concepts, the licensing was a strong motivator for using InfluxDB.

Finally, InfluxDB instances can be installed locally, which eliminates the dependency on third parties for the operation of the system and permits full control to the researchers. This independence, combined with the previously discussed motivators, convinced the researchers to use InfluxDB for this project.

2.4 What is the Victhorious platform

The Victhorious platform is a cloud hosting platform developed and provided by DeltaBlue, a sister company of Guardway. It is promoted as a cross-provider cloud hosting platform that can be used to develop applications which encompass multiple technologies. [9]

The Victhorious platform allows developers to build their applications and then deploy them in a cloud server hosted by Microsoft Azure, Google, Amazon and others worldwide. It features a rich user interface as well as support for application configuration and deployment automation. The Victhorious platform provides hosting for virtualized containers of many technologies, one of which is a virtualized instance of the Ubuntu Linux operating system.

[5] with limited liability and warranty

2.4.1 Motivation for using the Victhorious platform

As is discussed by the Acatech maturity study, connectivity is a crucial aspect of Industry 4.0. Additionally, all stages on the Industry 4.0 development path rely on digital technology. Cloud services provide a myriad of benefits for these purposes.

Cloud computing resources provide computing resources at a low capital expense, are rapidly scalable and have *Service Level Agreements (SLAs)* wherein a certain quality of service is guaranteed. These benefits are the reason why the researchers decided to employ a cloud platform for this project.

Specifically, the Victhorious platform is a central building block for the system that is developed. The primary reason for using the Victhorious platform for this project is that it provides access to a feature rich cloud platform at no cost due to it being hosted in-house rather than by an external party.

2.5 What is Grafana

Grafana is a tool for visualising, monitoring and analysing data. It allows users to easily create dashboards in which data can be visualised in a variety of ways. These dashboards provide a high customisability, allowing users to tailor them to their needs. Grafana has built-in support to connect with a number of databases such as Graphite, MySQL, and most important for this research, InfluxDB. [10]

By writing queries in InfluxQL, data can be extracted from an InfluxDB database that can then be visualised by a chart on a Grafana dashboard. Additionally, Grafana features an intuitive user interface with which basic queries can be quickly built, based on the data stored in the connected InfluxDB database. For more advanced queries, it also provides users with a direct query editor.

2.5.1 Motivation for choosing Grafana

When analysing timestamped data, having access to a visual representation of that data is especially beneficial. Data can be visualised in a number of different ways. One option is to write a simple web application that reads out the time series data and then uses one of the numerous charting libraries available. Another option is to use a tool that is made specifically for this purpose.

The researchers decided in favour of a pre-existing tool over a self-written web application. The reason for this decision is that these tools have already been developed by dedicated teams, which means their features are more mature and complete than an application created within the timeframe of this project for the sole purpose of this research.

The first tool that was considered was *Chronograf*, as it is developed by the same company that develops InfluxDB which guarantees interoperability. Additionally, Chronograf is advertised as part of the TICK stack which consists of the following tools: Telegraf, InfluxDB, Chronograf and Kapacitor [11].

Both Grafana and Chronograf support connection with InfluxDB out of the box, are *free and open source*, and can be run locally on the Victhorious platform. They both provide comparable features such as real-time data visualization, alerting on user-defined events and dashboards to quickly display data overviews.

Ultimately, Grafana was chosen over Chronograf for various reasons. Objectively, Grafana has support for targeted alerting conditions, which Chronograf lacks. Subjectively, the authors found Grafana easier to use, more intuitive and aesthetically pleasing.

3. Objectives

3.1 Primary objective

The aim of this thesis is to develop a generic solution to help companies prepare themselves for Industry 4.0. The primary objective for this thesis is to fulfil the preparing stages for the Industry 4.0 development track, which is illustrated in figure 4. This means that the desired outcome is a system that can aggregate sensor data and subsequently transmit it to a central platform, which in turn allows other applications to use aforementioned data. Possible uses of this data are workspace monitoring, enabling proactive maintenance, creating automated workflows and even predicting optimal production. [1]

This solution must be designed as such that it is modular and scalable. For horizontal scalability, that means the software that is developed must be generified and reusable. When this is done correctly, the implementation time of the physical components is minimized and the business logic that manages the production systems can be distributed and thus decentralized.

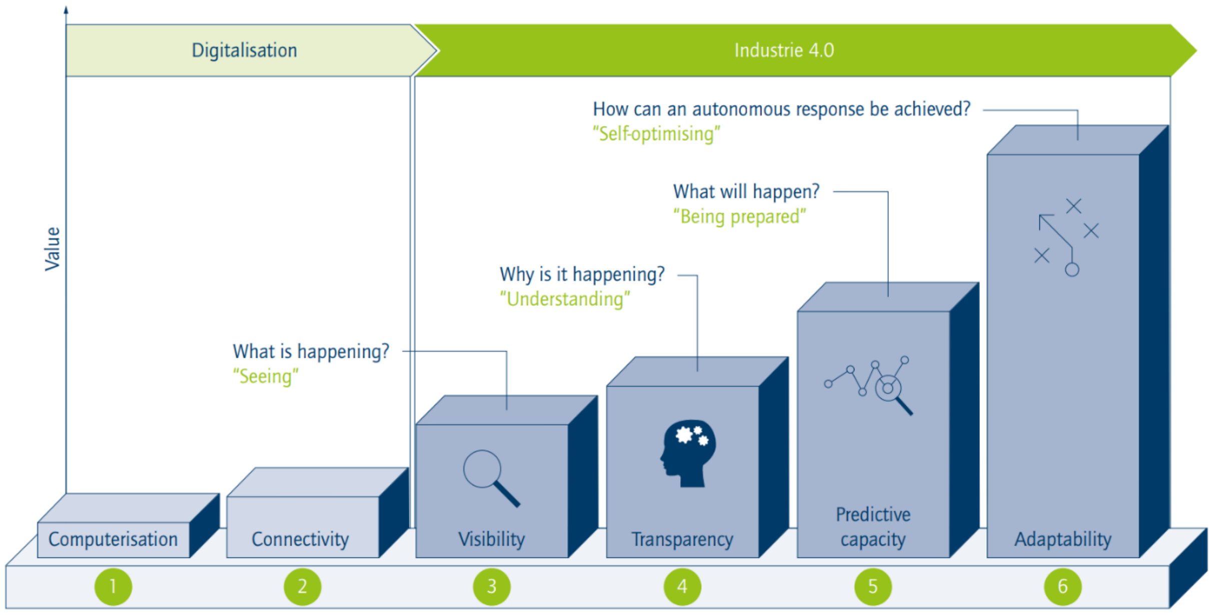

figure 4: The Industry 4.0 development track

3.2 Secondary objectives

The secondary objectives for this project are to improve the system developed for the primary objective so it does not only provide the preparation for Industry 4.0, but also serves as an exploratory introduction to Industry 4.0 functionality. The Industry 4.0 development track dictates the stages to attain full Industry 4.0 capabilities are, in order: visibility, transparency, predictive capacity and adaptability.

Fulfilling all of these stages is outside of the scope of this project, as the researchers are aware that doing so is a too ambitious challenge within the given timeframe. Within the scope of this project is to achieve the visibility stage, and potentially extend to the stages regarding transparency and predictive capacity if possible.

Concretely, the visibility stage refers to being able to visualize the data into graphs and other visual representations. The Transparency stage is achieved by integrating data into workflow engines or other business applications such as *ERP* systems which can perform automated data analysis. Predictive capability relies on machine learning algorithms to analyse the workflow of a production process and predict upcoming events within it.

