Tampere University

Iiro Kiviluoma

INTEGRATING INDUSTRIAL PROCESS DATA TO AZURE USING OPC UA AND AZURE IOT EDGE

Process Data Integrator

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Assistant Professor David Hästbacka
Adjunct Professor Henry Joutsijoki
January 2023

ABSTRACT

Iiro Kiviluoma: Integrating Industrial Process Data to Azure Using OPC UA and Azure IoT Edge
Master of Science Thesis
Tampere University
January 2023

Industrial Internet of Things (IIoT) is a high-level concept that aims to integrate industrial devices and software by utilizing the data produced by the devices to improve efficiency and productivity in industrial environment. As industrial systems produce huge amounts of data, cloud computing can be exploited to access huge amounts of computing power and data storage in scalable manner, which meets the technical requirements set by the high volume of data. Heterogeneous industrial assets contain various standards and formats for information exchange that can be unified into an uniform interface by Open Platform Communications Unified Architecture (OPC UA), which is seen as the major standard for implementing information exchange for industrial systems.

Process Data Integrator (PDI) is a component that aims to integrate data from OPC UA to Azure in a seamless and configurable way. The aim of this thesis is to introduce a reference architecture for PDI in a way that it supports various OPC UA server configurations and the data flow from OPC UA to Azure can be configured to contain only the data that can be used to produce business value. The reference architecture should support subscribing for data changes in OPC UA servers and reading historical data from the servers.

The first phase of the thesis (chapters 2-4) performs a literature review on the thesis' key topics in order to gain knowledge about the IIoT environment and the essential features of OPC UA and Azure. State-of-the-art OPC UA web integrations are also reviewed in order to compare them to the PDI reference architecture. The second phase of the thesis (chapter 5) compiles the obtained knowledge and the functional requirements into an enumeration of design requirements, which contain the required functionality of the PDI reference architecture introduced in the third phase of the thesis (chapter 6).

The PDI reference architecture can be divided into two categories: asynchronous data flow components and synchronous API components to control the data flow. The data flow components consist of Azure IoT Edge implementation and Azure cloud components. The IoT Edge implementation consists of IoT Edge modules and is responsible for communicating with OPC UA Servers and extracting data by subscribing for changes or reading historical data. The received data is preprocessed and delivered to cloud in two paths: batches and data streaming. The cloud implementation contains components for processing the data flow in cloud in addition to a web API to control the data flow. The web API can be used to control the data flow from OPC UA servers by specifying the included data nodes, sample rates and delivery paths to cloud.

In conclusion, the PDI reference architecture introduces a system that bridges the gap between OPC UA and Azure in a configurable manner that is compatible with various OPC UA servers out-of-the-box. The data flow can also be configured in a way where the collected data contains the important values in order to produce business value and is delivered to cloud in a way that fits the needs best.

Keywords: Industrial Internet of Things, Cloud, Azure, OPC UA, Azure IoT Edge, integrations

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Iiro Kiviluoma: Teollisen prosessidatan integroiminen Azureen OPC UA:n ja Azure IoT Edgen avulla
Diplomityö
Tampereen yliopisto
Tammikuu 2023

Teollinen esideiden internet (Industrial Internet of Things, IIoT) on korkean tason konsepti, joka pyrkii integroimaan teolliset laitteet ja ohjelmistot keskenään tavoitteenaan parantaa tehokkuutta ja tuottavuutta teollisessa ympäristössä. Siinä missä teolliset järjestelmät tuottavat valtavat määrät dataa, pilvipalvelut tarjoavat valtavat määrät laskentatehoa ja tallennustilaa vastatakseen suureen datamäärään. Heterogeeniset teolliset laitteet hyödyntävät useita tiedonsiirtostandardeja ja -formaatteja, jotka voidaan yhdenmukaistaa yhden rajapinnan taakse hyödyntämällä Open Platform Communications Unified Architecturea (OPC UA).

Process Data Integrator (PDI) on komponentti, jonka tarkoituksena on integroida teollista prosessidataa OPC UA:n avulla Azure-pilveen (Microsoftin pilvipalvelu) konfiguroitavalla tavalla. Opinnäytetyön tavoitteena on tuottaa PDI-viitearkkitehtuuri, joka tukee monenlaisia OPC UA -palvelinkonfiguraatioita ja mahdollistaa datavirran konfiguroinnin yritysarvon tuottamista tukevalla tavalla. Viitearkkitehtuurin tulisi tukea datan muutosten tilaamista ja historia-arvojen lukemista OPC UA -palvelimilta.

Työn ensimmäisessä vaiheessa (kappaleet 2-4) suoritetaan kirjallisuuskatsaus työn kannalta oleellisiin aihealueisiin, joita ovat IIoT, OPC UA ja Azure. OPC UA:n yhteydessä tarkastellaan myös uusinta teknologiaa hyödyntäviä OPC UA -web-integraatioita, joita verrataan PDI-viitearkkitehtuuriin. Työn toinen vaihe (kappale 5) yhdistää kirjallisuuskatsauksella hankitun tiedon ja toiminnalliset vaatimukset luetteloksi suunnitteluvaatimuksia, jotka työn kolmannessa vaiheessa (kappale 6) esitellyn PDI-viitearkkitehtuurin tulisi täyttää.

PDI-viitearkkitehtuuri voidaan jakaa kahteen osaan: asynkroniseen datavirtaan ja synkroniseen rajapintaan datavirran ohjaamista varten. Datavirtakomponentit koostuvat Azure IoT Edge -toteutuksesta ja Azure-pilvikomponenteista. IoT Edge -toteutus koostuu IoT Edge -moduuleista, joiden vastuulla on vastaanottaa dataa OPC UA -palvelimilta arvojen muutoksia tilaamalla tai historia-arvoja lukemalla. Vastaanotettu data esiprosessoidaan ja voidaan toimittaa pilveen kahdella toimitustavalla: kootuissa erissä tai jatkuvana tietovirtana. Pilvipuolen toteutus sisältää komponentit saapuvan datavirran käsittelyn lisäksi verkkorajapinnan datavirran hallintaa varten. Verkkorajapintaa voidaan hyödyntää säätelemällä datavirtaa OPC UA -palvelimilta määrittelemällä halutut datapisteet, näytteenottotaajuuden ja toimitustavan pilveen.

PDI-viitearkkitehtuuri esittelee kokonaisuuden, joka yhdistää OPC UA:n Azuren konfiguroitavalla tavalla ollen valmiiksi yhteensopiva useanlaisten OPC UA -palvelimien kanssa. Datavirtaa voidaan myös hallita tavalla, jossa kerätty data sisältää ainoastaan yritysarvoa tuottavan datan toimitettuna pilveen sen tarpeiden vaatimalla tavalla.

Avainsanat: Teollinen esineiden internet, pilvi, Azure, OPC UA, Azure IoT Edge, integraatiot

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This thesis is the result of several months of work that started in the spring of 2022 and continued to the early winter of 2023. At times, it felt like there could have been more hours in a day in order to make it easier to deal with studying, working and doing this thesis at the same time.

I want to thank Insta for offering me an opportunity to do the thesis in the field of integrating automation systems with cloud, which proved to be a much more interesting topic than I could have thought. From Insta, I would particularly like to thank Pekka Savolainen for coming up with the thesis topic and Henry Joutsijoki for supervising this thesis and reading it from word to word probably more times that he needed to. I would also like to thank David Hästbacka for supervising this thesis and helping me mould the big picture of the thesis to what it is. My dear girlfriend Elli also deserves special thanks for being my emotional backbone during the process of making this thesis into reality.

At the time of writing this, my university journey started about five and half years ago. Many people and student associations have accompanied me during the journey and I think this would be a proper place to express my gratitude for a few of these. Thanks to Maanantaisauna for being part of the university journey from the very first day of orientation week to this moment by providing the best "löylyt" and continuous support during my university studies. If one thing has been permanent during my studies, it has certainly been Maanantaisauna. Also, thanks to Koneenrakentajakilta's boards of 2019-2022 for all the great memories and work we did during these years. Last but not least, I would like to thank PerinneSeura for making my time at the university more fun than I could have ever imagined. Thinking back to all the craziest memories, somehow PerSe takes part in almost all of them...

Tampere, 28th January 2023

Iiro Kiviluoma

CONTENTS

1. Introduction . . . 1
   - 1.1 Objectives and research questions . . . 2
   - 1.2 Thesis scope . . . 3
   - 1.3 Thesis structure . . . 4
2. Industrial Internet of Things . . . 6
   - 2.1 IIoT Architecture . . . 9
   - 2.2 Industrial Big Data . . . 11
   - 2.3 Cloud computing . . . 12
   - 2.4 Edge computing . . . 15
3. OPC UA . . . 17
   - 3.1 OPC UA client-server . . . 18
   - 3.2 OPC UA information model . . . 18
   - 3.3 OPC UA services . . . 22
   - 3.4 OPC UA security . . . 23
   - 3.5 OPC UA and web integration . . . 24
4. Azure . . . 28
   - 4.1 Azure Functions . . . 29
   - 4.2 Azure Storage . . . 29
     - 4.2.1 Azure Blob Storage . . . 30
     - 4.2.2 Azure Queue Storage . . . 30
   - 4.3 Azure IoT Edge & Azure IoT Hub . . . 32
   - 4.4 Azure Event Hub . . . 35
5. Process Data Integrator: Requirements . . . 36
6. Process Data Integrator: Architecture . . . 39
   - 6.1 Overall architecture . . . 39
   - 6.2 IoT Edge architecture . . . 43
   - 6.3 Database schema . . . 47
   - 6.4 Data flow . . . 49
   - 6.5 API . . . 51
   - 6.6 Feature review . . . 53
     - 6.6.1 Populating the source tags . . . 53
     - 6.6.2 Viewing the current state . . . 54
     - 6.6.3 Creating a tag subscription . . . 55
     - 6.6.4 Processing telemetry data . . . 56

6.6.5 Reading history data. 58
7. Result analysis 59
7.1 Architecture and required components 59
7.2 OPC UA compatibility 61
7.3 Data flow control 62
7.4 Edge computing 64
7.5 Comparison summary 66
8. Conclusion 68
References 72
Appendix A: Insta PDI requirements 76

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services (cloud computing platform by Amazon)
Azure	Cloud computing platform by Microsoft
GCP	Google Cloud Platform
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
IIoT	Industrial Internet of Things
IoT	Internet of Things
IT	Information Technology
JSON	JavaScript Object Notation
JWT	JSON Web Token
M2M	Machine-to-Machine
OPC	Open Platform Communications
OPC AE	OPC Alarms & Events
OPC DA	OPC Data Access
OPC HDA	OPC Historical Data Access
OPC UA	OPC Unified Architecture
OT	Operational Technology
PaaS	Platform as a Service
PDI	Process Data Integrator
REST	Representational State Transfer
RPC	Remote Procedure Call
SaaS	Software as a Service
SOA	Service-oriented architecture
Tag	OPC UA *Variable Node* whose value can be read
UUID	Universally Unique Identifier

X.509	Standard defining the format of public key certificates
XML	Extensible Markup Language

1. INTRODUCTION

Modern industrial systems produce huge amounts of data that has not yet been completely utilized. While data has ever growing importance in competitiveness, companies are looking for more ways to unlock the full potential that resides in their everyday process data. Industrial Internet of Things (IIoT) is a high-level concept that aims to bring devices and software closer to each other by utilizing the data produced by devices, such as sensors, with the aim of using the data to improve efficiency and productivity in industrial environment. IIoT provides access to better insight into company's operations by integrating data produced by company's everyday operations with its information systems, which typically includes data acquisition, storage and analytics.

IIoT is segment of a bigger context of Internet of Things (IoT). While IoT usually refers to human-based "consumer IoT", IIoT on the other hand refers to more machine-oriented approach and focuses on integrating operational technology with information technology systems (Ferrari et al. 2018, p. 2). This thesis' context focuses on integrating industrial process data to cloud, which explains why IIoT is the main area of IoT that will be addressed in this thesis.

The high volume of data produced by industrial systems sets many requirements for IIoT applications. Nowadays, many IIoT applications exploit cloud computing, because it meets the technical requirements by providing huge amounts of computing power, accessibility, data storage and other software services in a flexible manner where the cloud solution can be scaled to match the needs set for the application. Cloud computing can also be used to take advantage of "pay-per-use" pricing model which charges based on used resources, similarly as electricity usage bills, which means there are no wasted resources.

As the data volume is high as already stated, not every bit of data produced by industrial systems is important in a bigger picture. One way to decrease the amount of data is to use edge computing. Edge computing is a concept that aims bringing computation closer to data sources, which can lead to benefits such as lower response times and decreased bandwidth usage. Edge computing allows preprocessing of data before it being integrated into cloud.

Existing heterogeneous industrial assets contain various different standards and formats for data exchange, which complicates integrating them with each other and to external services. Open Platform Communications Unified Architecture (OPC UA) aims to solve the problem by unifying the underlying data models and is seen as the major standard for implementing information exchange for industrial systems. OPC UA provides an uniform interface that can be used to communicate between devices from different vendors and also from industrial systems to cloud. In this thesis, OPC UA plays a role of providing an interface for process data acquisition from OPC UA servers.

1.1 Objectives and research questions

All the things mentioned so far are all dependent on one thing: data. If there is no data available, none of the previously mentioned benefits such as better insight into company's operations can be achieved. This puts enormous amounts of importance to process data integration part in IIoT applications. In the ideal state, process data integration is seamless, efficient and "runs on its own" without manual human interaction.

In this thesis, the term "process data integration" covers extracting data from an OPC UA server, transforming the data into an unified data schema in cloud and everything in between. The entity that performs process data integration is called "Process Data Integrator" or "PDI". In IIoT application context, PDI provides the foundation for an IIoT application by producing data for the application to work with. After PDI has produced the data for the application to use, use cases for the data might differ between applications based on their requirements. For example, one application might target data analytics, another might want to produce a situation picture and some applications might want to do both with the data produced by PDI.

The main objectives of the thesis are based on the requirement list provided by Insta (Appendix A), which contains the major functional and environmental requirements of PDI. Insta's requirement list specifies OPC UA as the data source and Azure as the cloud platform. The problem definition of the thesis focuses on producing an architecture for PDI that satisfies the requirements. In addition to the requirement list, edge computing is also included in the research problem definition and therefore should be taken into account in the architecture. Based on the research problem, a simplified representation of PDI and its relations to the data source and the cloud is illustrated in Figure 1.1. It should also be noted that the data transformer in cloud can receive data from multiple different data sources and it is the transformer's responsibility to convert all data from different sources to the unified data schema.

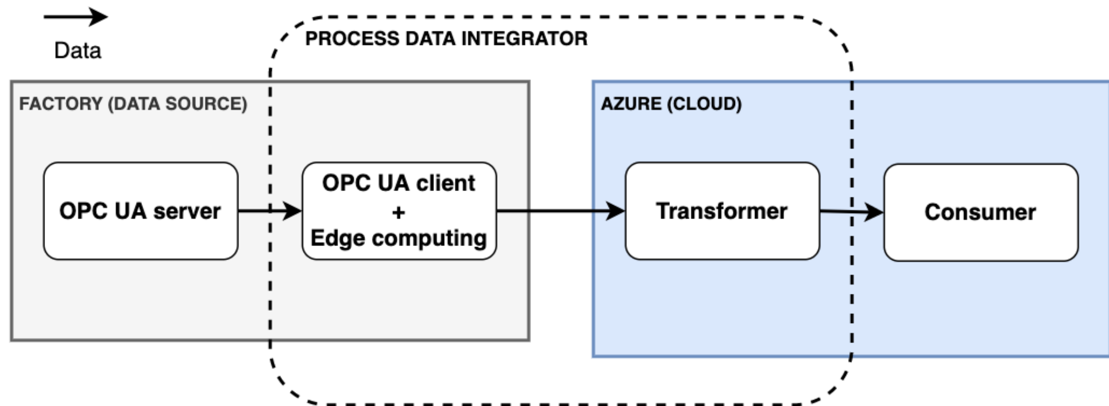

Figure 1.1. *Simplified representation of PDI and its relations to the data source and the cloud.*

The whole research problem can be described with the following research questions:

RQ1 What components does PDI require in order to integrate data from OPC UA server to Azure?

RQ2 How can PDI be configured to support variety of OPC UA server address space models, services and security measures?

RQ3 How can PDI be configured to control which and how data is collected from OPC UA server (see Industrial Big Data characteristics in Chapter 2.2).

RQ4 How can PDI benefit from edge computing between OPC UA data acquisition and sending the data to Azure?

1.2 Thesis scope

In order to prepare for the research and the empirical work, a few limitations are made in order to maintain the scope of the thesis. The limitations that apply to the thesis are enumerated below:

1. OPC UA is used to exchange data between operational technology (OT) and information technology (IT) systems.
2. Azure is used as the cloud platform that hosts the cloud-side implementation of PDI.
3. This thesis focuses on the OPC UA client-side implementation. OPC UA server-side implementation and server address space data modelling is not addressed.
4. Security in general is not addressed in this thesis. Many building blocks of PDI have built-in security features that are utilized, but they are not described further (e.g., OPC UA security mode/policy).

5. No graphical user interface (GUI). PDI provides an API (Application Programming Interface) for controlling its behavior.
6. Unified data schema is intentionally left undefined since it heavily depends on the application context. Because of this, only the input and triggers of cloud data transformer are addressed.
7. Control flow is not addressed in this thesis and PDI only covers data flow. However, data flow is a prerequisite for control flow which is why PDI implementation should be extendable to also cover control flow in future. (Data and control flows are defined in Chapter 2.1)

1.3 Thesis structure

This chapter scrutinizes the process used in this thesis to find answers the questions stated in Chapter 1.1. The thesis can be roughly divided in three phases that are shown in Figure 1.2. The phases are described in more detail in the rest of this chapter.

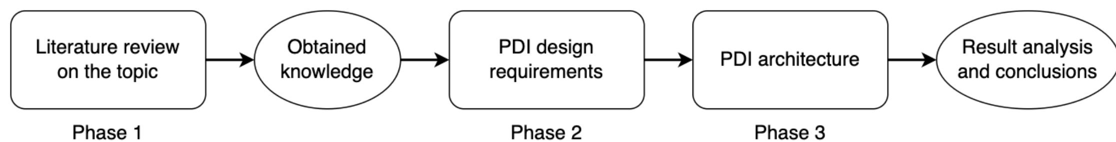

Figure 1.2. *Thesis structure.*

In phase one, a literature review is performed on the thesis' key areas to build basis knowledge on the thesis topic. Source material of the literature review consists of books, peer-reviewed articles and reliable online resources on the topic. The search keywords derive directly from the key areas and are searched upon on as their own or as a combination of keywords. The main search keywords of the literature review consist of "Industrial Internet of Things" (or "IIoT"), "Industrial Big Data", "Cloud computing", "Edge computing", "OPC UA" and "Azure". Citation pearl growth strategy is also used when gathering the source material.