4. Methods & Materials

4.1 First proof of concept

4.1.1 Objective

The objective of the first proof of concept was to prove that it was possible to make a simple data aggregator that could both send data to a central server as well as act on data that it received from that server.

4.1.2 Setup

The first proof of concept was made using two *Raspberry Pi*'s[6] that acted as sensor nodes and a third Raspberry Pi[7] that acted as a server, henceforth referred to as the *server Pi*. The relation between these Raspberry Pi's is visualized in figure 5. The two sensor nodes were both equipped with a Sense HAT[8] add-on board[9], providing them with the following sensors: Gyroscope, accelerometer, magnetometer, temperature sensor, barometric pressure sensor and a humidity sensor. The Sense HAT also provided an 8x8 RGB LED matrix. figure 6 shows such a sensor node including the sense HAT.

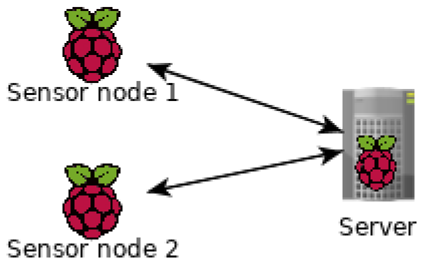

figure 5: Proof of Concept architecture

These sensor nodes were installed in our office, one was placed near the window (figure 8) and one near the HVAC unit on the ceiling (figure 7). These locations were chosen primarily because of their difference in temperature, with humidity and pressure being secondary. To test this proof of concept a hairdryer was used to influence the temperature of either one of the two sensor nodes.

[6] Raspberry Pi Model 3B+
[7] Model 3B+ as well
[8] HAT: Hardware Attached on Top
[9] https://www.raspberrypi.org/products/sense-hat/

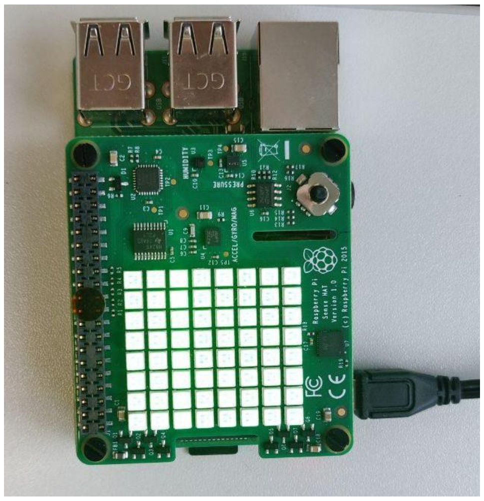
figure 6: Raspberry Pi sensor node with sense HAT

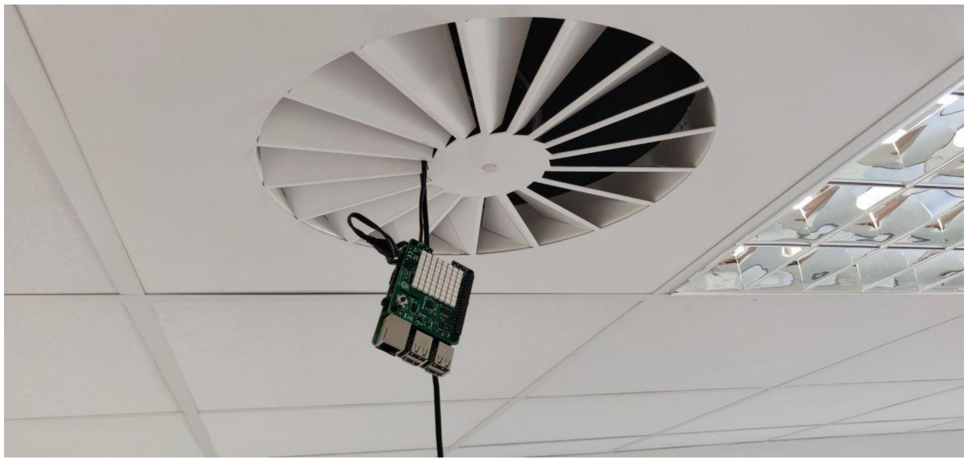
figure 7: Sensor node safely installed near the HVAC unit in the ceiling

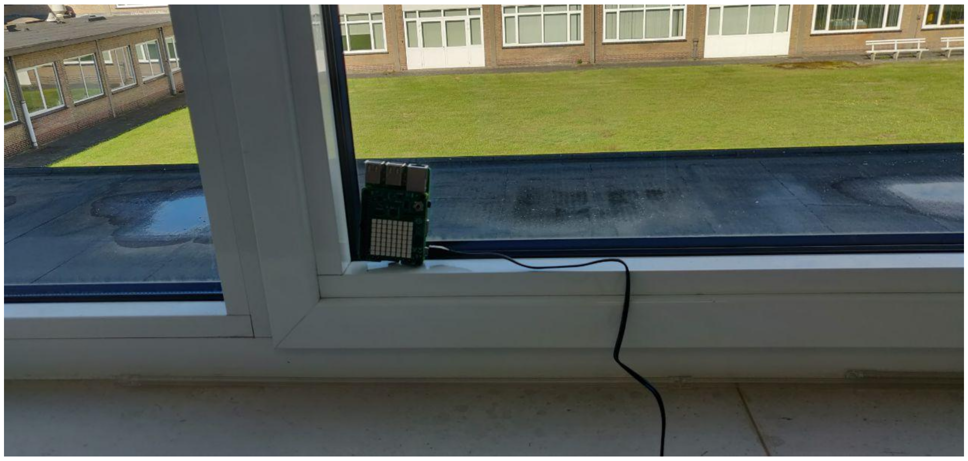
figure 8: Raspberry Pi sensor node located near window

4.1.3 Behaviour

The two sensor nodes were programmed to read data from their temperature, pressure and humidity sensors and forward it to the server Pi. The server Pi would then determine if the received values were within comfort range[10].

The server would then send a response back to the sensor node. The response depended on multiple factors. The first factor was whether the measured values were within the comfortable range. If the measured values were not within the comfortable range, the response depended on a penalty. The penalty would increase based on how far the measurements were outside of the comfort range, and weighted based on their respective properties. For example, when the pressure would significantly drop below 1 atm, the penalty would be more severe than if the air was more humid than the comfort range permitted.

Finally, the sensor node would then light up their LED matrix in a specific colour based on the received penalty. For a penalty of zero, the LED matrix would be coloured green. When the penalty indicated the values to be just outside of the comfort range, the LED matrix would turn yellow. For extremely uncomfortable circumstances, indicated by a high penalty value, the indicator would emit a red light.

The server Pi read the comfortable values from a provided configuration file. Such a file could be configured on the server, which dictates the business logic of this application. An example snippet of such a configuration file is included in Appendix A: Consolidated configuration file for the first Proof of Concept.

The server Pi starts by opening a socket and listening to it. Every time a new connection is made, the server Pi adds the new sensor node to a list of sensor nodes. It then starts a consumer thread and a producer thread. The two threads make use of that list of devices.

The consumer thread handles data that is received by the server Pi. In this proof of concept, the only data that is received consists of four values: A character indicating the type of message that is sent, a value for temperature, one for pressure and one for humidity. If the received message is regular data, the consumer stores this together with the sensor node in the original list. If the received message is a stop message, the sensor node that sent the message is unregistered and removed from the list.

At the same time, the producer thread handles data that has to be sent to the sensor nodes. Meanwhile, the producer thread calculates a penalty value using the stored sensor values. The penalty value is an indication of how comfortable the room in which the sensor node is placed is and is used by the sensor nodes to determine the colour their LED matrices should display. The calculated penalty value is compared to the previous penalty value, if there is one. If the penalty value has changed or if there was no previous penalty value, it will be sent to the corresponding sensor node. If there was no change, nothing will be sent.