The thesis topic is part of a bigger IIoT context. The literature review begins by reviewing IIoT's state of the art with the aim of obtaining basic knowledge of the aspects and challenges of an IIoT application from the thesis' point of view: overall architecture of IIoT applications, industrial Big Data, cloud computing and edge computing. The IIoT state of the art helps to identify the PDI scope inside the IIoT context, provides some early answers for the research questions in mind and lays out the theoretical background that will be addressed later in the thesis. The literature review is then extended into more technical areas of the topic: OPC UA and Azure. OPC UA review also contains the state of the art in addition to the technical background. Azure is mainly reviewed from the technical point of view, as the result of it being a cloud computing service.

After the state of the art and technical background literature review, phase two compiles the obtained knowledge and the requirement list (Appendix A) into an enumeration of design requirements that satisfy the research questions. Based on the design requirements, phase three produces an overall architecture of a PDI implementation that aims to fulfill the requirements. The result PDI architecture is then analyzed from the perspective of how the requirements were met and what are the challenges of the result architecture. The final conclusion of the thesis summarizes the result architecture with the outcomes of earlier chapters and talks about future visions on the thesis topic.

The chapters 2-4 contain the phase one by performing the literature review on the relevant topics from thesis' viewpoint: IIoT, OPC UA and Azure. The literature review is exploited to achieve the required knowledge about the topics relevant to cloud-based industrial OPC UA integrations. In the phase two, chapter 5 combines the achieved knowledge into an enumeration of requirements for the PDI reference architecture that should ensure the architecture covers all the important topics. The phase three is covered in chapter 6, which introduces the PDI reference architecture. The rest of the thesis, chapters 7 and 8, contain the result analysis analysis and the final conclusion of the thesis.

2. INDUSTRIAL INTERNET OF THINGS

Bruner (2013, ch. 1) describes Industrial Internet as an union between software and machines, where the network connected machines turn into web services ready to be coupled to software intelligence. One of the first introductions to the term Industrial Internet was given by GE (General Electric) in 2012 as their term for Industrial Internet of Things. (Evans and Annunziata 2012, p. 3) Industrial Internet can be seen as an umbrella term that covers many concepts that aim to achieve productivity and efficiency gains from industrial digitalization. Many concepts in addition to Industrial Internet, such as Industry 4.0 and smart manufacturing, deal with very similar topics that may be difficult to distinguish from each other, which is why many of them are often used interchangeably.

According to Gilchrist (2016, ch. 1) in a survey from January 2015, 87 % of industry business leaders did not have a clear understanding about Industrial Internet business models and technologies. This is somewhat expected as the complexities of the underlying technologies and practical implementations are ignored as Industrial Internet is often described at such high level. For clarity reasons, this thesis follows GE's example by using Industrial Internet of Things as a general term from now on. Boyes et al. (2018, p. 3) also agrees with the choice by stating that Industrial Internet is used as *"a short-hand for the industrial applications of IoT, also known as the Industrial Internet of Things, or IIoT".*

As previously stated, IIoT is a subset of IoT that focuses on industrial environment. Application areas of IoT can be diverse, which results in very different requirements for IoT applications. Because of this, it is important to acknowledge that IoT can be differentiated into four vertical aspects from the horizontal IoT concept, which can be seen in Figure 2.1. Even though the term "Internet of Things" refers to the concept of connecting physical devices to Internet, it is regularly used as a synonym for the consumer IoT that aims to improve human awareness with machine-to-user communication.

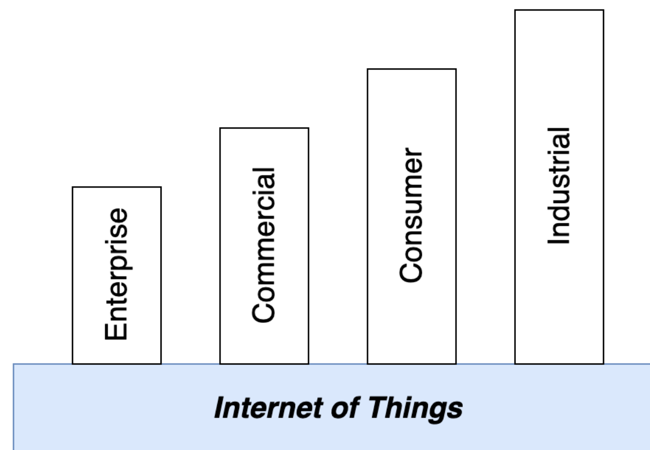

Figure 2.1. *Aspects of IoT. (Gilchrist 2016, ch. 1, modified)*

While consumer IoT is seen as revolutionary step since it focuses on new devices that can be connected to Internet, IIoT instead is considered as evolution because it aims to integrate operational technology with information technology systems by connecting the already existing industrial assets to information systems. As the data volume from IoT is heavily application dependent, the data volume from IIoT is very large as it mainly targets analytics. (Sisinni et al. 2018, p. 4725) Boyes et al. (2018, p. 3) also complies with this view by stating that IIoT is meant for bigger devices than smartphones and other smart devices as it pursues connect industrial assets to cloud over a network. Comparison of key differences between consumer IoT and IIoT is presented in Table 2.1.

Table 2.1. *Key differences between consumer IoT and IIoT. (Sisinni et al. 2018, p. 4725)*

	Consumer IoT	**IIoT**
Impact	Revolution	Evolution
Service model	Human-centered	Machine-centered
Current status	New devices and standards	Existing devices and standards
Connectivity	Nodes can be mobile	Nodes are fixed
Criticality	Not stringent	Mission critical
Data volume	Medium to high	High to very high

Gilchrist (2016, ch. 1) states that the reasons why companies should adopt IIoT originate from transforming business operational processes based on the results gained from applying advanced analytics to large data sets acquired from the company's everyday operations. IIoT provides access to better insight into company's operations through integration of machine sensors, middleware and information systems. Gilchrist continues that the actual business gains of IIoT are achieved through operational efficiency gains and increased productivity, which results in reduced downtime and optimized efficiency. According to Sisinni et al. (2018, p. 4729), productivity and efficiency gains can also be

achieved through smart and remote management. Even though most benefits of IIoT include changing company's operational processes, Boyes et al. (2018, p. 3) mentions that not all IIoT adaptations focus on changing the operational processes, but enabling real-time information to users, consumers and other processes.

As presented in Figure 2.1, IIoT is the largest vertical aspect of IoT. At this scale of operation, GE states that even if IIoT achieves just a one percent efficiency improvement then the potential gains are substantial. (Evans and Annunziata 2012, p. 4) Potential gains from Power of 1 % in different industry sectors are shown in Figure 2.2.

Potential Gains from Power of 1 %			
Industry	**Segment**	**Type of Savings**	**Estimated Value Over 15 years**
Aviation	Commercial	1 % fuel savings	$30 Billion
Power	Gas-fired system	1 % fuel savings	$66 Billion
Healthcare	System-wide	1 % reduction in inefficiency	$63 Billion
Rail	Freight	1 % reduction in inefficiency	$27 Billion
Oil & gas	Exploration & development	1 % reduction in capital expenditures	$90 Billion

Figure 2.2. *Potential gains from Power of 1 % in different sectors. (Evans and Annunziata 2012, p. 4, modified)*

Industrial companies have had sensors and devices producing data that is used to control operations for decades and machine-to-machine (M2M) communication and collaboration between machines has also existed for long in industrial environment. At first glance, the already existing M2M-technologies in industrial environment may look very similar to IIoT, but the difference is the scale of operation. For example, Big Data extracted from IIoT systems can be analyzed in cloud using advanced analytics at wire speed and data can be stored in cloud storage systems for future analytics performed in batches. These massive batch data analytics can be used to extract information and statistics in a way that would not previously have been possible. One other assumption about why IIoT has gained popularity is that complexity behind today's industrial systems has outpaced human operator's ability to recognize efficiencies which results machines to operate below their capabilities. (Gilchrist 2016, ch. 1)

2.1 IIoT Architecture

In order to understand how IIoT can be exploited to achieve business benefits, it is important to recognize the building blocks that form the big picture of IIoT. An IIoT reference architecture is a high-level abstraction of an IIoT solution that describes the overall structure and integrations between systems that reside within the solution. Different IIoT scenarios with unique requirements have produced various reference architectures, which are multilayered. Multilayered approach focuses on the services provided at each layer (functional domains), based on the used technologies, business needs and technical requirements set for the IIoT application. Information flows inside IIoT architectures can be simplified into two forms: data flow contains the data that will be processed and control flow contains the results of some computations. (Sisinni et al. 2018, p. 4726) The rest of this chapter examines two IIoT reference architectures and illustrates process data integration's role within the architectures.