The threads will keep on repeating their actions for every sensor node in the list. Afterwards they sleep for a few seconds before repeating their operations again. The threads keep looping until the program is interrupted. When the program is interrupted the server closes the socket which breaks the connection with all sensor nodes. Finally, the two threads are stopped, ending the program. This behaviour is also described in figure 9.

[10] Comfortable ranges were arbitrarily chosen

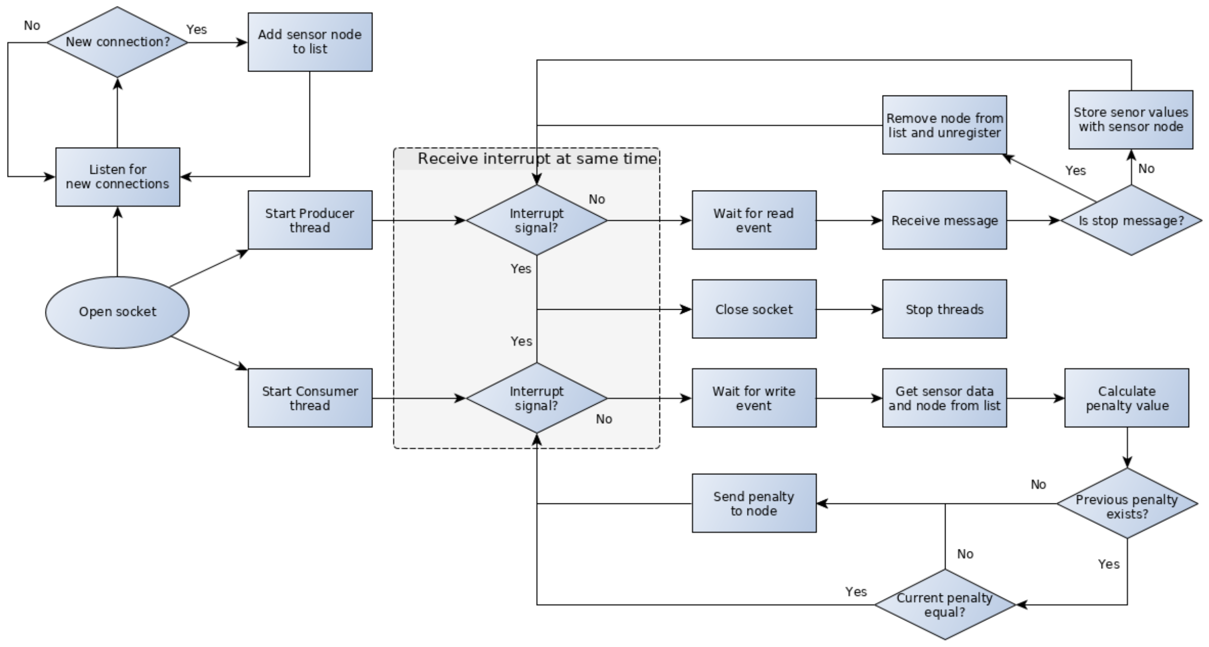

figure 9: Flowchart depicting server Pi logic

Sensor nodes connect to the server Pi via the socket that is opened. Once a sensor node has made a connection it will start two threads: a consumer and a producer thread.

The producer thread will read out sensor data and send it to the server Pi on regular intervals.

The consumer thread will listen for incoming messages from the server Pi that tell it what penalty value the server Pi calculated based on the sensor data that was sent. It will then change the colour of the LED matrix accordingly. This behaviour is also described in figure 10.

Both threads will keep on running until an interrupt signal is received[11]. When an interrupt signal is received, the sensor node will notify the server Pi that is has stopped by sending a special message containing a stop code. It will then close the connection and stop both threads.

[11] Interrupt signal is sent by pressing Ctrl-C in the terminal window the program is running

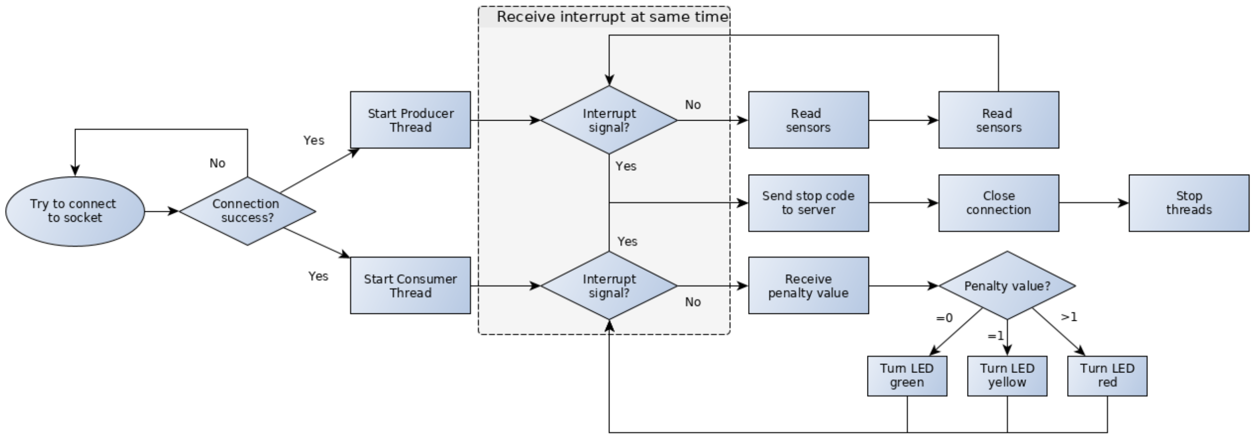

figure 10: Flowchart depicting sensor node logic

4.1.4 Problems encountered

In order to make a connection the sensor nodes had to be connected to the same network as the server Pi. This was a very limiting factor and was remediated by using the Victhorious platform instead of a raspberry Pi acting as server.

Additionally, the way in which the server Pi's logic was implemented was not very scalable. The server Pi was only able to act on messages that followed a very specific pattern and could also only respond to received messages in a very primitive way: by simply sending a single value back to the sensor node the message originated from. The next section describes how this problem is solved by using Apache Kafka as a messaging broker.

4.2 Apache Kafka and the Victhorious platform

4.2.1 Objective

A major shortcoming of the first proof of concept was a lack of horizontal and vertical scalability. Messages from the two sensor nodes were processed one after the other, even though the relative order between the messages of the sensor nodes was irrelevant. Hypothetically, adding a significantly large number of identical sensor nodes (vertically scaling) would cause the server Pi's logic to slow down significantly to the point it would no longer be able to keep up with the stream of messages directed towards it. Additionally, adding new, different types of sensor nodes equipped with different types of sensors (horizontally scaling) would require that a significant portion of the code for the server Pi be rewritten.

Another major shortcoming was that sensor nodes that want to communicate to the server Pi had to be on the same network as the server Pi. This means that if data has to be collected across a large company, the entire company would have to be on the same network. This would either place a restriction on the network infrastructure of the company as well as severely limit the scope in which data can be collected.

The primary goals of this phase of the research are to resolve these scalability and connectivity issues. In the scope of scalability, this means the computing power of the server needs to be improved to the point a raspberry Pi does no longer suffice. With regards to this, the researchers have decided to move the server logic to the Victhorious platform. Furthermore, Apache Kafka will be used to tackle both the scalability and connectivity shortcomings.

4.2.2 Setup

The Victhorious platform is used to replace the role of the server Pi of the first proof of concept. It serves as the central device to which sensor nodes and other data producers send their messages.

Apache Kafka runs on the Victhorious platform and is managed by a *Zookeeper* instance. Zookeeper is a prerequisite for Apache Kafka servers, since Apache Kafka relies on it to synchronize the cluster nodes and keep track of configurations, partitions, topics etc. (figure 11). Supervisor[12] is used to ensure that Apache Kafka and Zookeeper are started when the platform reboots.

The sensor nodes used here are the same as those used in the first proof of concept and are still placed in the same location. The sole modification done to them is that they have been extended with client software to connect to the Apache Kafka instance that is running on the platform instead of to the server Pi.

On the platform multiple applications will be run that implement three of the four Apache Kafka core APIs mentioned in the section 'What is Apache Kafka?'.

The three core APIs that were used are the Producer API, the Consumer API and the Streams API.

[12] http://supervisord.org

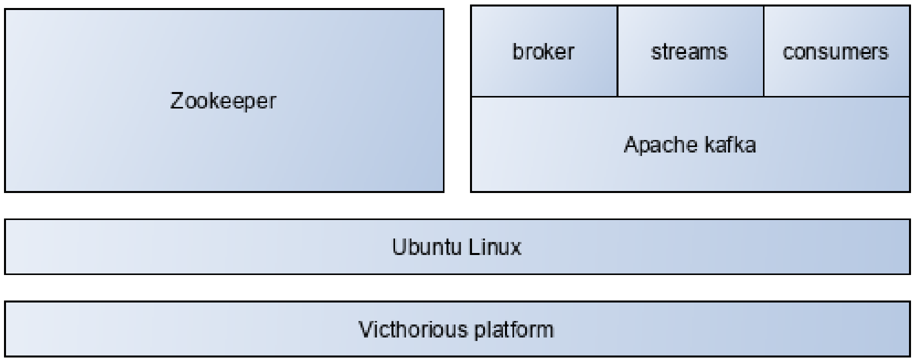

figure 11: Stack architecture with Apache Kafka and Zookeeper:

4.2.3 Behaviour

Apache Kafka functions by using applications that make use of the Producer API to send a stream of messages, also called records in this context, to an Apache Kafka instance. Such an instance is also called a Broker. These streams of messages can be transformed by passing through an application that makes use of the Streams API or be processed by an application that makes use of the Consumer API. A complete application consists of one or more producers to produce records, one or more consumers to consumer records, and any number of streams to transform the records as they move from producer to consumer.

Using the Producer API, it is possible to write an application for the sensor nodes that connect to Apache Kafka and send a stream of records containing the data collected by the sensors on the sensor nodes to the Apache Kafka instance. These records can then be processed in an arbitrary amount of ways such as filtering out irrelevant data, splitting the data into different sensible topics or performing calculations with the data to send to another topic for a consumer to later make use of. Finally, using the Consumer API it is possible to extract data from the topics and process it by storing it in a database, sending messages to other sensor nodes, sending the data to a different application so that machine learning can be applied to the data, etc.

Before writing *producer application*s for the sensor nodes Apache Kafka was tested by creating test applications that make use of the Apache Kafka core APIs that are then run entirely locally on the platform. The first applications consisted of a simple producer that generated records containing a random word chosen from an arbitrary list of words and sent these records to a topic, in this case named ‘test-input-topic'. A stream application then took records from that topic, displayed the contents of that record and then forwarded these records to a different topic named ‘test-output-topic'. Finally, a *consumer application* took records from ‘test-output-topic' and also simply displayed the contents of that record.

The next set of test applications that were written were very similar to the previous set, with the major difference being that instead of simple strings more complex objects were contained in the records. The objects used for these applications were simple representations of the planets in our solar system. The fields used to represent a planet were the planets name, capital and colour as strings, and its temperature, gravity and distance to the sun as floats[13]. To contain a planet object in a record, the object was first turned into a

[13] These fields were chosen to have a variety in data types, with all of them being static except for the planet’s temperature. Later on, when connecting Apache Kafka with InfluxDB, the chosen fields proved to help in understanding the difference between what InfluxDB defines as tags and what it defines as fields.

string representation of that object. Such a string representation was simply the six fields of a planet object in a fixed order, delimited by a colon. This string representation was then turned into a record and sent to a topic. Any application that took data from a topic containing these string representations could then turn that representation back into an object and process it as needed.

These test applications presented us with the problem of properly serializing objects that had to be stored in records. For an application with a single and consistent type of measurement a simple string representation was sufficient. However, if multiple different types of measurements have to be used, creating a custom serialization method for each of these measurements would significantly increase development overhead as well as reduce the maintainability of the code. To solve this, Google's GSON library was used to consistently serialize any simple object[14] to a JSON string. These JSON strings can then be deserialized to a map that represents the original object.

These first applications helped attain a better understanding of the workings of Apache Kafka from the standpoint of a developer.

[14] Sometimes also called a Plain Old Java Object or POJO

4.3 Connecting sensor nodes to Apache Kafka

4.3.1 Objective

In the previous step of the project, Apache Kafka and the Victhorious platform were used to solve the connectivity and scalability problems of the first proof of concept. Additionally, test applications were written in order to achieve understanding of how Apache Kafka works and how to utilize its core APIs. In this phase of the project, a producer application will be written for the sensor nodes so they can send their sensor data to Apache Kafka. On the side of the Victhorious platform, a consumer application will be written that displays the records containing the data sent by the sensor nodes.

4.3.2 Setup

The sensor nodes used in the first proof of concept will again be used to gather sensor data and send this data to Apache Kafka. They will remain in the same location they were placed during the first proof of concept. The only adjustment made to them will be the application they are running.

Whereas during the first proof of concept they were running a simple application to communicate with the server Pi, the sensor nodes will now run an application that makes use of the Producer API to stream data to Apache Kafka in order for it to be processed.

4.3.3 Behaviour

The sensor nodes collect data from their attached sensors, store this data in a map and then converts that map to a *JSON*[15] string using Pythons built in JSON library. Then, using the Python implementation of the Producer API, this string is subsequently sent to a topic on the Apache Kafka instance. A consumer application on the Victhorious platform finally reads that data from the topic and displays it.

4.3.4 Next step

Currently, the data received by the consumer is simply displayed and stored in a topic. In order for this data to be analysed it has to be visualised. Additionally, while the records in a topic are *persistently stored*, the data structure of Apache Kafka topics is not very well suited for performing analysis on the data. Topics require a consumer to be used to extract the data from it in order to perform data analysis. These shortcomings are resolved by using InfluxDB as persistent data storage and later on connecting Grafana to InfluxDB in order to create dashboards in which the data can be visualised in a more useful way.

[15] JavaScript Object Notation

4.4 Using InfluxDB to store data

4.4.1 Objective

Whereas the previous chapters discussed the preliminary steps for the Industry 4.0 development track; This chapter serves as an intermediary step between the connectivity and visibility stage. As such, it is also the first step towards actual Industry 4.0 capabilities.

As highlighted in the chapter discussing Apache Kafka, it is a versatile tool to create a robust and scalable messaging system. It even has a method to store data received on a topic persistently because of its logs stored on disk, as well as Kafka streams that can be converted to Kafka tables. Although this use is intended and feasible, there are better alternatives with regards to data storage scalability. One of these alternatives is the use of time series databases (TSDB), which are designed to handle the amount of continuous data that IoT systems produce.

Timestamped records are not only the preferred format for storing continuous data, they also allow to analyse it over time after being stored. Additionally, most TSDBs provide useful tools to retrieve and manipulate the stored data.