One widely accepted IIoT reference architecture is a three-tier architecture, where the tiers consist of edge, platform and enterprise tiers that are connected by proximity, access and service networks. Visualization of the three-tier IIoT architecture is shown in Figure 2.3. The edge tier is where all data from sensors and actuators is collected and transmitted over the proximity network to an edge gateway. The platform tier receives data from the edge tier over the access network, to which the edge gateway is connected to. The platform tier is responsible for data transformation and processing (data flow) and managing the data flowing from the enterprise tier to the edge tier (control flow). The last tier, the enterprise tier, is the one that implements application and business logic for the domain-specific applications and user interfaces. (Gilchrist 2016, ch. 4; Sisinni et al. 2018, p. 4726-4727)

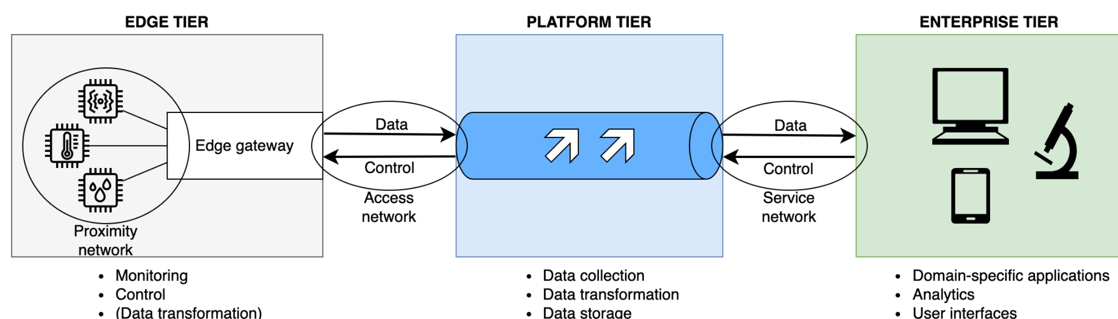

Figure 2.3. *Three-tier IIoT architecture. (Gilchrist 2016, ch. 4; Sisinni et al. 2018, p. 4727, modified)*

Figure 2.3 shows the relations between the three-tier architecture and functional domains. However, not all functional domains are bound to a certain tier. For example, data transformation can happen both in the edge tier or the platform tier. It is noteworthy that

the data transformation in the platform tier is performed on all incoming data, whereas the data transformation in the edge tier is device-specific and optional. (Gilchrist 2016, ch. 4)

Boyes et al. (2018, p. 7) present a slightly different IIoT architecture that contains four tiers, which is visualized in Figure 2.4. It differs from the three-tier IIoT architecture by including the edge IT as a separate tier, which highlights edge computing usage in IIoT applications. Edge computing allows computing to happen closer to the data source before the cloud, which can be exploited to reduce the load on the cloud. Benefits of edge computing will be addressed in more detail in Chapter 2.4. One more thing to take into consideration in the four-tier IIoT architecture is visualization, which is available on the both sides of the edge. Visualization on the left side of the edge is only capable of visualizing information of a single data source. Visualization on the right side of the edge happens in the cloud, which allows exploitation of data collected from multiple data sources providing a comprehensive overall view.

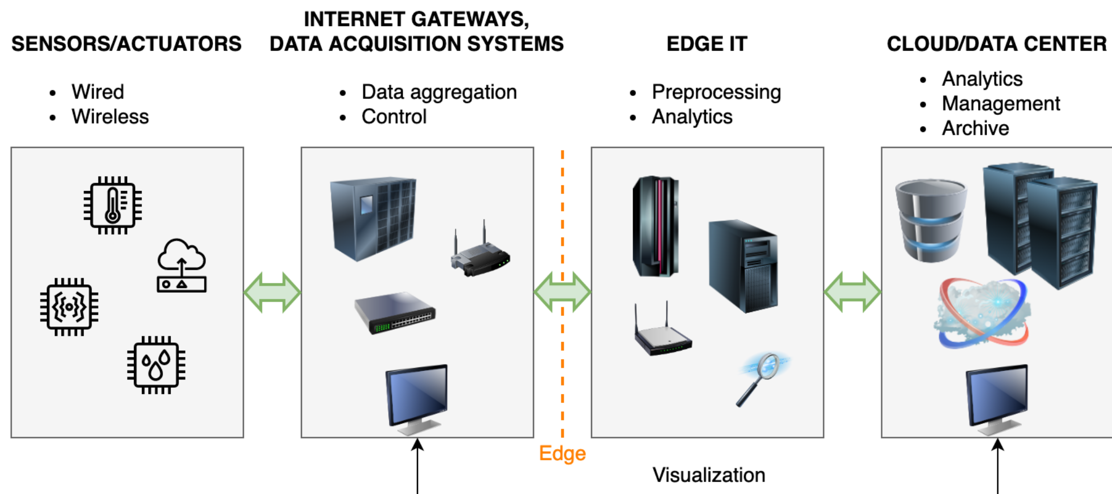

Figure 2.4. *Four-tier IIoT architecture. (Boyes et al. 2018, p. 7, modified)*

Both IIoT architectures exploit process data integration at some stage. Process data integration can be considered as a combination of the following functional domains:

- Data collection
- Edge data preprocessing
- Cloud data transformation

In the three-tier architecture, a single tier that performs process data integration can not be determined. Process data integration functions span into both edge and platform tiers but only cover a section of functions in both. The four-tier architecture provides the edge IT tier, which makes it easier to identify where the functions reside within the architecture. All the functions are performed on the right side of the edge: data collection and edge

data preprocessing are performed in the edge IT tier and cloud data transformation is naturally performed in the cloud tier.

2.2 Industrial Big Data

Industrial Big Data plays a key role in IIoT applications as advanced analytics depends on it. As previously stated, IIoT focuses on integrating existing heterogeneous industrial assets to information systems. Data produced by the heterogeneous sources come in large volumes and various forms. This sets requirements that, according to Gilchrist (2016, ch. 3), cannot be met by traditional databases and data processing tools.

The main purpose of Big Data analytics is to support decision making by providing accurate and reliable information based on Big Data. In order to analyze Big Data, one should be familiar with some basic Big Data characteristics before starting the analytics. (Saeed and Husamaldin 2021, p. 2) The characteristics are expressed with varying amount of V's. Mourtzis et al. (2016, p. 291) present that the basic characteristics consist of three V's, with new ones constantly being introduced. This Chapter introduces five V's that are essential from this thesis' viewpoint.

Volume
Volume refers to the amount and size of data being high. According to Gilchrist (2016, ch. 3), the ability to analyze large data sets is the whole purpose of Big Data as larger data sets produce more reliable results and forecasts.

Velocity
Velocity refers to how quickly the data can be used to make decisions after receiving it. Big Data applications generally send data in big batches, but some applications require data in real-time or almost real-time. These applications can not afford to wait for the batch to arrive, so it is essential that the important data can be obtained in real-time or almost real-time. In IIoT context, a device that sends data might also expect a response. In these cases, it is not beneficial to wait a data batch to be sent and analyzed as the data should be analyzed in almost real-time in order to report back to the device in a reasonable time. (Saeed and Husamaldin 2021, p. 2; Gilchrist 2016, ch. 3)

Variety
Variety refers to heterogeneous data sources, such as raw sensor feed or web API. Data coming from multiple heterogeneous sources means multiple different data formats and standards which have to be taken into account. The mere complexity of data sources poses a notable risk in data handling, as there is accurate and inaccurate data mixed together with different formats and units of measurement. (Gilchrist 2016, ch. 3; Vranopoulos et al. 2022, p.2)

Veracity

Veracity refers to innate unreliability in data sources. (Vranopoulos et al. 2022, p. 2) The previous three V's all rely on the data actually being true. For example, if a sensor produces false measurements the data is useless to begin with and any analytics applied to it lead to incorrect results. (Gilchrist 2016, ch. 3)

Value

Value refers to the ability to turn the collected data into actual business value and then leverage it to achieve goals. The end goal of Big Data analytics is to provide information that will help companies to make better decisions, which is why value is one of the characteristics that is critical to business decisions. (Vranopoulos et al. 2022, p. 2) Value is closely connected to the decision of what data to collect from all available data. It is a popular practice within industry to collect everything, but the data is only valuable if its relevance to the business value can be determined. (Gilchrist 2016, ch. 3)

The business value of industrial Big Data can further be illustrated by looking at a survey made by McKinsey Global Inc. (Mourtzis et al. 2016, p. 291) The survey claims that Big Data exploitation in manufacturing can halve development and assembly costs and can lead to 7 % reduction in working capital.

2.3 Cloud computing

Benefits of cloud computing satisfy technical requirements of IIoT, which is why it is widely exploited in IIoT applications. Services provided by cloud computing platforms, like huge amounts of computing power and storage, in addition to "pay-per-use" pricing model make cloud computing a very tempting opportunity for IIoT applications.

A comprehensive cloud computing definition by NIST (Mell and Grance 2011) describes cloud computing as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction"*. The NIST definition describes a cloud infrastructure as a collection of hardware and software that enables five essential cloud computing characteristics (Mell and Grance 2011, p. 2):

1. *On-demand self-service:*
   Cloud computing resources can be acquired and used at any time without cloud provider interaction.
2. *Broad network access:*
   Cloud computing capabilities are available over the network and can be accessed with heterogeneous client devices (e.g., laptops, mobile phones).
3. *Resource pooling:*
   Cloud provider's resources are pooled to serve multiple consumers in a manner

where the resources are dynamically assigned and reassigned based on consumer demand. The customer generally has no certain knowledge over the location of provided resources, but the location may be specified at a higher level (e.g. data center, country). This may be an issue when managing vulnerable data whose more precise location should be known at all times.

4. *Rapid elasticity:*
   Cloud computing capabilities can be elastically provisioned to rapidly scale with the consumer's demands. From the consumer's viewpoint, the capabilities that can be provisioned may seem to be unlimited and can be appropriated at any time to fit the needs.

5. *Measured service:*
   Cloud systems typically have some metering capabilities over its resources. Resource usage can be monitored, controlled and reported, which provides transparency for both the consumer and the provider, specifically in "pay-per-use" pricing model.

Services provided by cloud providers depend on the service model. Cloud service models usually refer to the three generally accepted service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Each service model abstracts underlying services which reduces effort to build and deploy systems by the service consumers. In traditional approach, the IT personnel must build and manage everything from physical server machines to security measures and routine updates. All three service models provide abstraction and automation for managing the cloud infrastructure which allows consumers to spend less time on managing infrastructure and more time on solving their business problems. (Kavis 2014, ch. 2) The NIST cloud computing definition describes the three cloud service models as follows (Mell and Grance 2011, pp. 2-3):

- *IaaS:*
  The service consumer has capability to provision fundamental computing resources such as processing, storage and networking. The consumer can not manage or control the underlying cloud infrastructure, but is able to deploy and run arbitrary software (e.g., operating systems, applications).