The primary objective of this chapter is to expand the existing system by addition of a persistent data storage system other than those that Apache Kafka provides. Specifically, an InfluxDB database will be installed on the platform for this purpose. Subsequently, a generic application will be created that allows Apache Kafka to write received records to the InfluxDB database in a consistent and reusable manner.

4.4.2 Setup

As illustrated in figure figure 12, Instances of Zookeeper and Apache Kafka have already been installed on the platform. In this chapter, an InfluxDB database will be added to that stack and configured. Then, a generic Apache Kafka consumer will be made, whose function is to write records from Kafka topics into InfluxDB. Since horizontal scalability is a focus of this project, this consumer should be reusable for a variety of records.

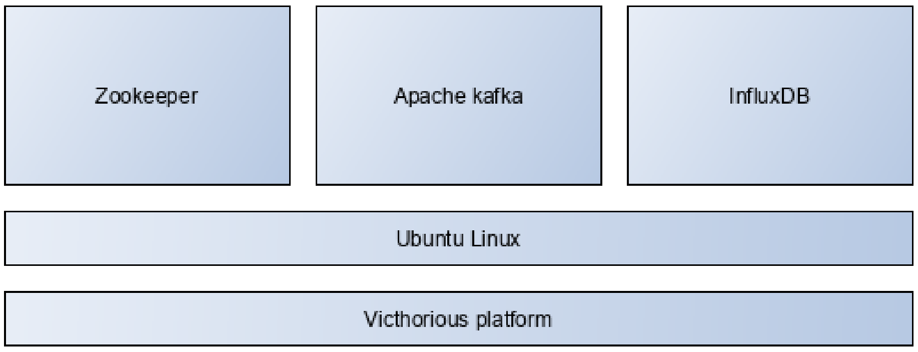

figure 12: Stack architecture including with Apache Kafka, Zookeeper and InfluxDB

In order to achieve the objective for this chapter, the stack also needs a method to interface Apache Kafka with InfluxDB. This interface is based on an apache Kafka Consumer so it actually consumes the records of the related topic. Since the desired outcome is also horizontally scalable, it should also be reusable for a variety of information. The solution lies in a generified Kafka consumer with built-in support for InfluxDB integration.

4.4.3 Behaviour

The interface between Apache Kafka and InfluxDB is based on a generic threaded influx consumer. This generic consumer will have a built-in interface with an InfluxDB database via the Influx API. Upon creation, classes that inherit from this consumer must specify the properties for the Apache Kafka consumer such as the topic to read and the location of the zookeeper cluster they want to communicate with.

The developer can then write a thread containing any logic that they want which converts Apache Kafka records to points that can be written to the InfluxDB database. Finally, the new class can be run from anywhere and will adapt the behaviour described in the provided thread. As a result of this, a multitude of applications can be created with varying behaviours that interface with InfluxDB in a reusable and standardized manner.

When the consumer is started, it starts reading out records from the specified topic and converts the data contained in these records to an InfluxDB point as specified by the developer. More specifically, the developer has to decide which fields contained within the record will be used as InfluxDB fields and which will be used as tags. Additionally, the developer has to specify either what timestamp will be used or to omit the timestamp, in which case the timestamp will be equal to the unix epoch time at which the record was written to InfluxDB. Once the point is created, the consumer then writes that point to a specified InfluxDB measurement using the InfluxDB API.

4.4.4 Next step

Now that the intermediate step towards the third stage of the Industry 4.0 development track has been finished, the project is ready for the first true Industry 4.0 capability. InfluxDB allows to store data and read it in a more usable and optimized manner than Apache Kafka topics can provide.

Additionally, InfluxDB opens up a range of new options regarding data analysis; In particular, because of the ease of reading data from these databases via InfluxQL queries. One of these options is visualizing the data with a visualization tool such as Grafana and Chronograf, which will be discussed in the next chapter.

4.5 Visualising data using Grafana

4.5.1 Objectives

The goal of this step of the research is to couple a visualisation tool to the InfluxDB data base in order to visualise the stored data. This visualisation is necessary in order to perform proper analysis on the collected data. The visualisation tool that will be used for this purpose is Grafana. Once Grafana has been coupled to InfluxDB, the next step is to create a dashboard that visualises the data collected by the sensor nodes installed in the office. InfluxDB

4.5.2 Setup

As visualised in figure 13, Grafana was added to the stack on the Victhorious platform. Then, the platform was configured to open a port on which the Grafana web interface could be accessed remotely.

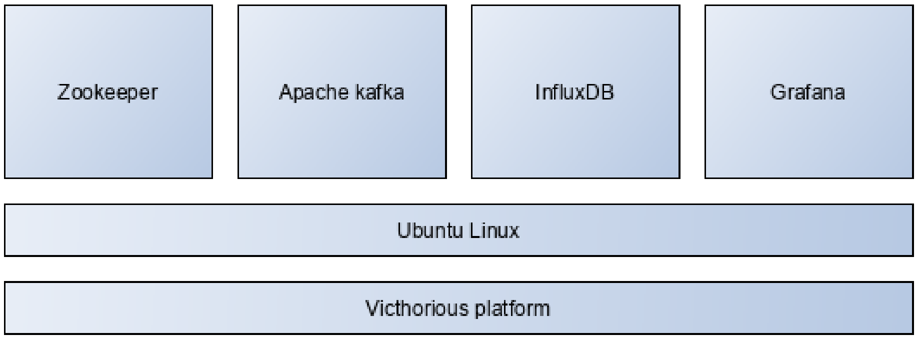

figure 13: Stack architecture including Zookeeper, Apache Kafka, InfluxDB and Grafana

4.5.3 Behaviour

To visualise data in Grafana, a data source must first be coupled. In this case an InfluxDB database was coupled. Once a data source is coupled, a dashboard must be created. Panels containing graphical representations of data are placed in this dashboard. Examples of such representations are graphs, heatmaps, tables and gauges. The data that has to be visualised is taken from the database by writing one or more InfluxQL queries that read the requested data.

A dashboard was made to visualise the data collected by the sensor nodes. This dashboard contains six panels. Five of these panels are graphs that directly display the data collected by the temperature, pressure, humidity, accelerometer and gyroscopic sensors of both sensor nodes. The sixth panel is a gauge that shows what temperature is currently measured by the sensor node placed at the ceiling of the office.

This dashboard can then be accessed via a web browser by entering the domain name of the platform along with the port on which Grafana provides its web application. The user then has to provide a login and password and can then select which dashboards they want to see or edit. An example dashboard can be seen in figure 14.

Grafana allows developers to create different users which have read or write access to specific dashboards. In the case of this project, a single account was used across all dashboards. When adding future data sources that are provided by external companies, specific accounts that allow creating and reading select dashboards must be made.