- *PaaS:*
  In addition to IaaS, PaaS provides the service consumer a collection of programming languages, libraries, services and tools supported by the cloud provider. Like in IaaS, the consumer has no control over underlying cloud infrastructure but has control over deployed applications and configurations for the application hosting environment.

- *SaaS:*

The service consumer has capability to use the cloud provider's applications running on a cloud infrastructure. The cloud applications can be accessed with heterogeneous client devices through an API or a client interface, such as a web browser. The consumer has very little control over anything, with the possible exception being limited user-specific application settings.

Figure 2.5 displays cloud stacks with different cloud service models. A cloud stack is a cloud architecture built on one or more cloud service models. The figure illustrates that the service models are built on top of each other and where the responsibilities lie between the provider and the consumer. Some example components of each service model and cloud stack are also illustrated in the figure.

Service Model	Cloud Stack	Components	Responsible		
	User	Registration; Login, Usage	Consumer	Consumer	Consumer
SaaS	Application	User Interface, Reports; Authentication/ Authorization	Consumer	Consumer	Provider
SaaS, PaaS	Application Stack	Application Server, Middleware; Database, Monitoring	Consumer	Provider	Provider
SaaS, PaaS, IaaS	Infrastructure	Data Center, Storage; Firewall, Load Balancer	Provider	Provider	Provider

Figure 2.5. *Cloud stacks with different cloud service models. (Kavis 2014, ch. 2, modified)*

The NIST definition also specifies the following cloud deployment models (Mell and Grance 2011, p. 3):

- *Public cloud:*
  The cloud infrastructure is provisioned for open use by a cloud provider and it serves multiple consumers. Hsu et al. (2014, p. 476) state that the key feature of public cloud is availability to the general public through the Internet. Example public cloud providers include Amazon (AWS), Microsoft (Azure) and Google (GCP).
- *Private cloud:*

The cloud infrastructure is provisioned for exclusive use within an organization. It may be owned and managed by the organization, third-party or combination of them and can potentially be hosted on-premise. Private cloud gives the organization greater control over the infrastructure and its resources compared to the public cloud. (Hsu et al. 2014, p. 477)

- *Community cloud:*
  The cloud infrastructure is provisioned for exclusive use within a community or group of consumers that share objectives (e.g., security requirements, policy). It may be owned and managed by one of the organizations, third-party or combination of them and can potentially be hosted on-premise.
- *Hybrid cloud:*
  The cloud infrastructure is a composition of multiple distinct cloud infrastructures: public, private or community.

Cloud providers use "9s" to measure cloud availability. The 9s come from the literal count of the number 9s in the availability. The more 9s there is in the cloud availability, the more an application running in cloud is available or "running". For example, an availability of five 9s (99.999%) indicates that cloud can be unavailable for 864 milliseconds a day or little over 5 minutes a year. Larger revenues mean larger losses in a short period of unavailability, which is why companies want to avoid the downtime. (Machiraju 2019, ch. 9)

From IIoT application context, PaaS service model seems to offer many building blocks required by IIoT applications. In addition to the raw computing power, storage and networking of IaaS, cloud platforms like AWS and Azure provide services, libraries and tools that can be used in developing and deploying IIoT applications. For example, Azure PaaS offers IoT components, integrations components, analytics and artificial intelligence (AI) in addition to DevOps and developer environments. Azure also provides management and security frameworks and tools to support IIoT application implementations. (Stackowiak 2019, ch. 2) Azure is addressed in more detail in Chapter 4 from IIoT application context.

2.4 Edge computing

Traditional centralized processing approach is to upload massive batches of collected data to a cloud server for unified processing. Despite the processing power and storage capability of cloud computing, the speed of Big Data growth has surpassed the load limitations of network bandwidth which causes a bottleneck that leads to increased latency. Edge computing is a computing mode that performs computing at the edge of network (see Edge IT in Figure 2.4). The "edge" in edge computing is relative to cloud computing and it refers to the resources (e.g., computing, storage) on the path from edge to cloud. Depending on the application context, the edge can be viewed as one or more

nodes on the edge-to-cloud path. (Ning et al. 2020, pp. 67-68)

Buyya and Srirama (2019, ch. 1) mention bandwidth and latency as part of the main challenges of a cloud-centric IoT approach called "BLURS": bandwidth, latency, uninterrupted, resource constrained and security. The first two are focused in more detail because they are closely connected to the thesis research questions. Data nodes can generate large amounts of data in a little time, which makes managing things at distant cloud very impractical. Cloud-centric approaches also include the latency from a connection to distant cloud, which is not suitable for applications that require low latencies.

Buyya and Srirama continue by stating that edge computing can be utilized to improve efficiency and reduce unnecessary costs. Ideally, when the process happens near data sources, the computation results are available much faster than in cloud-based computation. Outgoing communication bandwidth and unnecessary costs are also greatly reduced because most systems use gateway devices to perform the outgoing tasks. Edge computing also provides rapid responses for applications that require low latencies, which is derived from the computation being executed closer to the data sources.

3. OPC UA

Open Platform Communications Unified Architecture (OPC UA) is a platform-independent and service-oriented architecture (SOA) that enables standardized information exchange for industrial systems. OPC UA was released in 2008 and it combines all the individual OPC Classic, a predecessor of OPC UA, specifications into a single framework. The OPC Classic specifications were based on Microsoft Windows technologies and provided separate definitions for accessing process data, alarms and historical data: OPC Data Access (OPC DA), OPC Alarms & Events (OPC AE) and OPC Historical Data Access (OPC HDA) (*Classic - OPC Foundation* n.d.). According to Pauker et al. (2016, p. 322), OPC Classic is implemented in almost all industrial automation systems establishing it as well-accepted in the industry. Pauker et al. mention platform-independence, data exchange beyond local network and generic information model specification as the three major improvements of OPC UA compared to the older OPC Classic specifications.

OPC UA allows application to exchange information using a client-server model. Recent OPC UA specifications have also introduced another approach in addition to the client-server model, publish-subscribe, which aims to further enhance capabilities of OPC UA by offering better performance in variety of areas (Ioana et al. 2021, p. 2). The client-server is the mature one of the two, which is why it is the communication model used in this thesis. An OPC UA server typically provides information in the form of "nodes", which are accessible to OPC UA clients. In addition to the information, OPC UA also addresses security concerns by providing a set of security controls. Being future-proof is also a concern of OPC UA as it allows new technologies, such as new security algorithms and encoding standards, to be integrated into OPC UA while maintaining backwards compatibility. (*Unified Architecture - OPC Foundation* n.d.)

OPC UA specification consists of multiple parts. For example, the first seven parts focus on the core concepts of OPC UA (e.g., overview, security, information model) and parts eight to thirteen focus on access types (e.g., data access, history data access, alarms and conditions) (Pauker et al. 2016, p. 323). At the time of writing, the OPC UA specification consists parts 1-23, 100, 110 and 200. (*OPC UA Online Reference* n.d.)

This part of the thesis goes through the basics of OPC UA and the features that are essential when addressing the research questions. OPC UA specific terms are written

in *italics* (e.g., *AddressSpace*, *Node*) to emphasize that they refer to a term defined in the OPC UA specification. The features are viewed from a technical viewpoint as they are utilized in the empirical design phase. Current state of the art solutions of integrating OPC UA with web technologies are reviewed at the end of this chapter for reference.

3.1 OPC UA client-server

In client-server model, OPC UA system can consist of multiple OPC UA *Clients* and *Servers*. *Clients* can concurrently interacts with multiple *Servers* and vice versa. A component in OPC UA system can be both, *Client* and *Server*, at the same time to allow interaction with other *Servers*. (*Part 1: Overview and Concepts - OPC UA Online Reference* 2017, ch. 6.1) Figure 3.1 illustrates an OPC UA system architecture that includes a combined *Client* and *Server*.

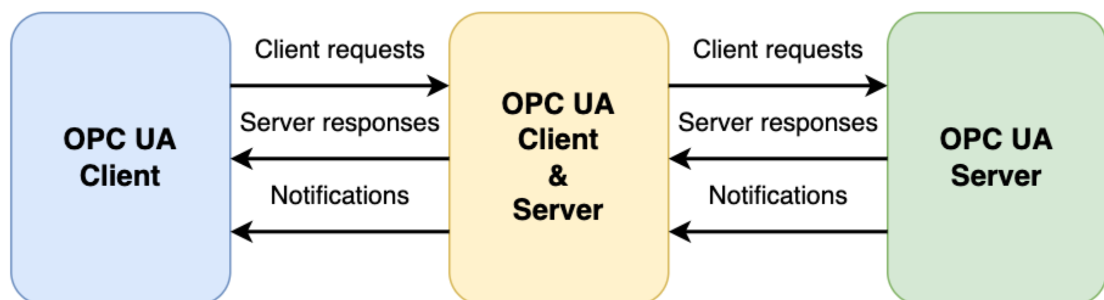

Figure 3.1. *OPC UA system architecture (*Part 1: Overview and Concepts - OPC UA Online Reference *2017, ch 6.1, modified)*

Both *Clients* and *Servers* are OPC UA *Applications*, which is an abstraction of the common functionality between all OPC UA applications. *Application* contains information such as *ApplicationDescription*, which includes global identifier and certificate of the application instance, name of the application and *ApplicationType*. *ApplicationType* specifies whether the *Application* is *Client*, *Server*, *Client* and *Server* or *DiscoveryServer*. (*Part 4: Services - OPC UA Online Reference* 2021, ch. 7.2-7.4) *DiscoveryServer* is a server that can be used to find all *Servers* registered to the *DiscoveryServer*. (*Part 4: Services - OPC UA Online Reference* 2021, ch. 5.4.1)

3.2 OPC UA information model

Information modeling framework is the fundamental element of OPC UA and it defines the rules necessary for building information models with OPC UA (*Unified Architecture - OPC Foundation* n.d.). OPC UA allows creation of information models from simple to very complex ones with its complete object-oriented capabilities. According to Pauker et al. (2016, p. 323), the base principles of OPC UA information modeling consist of the

following:

- Object-oriented techniques with inheritance and type hierarchies
- Type information that can be accessed the same way as an instance
- Network of nodes allows to connect information in variety of ways
- Extendable node type hierarchies and reference types between nodes
- No modeling limitations in order to allow an appropriate information model design
- Server-side is always responsible for the information modeling in OPC UA.