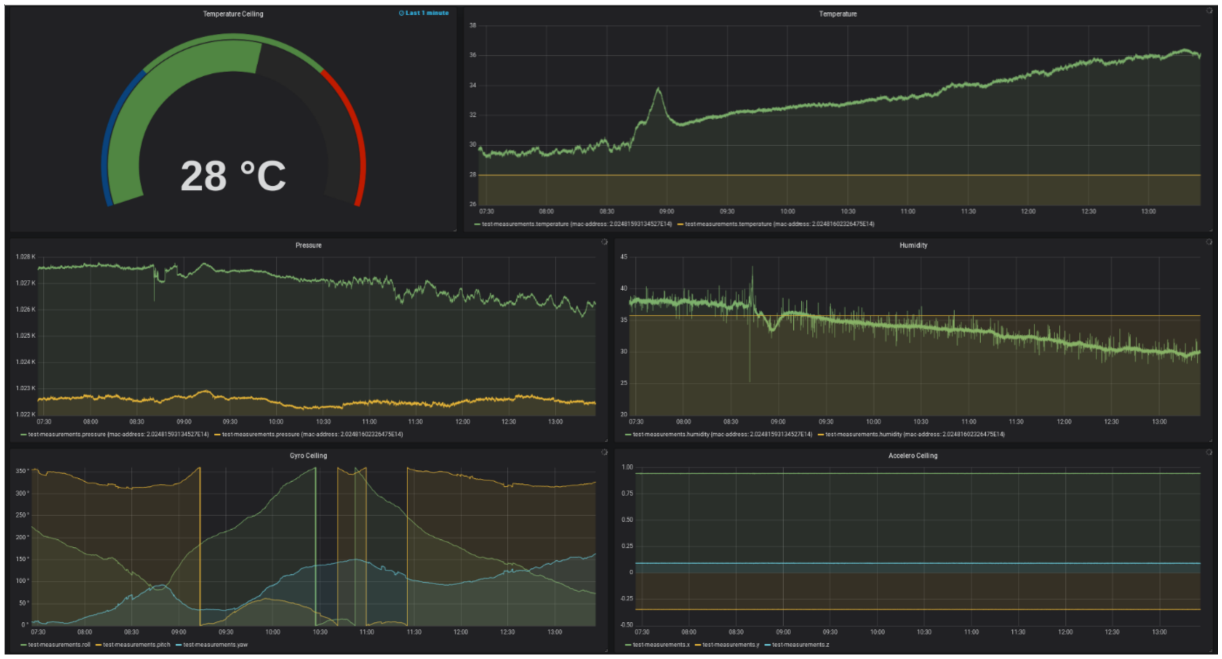

figure 14: A Grafana dashboard depicting sensor values in the office

4.5.4 Next step

At the end of this step a complete application to monitor any kind of data has been developed. There is a generic and consistent way of storing data, a fast and scalable messaging system is used to guarantee that data is processed as fast as possible while allowing for a variety of types of data, and finally an easy to use visualisation tool is used to make analysing this data easier.

The primary shortcoming at this phase of the project is that the producer, consumer and *streams applications* all have to be written specifically for each case. In order to make the project more scalable, generic and extensible applications have to be written that allow developer to focus on how the data is collected, processed and consumed rather than having to focus on how to make use of the Apache Kafka APIs. This will be the focus of the next step in this project.

4.6 Creating generic Apache Kafka applications

4.6.1 Objective

The goal for this step of the project is to create generic applications that make use of the Consumer, Producer and Streams APIs. These generic applications are intended to make developing applications for specific cases easier.

Currently, a new application has to written for each case. This has a negative impact on the scalability and maintainability of the project. Developers will be able to extend these applications in order to easily create new applications without having to deal with the overhead of writing boilerplate code, while also being ensured that the Apache Kafka side of the applications is handled in a consistent manner.

While the Apache Kafka core APIs have implementations in a variety of languages, the generic applications will only be written for the Java implementation. The reason for this is that the authors had most experience in using the Java implementations as well as that developing generic applications for the other languages would be too time consuming.

4.6.2 Behaviour

Generic Producers

Two types of generic producers were developed: A regular generic producer and a generic threaded producer. The regular generic producer was mostly made to take care of the boilerplate code regarding configuration of a producer.

These producers can best be seen as a wrapper around the constructor of the KafkaProducer class provided by the Java implementation of the Producer API. The developer only has to provide the name of the topic that the producer will send records to, the URL to their Apache Kafka broker and the logic that generates the records they want.

The generic threaded producers provide the same functionality as generic producers, while also taking care of the boilerplate code regarding the creation of threads that deal with the logic of creating records in a continuous way.

Generic Streams

Only one kind of generic streams application was developed. The generic streams application is very similar to the generic producers in that it can mostly be seen as a wrapper around the constructor for the KafkaStreams class that is already provided.

The developer has to provide an application id, the URL to the Apache Kafka broker they want to use and provide the topology. A topology in a streams application defines from what topics the application will read records, what logic is applied to these records and to which topics the stream should send the resulting records.

No new code was made to make creation of topologies easier as the Streams API already provides an excellent builder class for creation of topologies.

Generic Consumers

Four types of generic consumer applications were developed, each of which extended the functionality of the previous one. First, a generic consumer application was made that eliminated the configuration boilerplate code. Extending from that application was the generic threaded consumer application which, similarly to the generic producers, eliminated the boilerplate code regarding threading. The application that extends from the threaded consumer then took care of the boilerplate code of connecting a consumer application with an InfluxDB database. The final application which extends the InfluxConsumer introduced a generic way to convert a JSON string into a point that can be inserted in the database.

4.6.3 Closing remark

This was the last development step in the project. A generic and extensible way of gathering data and then visualising this data has been created, realizing the third step towards Industry 4.0 according to the Acatech Industry 4.0 maturity study. [1]

5. Results

5.1 Structure

The final system consists of the following modules: Apache Kafka, InfluxDB and Grafana, as is illustrated by figure figure 15. These modules are all running on the Victhorious platform. Using Apache Kafka guarantees horizontal and vertical scalability. InfluxDB provides the system with a persistent data storage, optimized for time series data. Grafana provides a way of easily visualising and analysing the data that is stored in InfluxDB.

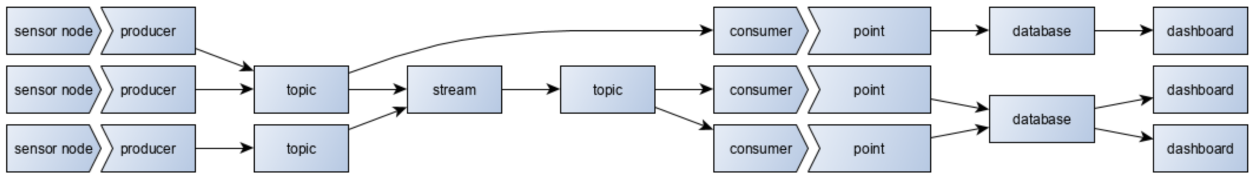

figure 15: A diagram depicting an example architecture data flow

The flow from capturing data to analysis is as follows: First, a sensor node collects data from its sensors. Then, a producer application running on the sensor node then converts this data to a record containing that data as a JSON string. This record is subsequently sent to a topic. Then, either a consumer application or any number of streams applications take records from that topic. In case of the streams applications, the records will be processed by the logic in those streams applications and then sent to another topic. In case of a consumer application, records are either taken directly from the topic that the producer application wrote to, or the topic that the streams application wrote to. The consumer application converts the JSON string to an InfluxDB point and writes the resulting point to a specified measurement in the relevant database. Finally, Grafana visualises that data in a dashboard after which it can be visually analysed.

5.2 Test-project: Raspberry Pi with sense HAT

Throughout this project, the Raspberry Pi setup for gathering atmospheric data has been a guiding theme for building new functionality. It has primarily served as a readily available data source to use for building new functionality.

Secondly, because the sensor nodes were located inside the researchers' office, the measured variables could be directly controlled by interacting with the sensor node. For example, the temperature of the sensor node could be slowly raised or lowered by using the office HVAC, or quickly raised with a blow dryer. The accelerometer sensor data could be manipulated as well by moving the sensor node.

In this application, both sensor nodes read sensor data from their attached Sense HAT. This data is formatted to a JSON string and put in a record, which is then sent to a topic on the Apache Kafka broker. Meanwhile, a consumer application is listening for records on that topic. When a record is added to that topic, the consumer application will convert the JSON string to an InfluxDB point and write it to the InfluxDB database. Finally, using Grafana, it is possible to see this sensor data in a structured and human readable manner. This behaviour is illustrated in a diagram in figure figure 16.

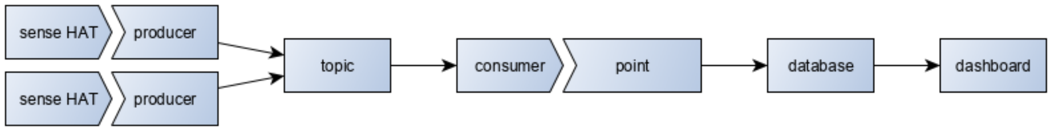

figure 16: A diagram depicting architecture for first proof of concept data flow

5.3 Test-project: Helldivers

In order to illustrate that the resulting system can handle more complex information from various data sources, an application was created that gathered data from the web server of an online videogame[16]. The web API used to read data from that server has previously been reverse engineered and documented [12]. Using this documentation, a Java implementation that retrieved data from the web server in question was created.

A producer application then made use of that the java implementation to pull data from the server. That data was subsequently structured in a JSON string in a record and sent to a topic. A streams application then retrieved data from that topic and split it into the three relevant categories to post that data to three topics made for these categories. Three consumer application then each read from their respective topic, and placed the data in an InfluxDB database. Finally, four dashboards were made using Grafana. The full data flow for this implementation is illustrated in figure 17. One of the dashboards serves as a general overview, and the other three display specific data for each of the three enemy factions in the game. An example dashboard can be seen in figure 18.

[16] HELLDIVERS™ by ArrowHead Studios

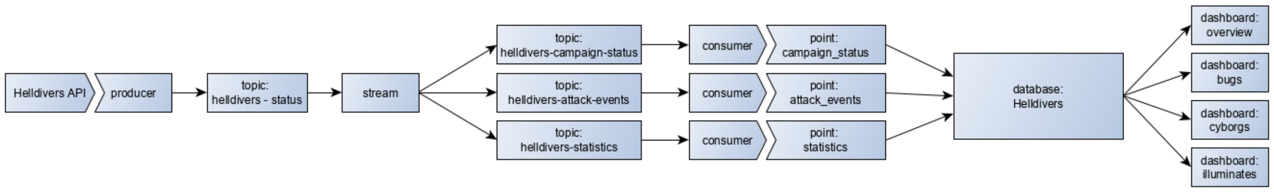

figure 17: A diagram depicting the architecture for Helldivers test application data flow

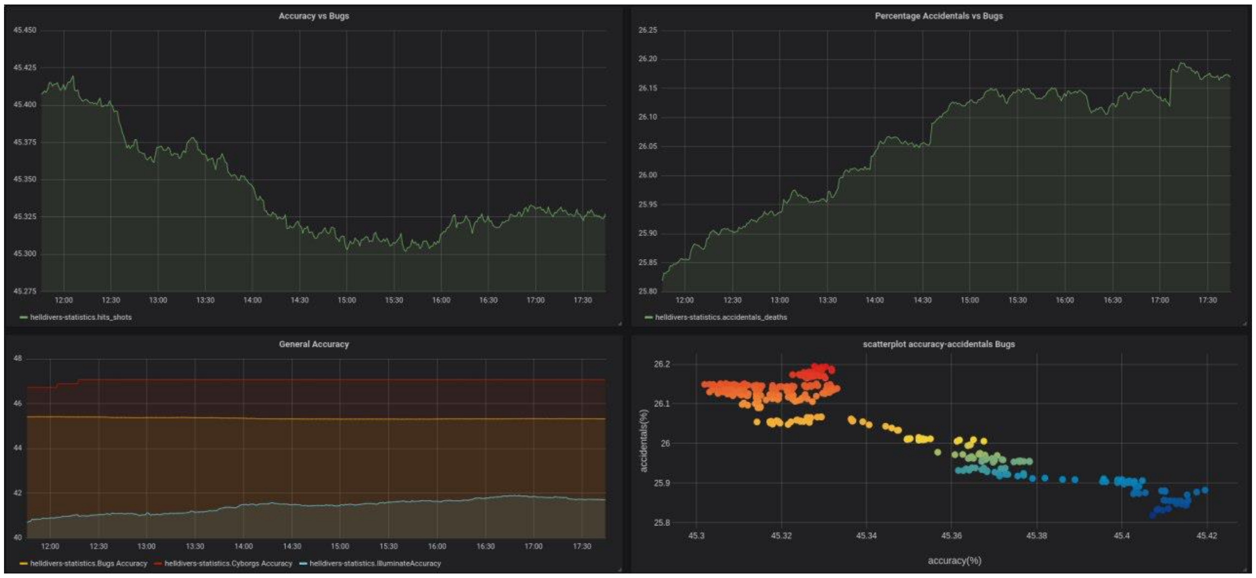
Accuracy vs Bugs
helldivers-statistics.hits_shots
Percentage Accidentals vs Bugs
helldivers-statistics.accidentals_deaths
General Accuracy
helldivers-statistics.Bugs Accuracy
helldivers-statistics.Cyborgs Accuracy
helldivers-statistics.IlluminateAccuracy
scatterplot accuracy-accidentals Bugs
accidentals(%)
accuracy(%)

figure 18: An excerpt from the overview dashboard created for the Helldivers test application

5.4 Use case: Vaillant

One of the main objectives for achieving Industry 4.0 readiness is to have a solution available to create a digital shadow of an existing physical system. To test this, a monitoring system for a Vaillant heating system was developed.

For connectivity with the Vaillant system, version 0.1.0 of the vr900-connector program by Thomas Germain was used [13]. A producer application was made that reads out all data that the Vaillant API could provide, filters out irrelevant data and sends this data to a topic.

In similar fashion to the other examples, a dashboard was made with Grafana to display these values in a way that can be visually analysed. The owner of the Vaillant heating system in question experienced problems with it and could use the developed system to help determine what the cause of these problems was.

For privacy reasons, as well as for the sake of brevity no diagram describing the behaviour of this system is shown.

6. Conclusion

6.1 Remarks and Observations

The primary objective for this thesis was to fulfil the preparing stages for the Industry 4.0 development track. This means that the desired outcome should be a system that can aggregate sensor data and subsequently transmit it to a central platform, which in turn allows other applications to use aforementioned data. [1]

At the end of this project, a system has been developed which provides a number of features. First, a variety of sensor nodes can be connected by means of a robust messaging system. Then, it provides a platform to aggregate and persistently store the data from those sensor nodes in a scalable manner. And finally, the collected data can be summarized and represented by medium of a visualization tool. The full architecture for this application is displayed in figure 19.