As stated earlier, OPC UA *Servers* use nodes (*Nodes*) to provide information. *Nodes* are located within a server address space (*AddressSpace*), which contains all the *Nodes* inside the *Server* that are used to represent real objects, their definitions and references to each other. (*Part 1: Overview and Concepts - OPC UA Online Reference* 2017, ch 6.3.4) Figure 3.2 visualizes an example OPC UA *Server* structure with three machines portrayed as *Object Nodes* and their sensors portrayed as *Variable Nodes*. In the simplified example, arrows in the figure represent "component of" relation. For example, variables "Variable 1.1" - "Variable 1.3" are components of "Object 1". OPC UA *Client* can access the machines and their sensors via the *Nodes* in the *AddressSpace*.

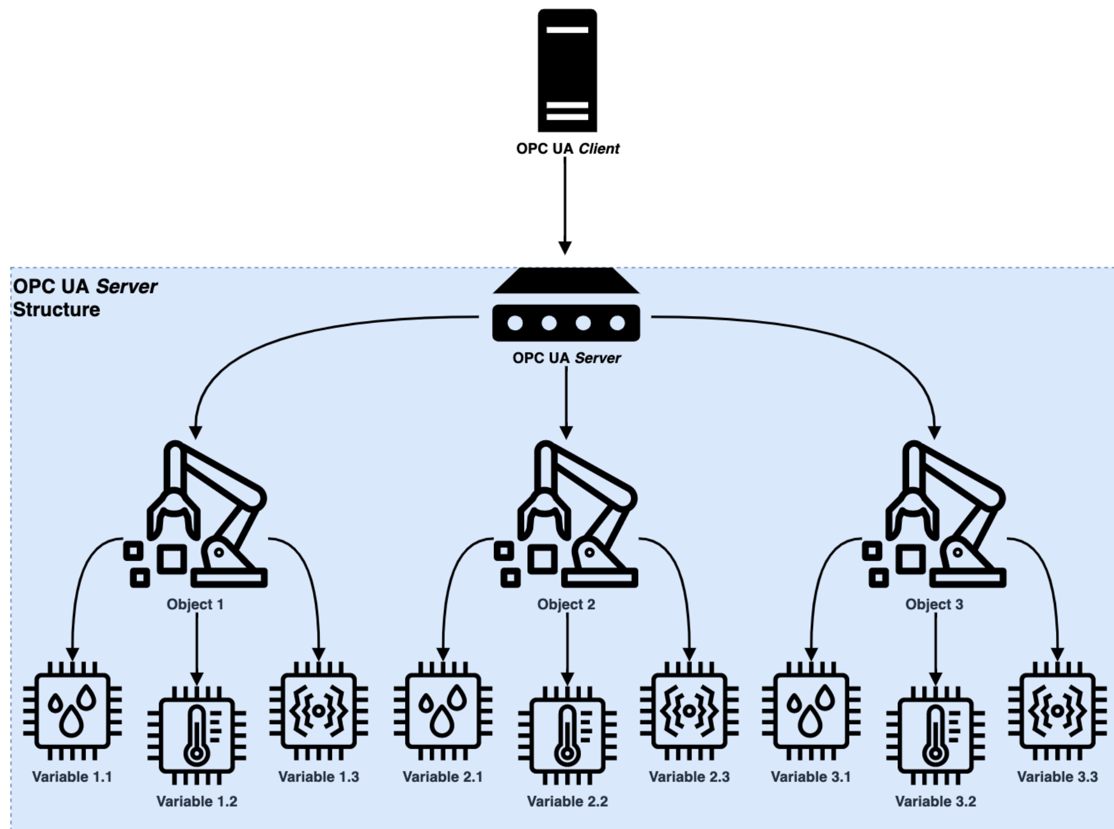

Figure 3.2. *Example OPC UA server structure.*

In Figure 3.2, *Nodes* are used to represent machines and their sensors. For example, to be able to differentiate high-level machine *Nodes* from the sensor *nodes*, the *Nodes* have various features in the form of classes, attributes and references. Each *Node* in *AddressSpace* has class information assigned to it in form of *NodeClass*, which defines the set of *Attributes* for a *Node*. In addition to *Attributes*, *Nodes* in *AddressSpace* are connected to each other by *References*. All *NodeClass* variations are derived from a common *Base NodeClass* that contains *Attributes* present in every OPC UA *Node*. These *Attributes* are *NodeId*, *NodeClass*, *BrowseName* and *DisplayName*. In addition to the mandatory *Attributes*, *Base NodeClass* also features optional *Attributes* (e.g., *Description*). (*Part 3: Address Space Model - OPC UA Online Reference* 2022, ch. 5.2) *Base NodeClass* and the *NodeClasses* inherited from it are described below (*Part 3: Address Space Model - OPC UA Online Reference* 2022, ch. 5):

- ***Base NodeClass*:** Base class for all the other *NodeClasses* that specifies the mandatory *Attributes* for every *Node*:
  - *NodeId*: Unique identifier of *Node*
  - *BrowseName*: Non-localized name used for creating paths in *AddressSpace*
  - *DisplayName*: Localized human-readable display name of *Node*.

  Base NodeClass also contains optional *Attributes*. For example, *Description* explains the meaning of *Node*. *Base NodeClass* does not specify any *References*.
- ***Object*:** *Objects* represent systems, components, real-world objects and software objects with references. Cavalieri et al. (2019, p. 46) describe *Objects* as containers for other *Objects*, *Variables* and *Methods*.
- ***Variable*:** *Variables* represent values and are always defined as *Properties* or *DataVariables*. *Value* attribute holds the current value and *DataType* attribute holds its type. *Value* also holds two timestamps for the value: *SourceTimestamp* (original timestamp set by the data source) and *ServerTimestamp* (timestamp when the server received the value).
- ***Method*:** *Methods* are callable functions that initiate actions within OPC UA server (Cavalieri et al. 2019, p. 46).
- ***View*:** *Views* can be used to define a subset of existing *Nodes*.
- ***ObjectType*:** *ObjectTypes* provide type definitions for *Objects*. All *ObjectTypes* are directly or indirectly inherited from *BaseObjectType* and can be used to extend *Objects* with additional functionalities. For example, *Objects* with *FolderType* type definition are used to organize *AddressSpace* (Cavalieri et al. 2019, p. 46).
- ***VariableType*:** *VariableTypes* provide type definitions for *Variables*. All *VariableTypes* are directly or indirectly inherited from *BaseVariableType* and can be used to extend *Variables* with additional information.
- ***ReferenceType*:** *References* between *Nodes* are defined as instances of *ReferenceType*

Nodes. For example, some widely used *ReferenceTypes* are *HasComponent*, *HasProperty* and *Organizes*.

- ***DataType***: *DataType* provides type definition for *Value* attribute of *Variable*. For example, *Float* can be used to represent numeric values and *Boolean* for on-off values.

OPC UA information model specification (*Part 5: Information Model - OPC UA Online Reference* 2022) describes standardized *Nodes* of *AddressSpace* and the model of an empty OPC UA server. It introduces variety of predefined types that can be used in information modelling without the need for user-defined types. The predefined types contain standard *ObjectTypes*, *VariableTypes*, *ReferenceTypes* and *Variable DataTypes*. The specification describes the standard hierarchical structure of *AddressSpace* by defining the *Nodes* that are present in every OPC UA server. Under the server root, there are three *FolderType Objects*: *Objects*, *Views* and *Types*. *Objects* folder contains *Server Object* that can be used to access information about the OPC UA server in question and the information model built with *Nodes* that is used to model the real-world system. *Views* folder contains *Views* into the *AddressSpace* and *Types* folder contains all known type definitions within the *AddressSpace*. Figure 3.3 shows the example OPC UA server structure from Figure 3.2 modelled into an *AddressSpace* and visualized with Prosys OPC UA Browser client application.