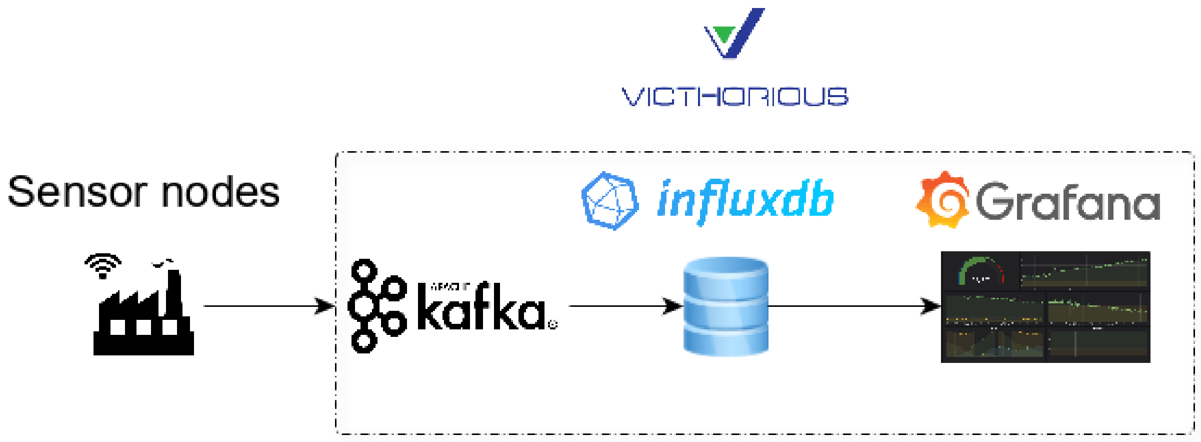

figure 19: A High level overview of the final architecture

The first technology used to attain this functionality is Apacha Kafka, which provides a robust and scalable messaging system. Apache Kafkas strength lies in its combination of both a publisher-subscriber model and a messaging queue model. Because of this, it is a versatile tool for creating this system. [6]

Secondly, the time-series database InfluxDB provides flexible, persistent and easily queried storage of the collected data. Through use of its query language InfluxQL, data stored in an InfluxDB database can be easily and rapidly extracted. [7]

Thirdly, the data visualisation tool Grafana allows for intuitive visualisation and analytics of the collected data. Grafana provides this functionality in the form of easily customised dashboards that display data in a variety of visualisation methods. The summarized data in these Grafana dashboards can be used to perform data analysis. [10]

And finally, the Victhorious platform serves as a shared environment for these tools to run on. While this is not a necessity, the reduced overhead that a cloud platform provides was remarkably beneficial to the development of this solution. [9]

In conclusion, the requirements for the primary objective have been fulfilled: The resulting system provides generified digitalization and connectivity capabilities to enterprises that wish to use it. Despite a focus on ease of use and extensibility, the researchers recommend employing someone with programming experience in order to properly utilize this system for specific needs.

The secondary objectives were achieved up to the visibility stage. While the researchers had hoped to initiate steps toward the transparency stage, a lack of reliable real-world data sources provided by external companies prevented this.

6.2 Drawbacks and limitations

As previously mentioned, the researchers had hoped to achieve Industry 4.0 capability to a higher degree. However, external companies to provide real-world data sources were unfortunately unwilling or unable to cooperate. Such partnerships and more importantly data sources are critical to the development of further stages of Industry 4.0 development.

During this project, a heavy emphasis was placed on developing a solution that was highly reusable. Although the resulting solution has a reusable codebase that eliminates most boilerplate code, some programming knowledge is required to properly implement functionality pertaining a specific case.

6.3 Further work

After completing this project, the researchers have the following suggestions for further development and research:

6.3.1 Graphical front-end

Adding a graphical front end to the system would eliminate the need for programming knowledge to configure the system. This would reduce the time and cost needed to get the system set up as well as reduce maintenance costs.

6.3.2 Distribution

Providing an easy to use method of distributing the created system would be beneficial for two reasons. The first reason being that companies can use their own platforms to run the system on, which reduces the cost to the company. The second reason being that this would reduce setup time significantly.

6.3.3 ERP Integration

Extending the existing backend by implementing a generified ERP integration would be the first step towards the transparency stage of Industry 4.0. In order to be a truly generic solution, a focus must be laid on providing out of the box connectivity with existing ERP systems.

6.3.4 Workflow engine

A workflow engine would allow for easy automated processing and alerting. Such a system would be essential in enabling proactive maintenance. This would also serve as an intermediate step towards the predictive capacity stage of Industry 4.0.

6.3.5 Translation of the generic applications

Translating the generic applications to other programming languages would help in reducing the system specific knowledge that is needed in order to use the system.

6.3.6 Maturing the system by medium of a real-world testbed

At the time of writing the system was only tested using a limited set of cases. Properly testing the system by using a real-world production facility as testbed would prove beneficial to creating a bug free system.

6.3.7 Predictive capacity

Applying machine learning algorithms to the collected data in order to predict machine defects, optimal production orders, delivery congestions is one of the steps needed in order to attain the predictive capacity stage of Industry 4.0.

Table of Literature

[1] G. Schuh, R. Anderl, J. Gausemeier, M. ten Hompel and W. Wahlster, Industrie 4.0 Maturity Index, Munich, Germany: Acatech, 2017.

[2] deltablue , „DeltaBlue Coud | DeltaBlue,” deltablue, 2019. [Online]. Available: https://delta.blue/cloud. [Geopend 2 Juni 2019].

[3] Apache Software Foundation, „Apache Kafka,” Apache Software Foundation, 2017. [Online]. Available: htttps://kafka.apache.org/intro. [Geopend 2 Juni 2019].

[4] InfluxData, Inc., „InfluxDB Open Source Time Series Database | InfluxDB | InfluxData,” InfluxData, Inc., 2019. [Online]. Available: https://www.influxdata.com/products/influxdb-overview/. [Geopend 2 Juni 2019].

[5] Grafana Labs, „Grafana - The open platform for analytics and monitoring,” Grafana Labs, 2019. [Online]. Available: https://grafana.com. [Geopend 2 Juni 2019].

[6] H. Lasi, P. Fetke, H.-G. Kemper, T. Feld en M. Hoffmann, „Industrie 4.0,” *WIRTSCHAFTSINFORMATIK,* vol. 4, nr. 56, pp. 261-264, 2014.

[7] L. Van De Loock, „Industrie 4.0 | vlaanderen.be,” Vlaamse Gemeenschap, [Online]. Available: https://www.vlaanderen.be/vlaamse-regering/industrie-40. [Geopend 2 Juni 2019].

[8] G. E. Moore, „Cramming more components onto integrated circuits,” *Electronics,* vol. 8, nr. 38, 1965.

[9] D. Zhuelke, „SmartFactory-Towards a factory-of-things,” *Annual reviews and control,* nr. 34, pp. 129-138, 2010.

[10] Solid IT gmbh, „historical trend of time Series DBMS popularity,” DB-engines, 2019. [Online]. Available: https://db-engines.com/en/ranking_trend/time+series+dbms. [Geopend 2 Juni 2019].

[11] InfluxData, Inc., „Open Source Time Series Platform | InfluxData,” InfluxData, Inc., 2019. [Online]. Available: https://www.influxdata.com/time-series-platform/. [Geopend 2 Juni 2019].

[12] M. Zechner, T. Cichoń, P. Polmans en C. Dommange, *The Unofficial Helldiver API Documentation,* EXceptional Threaded Computing, 2018.

[13] T. Germain, „thomasgermain/vr900-connector,” 2019. [Online]. Available: https://github.com/thomasgermain/vr900-connector. [Geopend 2 Juni 2019].

Appendices

Appendix A: Consolidated configuration file for the first Proof of Concept

```
{
 "temperature":{
  "ranges":
  [
   {
    "min":"0",
    "max":"10",
    "state":"vlo",
    "penalty":"2"
   },
   {
    "min":"10",
    "max":"20",
    "state":"lo",
    "penalty":"1"
   },
    …
  ],
  "default":{
   "state":"error",
   "penalty":"255"
  }
 },
 "pressure":{
  "ranges":[
   {
    "min":"0",
    "max":"975",
    "state":"under",
    "penalty":"3"
   },
   …
  ],
  "default":{
   "state":"error",
   "penalty":"255"
  }
 },
 "humidity":{
  "ranges":[
   …
  ],
  "default":{
   "state":"error",
   "penalty":"255"
  }
 }
}
```