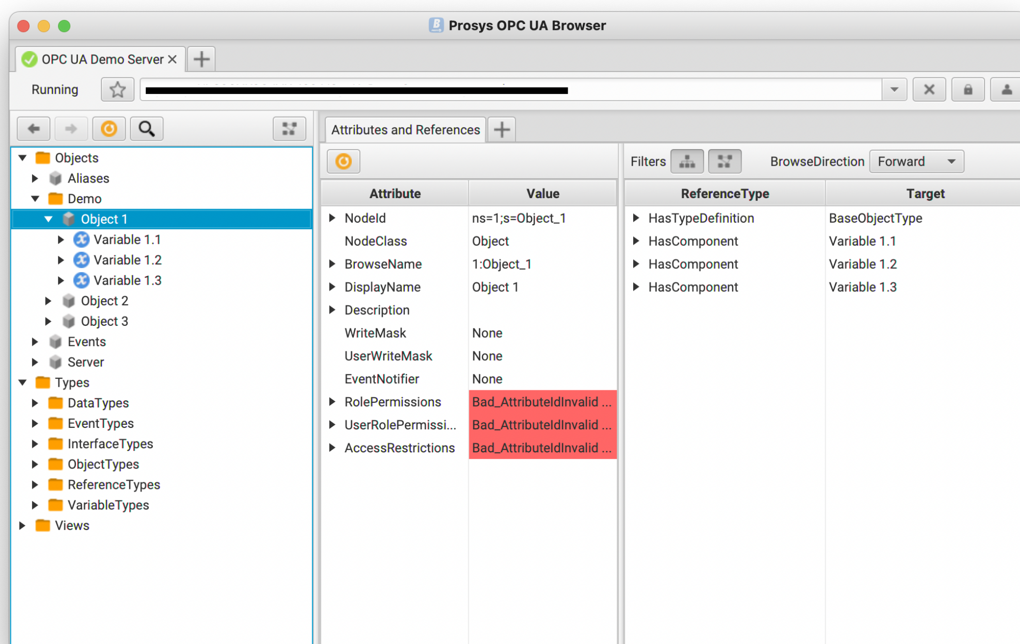

Figure 3.3. *Example OPC UA server AddressSpace from Figure 3.2 visualized with Prosys OPC UA Browser.*

Complex industrial systems can result in extreme amount of *Nodes* and deep hierarchies when modelled into an OPC UA *AddressSpace*, which makes the modelling a difficult task when done manually. Because of this, complex OPC UA *AddressSpace* contents are usually generated with a script or some algorithm. For example, Girbea et al. (2012) present algorithms that can be exploited to generate an OPC UA *AddressSpace* from existing data source (e.g., database, XML files, text files etc.). They mention shorter development times and easier maintenance as the main advantages of the *AddressSpace* generation algorithms. *Nodes* can be added, updated or deleted by only applying changes to the data source.

3.3 OPC UA services

OPC UA implements service-oriented architecture by diving its functionality to *Services* that are implemented by OPC UA *Servers* and called by OPC UA *Clients*. *Services* are the collection of abstract Remote Procedure Calls (RPC) and OPC UA *Servers* can choose which *Services* to implement, which is why not every OPC UA *Server* supports all the *Services*. (*Part 4: Services - OPC UA Online Reference* 2021, ch. 1)

Services are further organized into *Service Sets*, which define sets of related *Services*. OPC UA specification (*Part 4: Services - OPC UA Online Reference* 2021, ch 4.1) defines the following service sets:

- ***Discovery Service Set*:** *Services* that allow *Clients* to discover the required information in order to access the *Servers*.
- ***SecureChannel Service Set*:** *Services* that allow *Clients* to establish secure communication channels with *Servers*.
- ***Session Service Set*:** *Services* that allow *Clients* to manage *Sessions* and authenticate users on whose behalf the *Clients* are acting.
- ***NodeManagement Service Set*:** *Services* that allow *Clients* to manage *AddressSpaces* by adding, modifying or deleting *Nodes*.
- ***View Service Set*:** *Services* that allow *Clients* to browse *AddressSpaces* through subsets of *Nodes* called *Views*.
- ***View Service Set*:** *Services* that allow *Clients* to query subsets of data from *AddressSpace* or *View*.
- ***Attribute Service Set*:** *Services* that allow *Clients* to read and write *Attributes* of *Nodes*.
- ***Method Service Set*:** *Services* that allow *Clients* to call *Methods*.
- ***MonitoredItem & Subscription Service Set*:** *Services* that allow to subscribe to *Nodes* in *AddressSpace*.

From the thesis viewpoint, the most important *Service Sets* are *View*, *Attribute* and *MonitoredItem & Subscription*. Other *Service Sets* are also utilized in the data extraction process, but the mentioned sets contain the crucial functionalities. These sets contain the essential *Services* in order to define the available data *Nodes* from an OPC UA *Server* and extract values from them. *View Service Set* allows the *Client* to browse through an *AddressSpace* with *Browse* and *BrowseNext* services and acquire information about the *Nodes* that are available in the *AddressSpace*. *Attribute Service Set* provides *Read* and *HistoryRead* services that can be used to read current and historical values of *Nodes*. *MonitoredItem & Subscription Service Set* can be exploited by the *Client* to define *MonitoredItems* and subscribe to changes in them. Each *MonitoredItem* identifies the *Node* to be monitored and the *Subscription* that is used to send *Notifications* that can be transferred to the *Client* (*Part 4: Services - OPC UA Online Reference* 2021, ch. 5.12.1.1). Each *MonitoredItem* also required a sampling interval (or sample rate) which defines the cyclic rate that the *Server* uses to sample the item from its source (*Part 4: Services - OPC UA Online Reference* 2021, ch. 5.12.1.2). In the client-server model, the changes are published to the *Client* in response to *PublishRequests* (*Part 4: Services - OPC UA Online Reference* 2021, ch. 5.13.2.1).

3.4 OPC UA security

Security is one the core concepts of OPC UA. The specification defines three security modes for communication: *None*, *Sign* and *SignAndEncrypt*. Security modes are complemented with multiple *SecurityPolicies* that specify the cryptographic algorithms used for signing and encryption. (*Part 2: Security Model - OPC UA Online Reference* 2018, ch 4.8) For example, *SecurityPolicies* include *Basic128Rsa15*, *Basic256* and *Basic256Sha256*. *None SecurityPolicy* is also defined for configurations with the lowest security needs and should be disabled if any other *SecurityPolicy* is available. (*Part 7: Profiles - OPC UA Online Reference* 2017, ch. 6.6.161-6.6.166)

Whereas *None* security mode offers unprotected communication, *Sign* offers authenticated communication. *SignAndEncrypt* also offers confidential communication in addition to authenticated communication. All security modes and policies rely on the fact that OPC UA *Client* and *Server* must perform a security handshake in order to establish a communication channel using X.509 compliant *Application Instance Certificates* (*Application Authentication*). The security handshake is illustrated in Figure 3.4. Both parties maintain their own lists of trusted certificates, *Trust Lists*. A received certificate is considered valid if the certificate is in the *Trust List*. The certificate can also be part of a certificate chain and the certificate is also considered valid if the anchor of the chain is in the *Trust List*. (Kohnhauser et al. 2021, pp. 99300-99301)

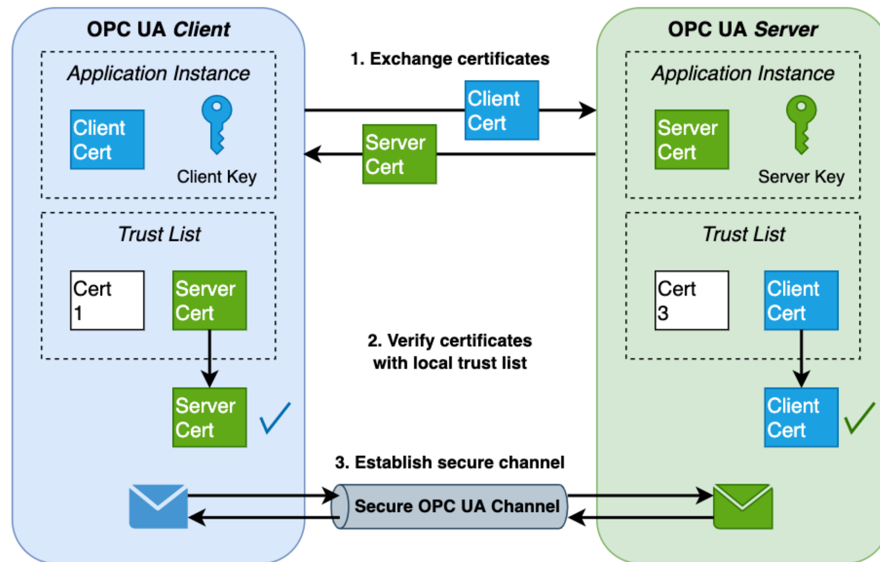

Figure 3.4. *OPC UA security handshake (Kohnhauser et al. 2021, p. 99301, modified)*

In addition to security modes and policies, OPC UA also supports *User Authentication* with user tokens (not to be confused with keys in Figure 3.4). OPC UA *Applications* support the following user tokens: username/password, X.509 certificate and JSON Web Token (JWT). (*Part 2: Security Model - OPC UA Online Reference* 2018, ch. 5.2.3) OPC UA also provides role-based security standard and *Servers* can choose to what extent the standard is implemented. OPC UA approach is to attach *Permissions* to *Roles* and *Clients* are granted roles based on the connection information. *Roles* are application specific and can be based on *User Authentication*, *Application Authentication* or security modes. (*Part 2: Security Model - OPC UA Online Reference* 2018, ch. 4.12)

3.5 OPC UA and web integration

Various efforts have been made to integrate OPC UA with web technologies in IIoT environment. Successful integration means that the potential benefits of IIoT (see Chapter 2) can be achieved by using OPC UA as the data source of an IIoT application, which is the main goal of the PDI. One common integration method has been wrapping an OPC UA *Client* behind a Representational State Transfer (REST) web service or API. The next part reviews three existing OPC UA and REST integrations by Cavalieri et al. (2019), Habib et al. (2022) and Gruner et al. (2016).

Cavalieri et al. (2019) proposed an OPC UA Web Platform based on REST that provides a simplified interface to access information maintained by OPC UA *Servers*. OPC UA Web Platform consists of a REST API and a middleware. The REST API is used by web users to access the platform and the middleware is the actual actor that performs the requested operations using an OPC UA *Client*. This way the web user can be considered

as a generic application running on a generic (smart) device and has no need to be OPC UA compliant nor implement OPC UA communication stack. The architecture of OPC UA Web Platform by Cavalieri et al. is illustrated in Figure 3.5.

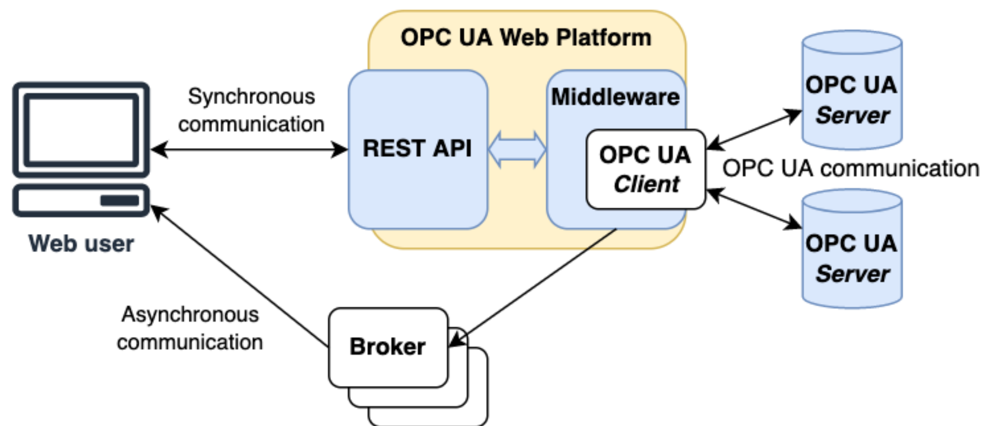

Figure 3.5. *OPC UA Web Platform by Cavalieri et al. (2019, p. 51, modified)*

OPC UA Web Platform by Cavalieri et al. maps various OPC UA *Services* to HTTP requests against the REST API. For example, the web user can ask information about OPC UA *Servers* and their *Nodes* via the API. The platform also uses brokers (also seen in Figure 3.5) to handle asynchronous communication between OPC UA *Servers* and the web user. A POST request to `/data-sets/dataset-id/monitor` can be used to create *MonitoredItems* and subscribe to their value changes. The monitoring request contains the OPC UA *NodeIds* that should be monitored and the sample rates that indicate how often value changes are wanted. The value changes are published to the broker and topic defined in the POST request. The web user can then subscribe to the broker topic in order to receive asynchronous updates about the value changes.

Habib et al. (2022) also proposed a REST-based OPC UA middleware approach. The proposed system takes few steps further as it also includes a cloud platform to collect and store the data from OPC UA for further usage, a Telegram messenger bot as remote controller and an OPC UA web service client for remote supervision. A simplified architecture of the proposed system by Habib et al. is visualized in Figure 3.6.

The OPC UA middleware acts as the crucial integration component between the OPC UA *Server* and the external services. Communication between the OPC UA *Server* and the middleware is handled with OPC UA *Client-Server* connection and communication methods beyond the middleware vary between the external services. The middleware uses an endpoint URL to establish communication with the OPC UA *Server*. As the cloud platform is used to collect and store the data, it can be further used for visual data analysis and aggregated bulk data management. Data is sent to the cloud platform using HTTP requests. The Telegram bot is also used to receive event notifications in addition

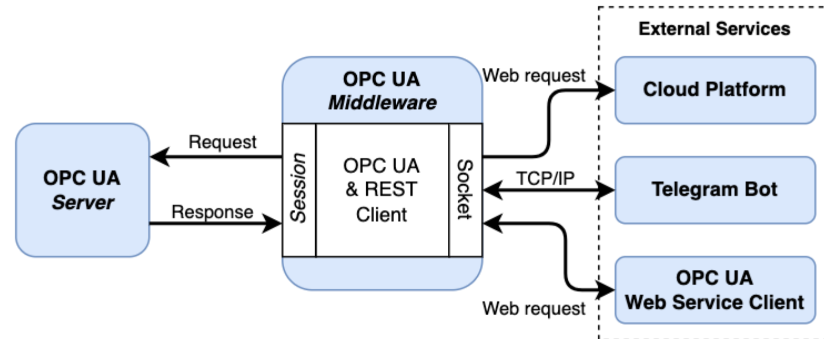

Figure 3.6. *Simplified architecture of REST-based OPC UA Middleware by Habib et al. (2022, p. 6, modified).*

to remote controlling. OPC UA *AddressSpace* can be remotely supervised with the OPC UA web service client. (Habib et al. 2022)

Gruner et al. (2016) introduced "RESTful OPC UA" extension to exploit the advantages of RESTful interfaces in industrial communication. According to them, the advantages of RESTful in industrial context include resource-oriented information model, stable interfaces and reduced resource consumption. SOAs utilize services to abstract the replaceable service providers for loose coupling of components. However in cyber-physical systems, the operations and their physical counterparts are never fully decoupled which allows resource-oriented information model to be exploited to explore the structure of resources using the REST principles. RESTful architecture only offers few services and representation formats are used to encapsulate the differences of the resources. RESTful service definitions are very stable since they are intended for general purpose. Stateless interfaces such as REST reduce the overhead from session management for short-lived communications since per-connection state does not need to be managed. (Gruner et al. 2016, pp. 1834-1835)

Gruner et al. (2016, p. 1838) evaluated the effects of stateless OPC UA with the RESTful extension compared to the standard OPC UA. A typical OPC UA message exchange for a singular service call includes multiple messages to able to establish a connection, call the service and close the connection. With stateless service call, the amount of messages in a singular service call can be significantly decreased. The message exchanges between standard OPC UA and stateless RESTful OPC UA are illustrated in Figure 3.7.

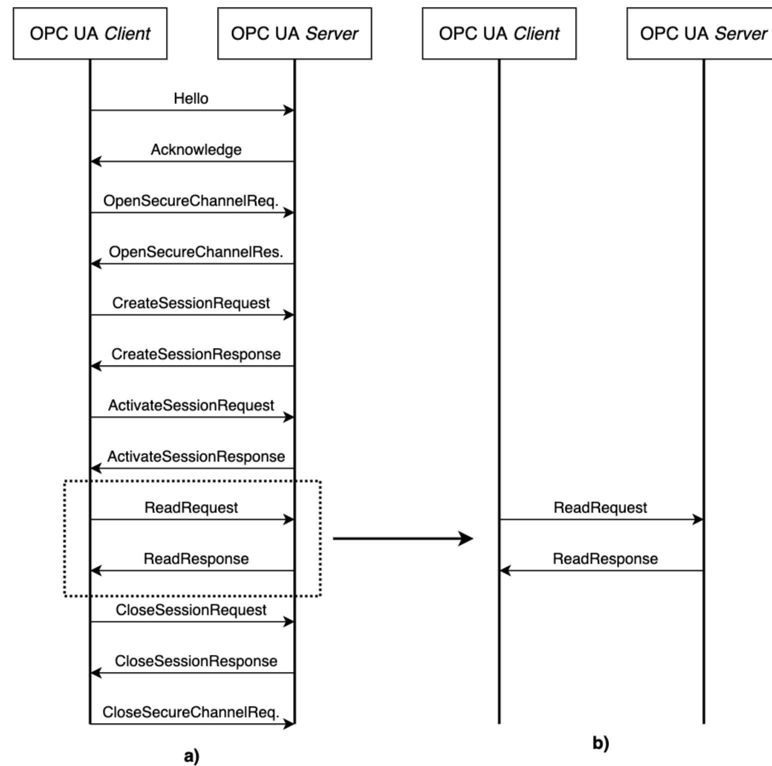

Figure 3.7. *A single ReadRequest message exchange comparison between **a)** Standard OPC UA and **b)** RESTful OPC UA by Gruner et al. (2016, p. 1838, modified).*

As illustrated in Figure 3.7, the standard OPC UA message exchange includes 13 packages to be able to perform a single *ReadRequest* if a session has not been previously established. On the other hand, the amount of messages required for a stateless *ReadRequest* is only 2. With the message amount going down from 13 to 2, the improvement equals to more than 50 %. (Gruner et al. 2016, p. 1838)

4. AZURE

Azure is a cloud computing platform develped by Microsoft that was first released to the general public on the 1st of February, 2010. Azure started as a PaaS product and consisted from the four pillars of the platform: compute service for running web applications, Azure Blob Storage for object storage, SQL Azure database service that resembled Microsoft SQL Server and Azure Service Bus as a message broker. Few years later, Microsoft started to invest in IoT services after realizing that IoT as one of the biggest drivers of data-driven workloads. Azure became one of the few public cloud computing platforms that offer end-to-end connected stack powered by services like IoT Hub, Event Hub and SQL database, which play a part in this thesis. In 2020, Microsoft expanded Azure's capabilities to the edge by introducing Azure IoT Edge, which is used for on-premise edge deployments in this thesis. (MSV 2020) At the time of writing, Azure contains more than 200 products and cloud services and 95% of Fortune 500 companies trust their business on Azure. (*What is Azure?* N.d.)

In Gartner's report of cloud providers, Bala and Gill (2021) place Azure as the second in the leader category, right behind AWS. They describe Azure as strong in all use cases and particularly well suited for Microsoft-centric organizations. Enterprises have also built trust in Microsoft over many years, which increases Azure's enterprise appeal. The report states that Azure offers the broadest set of capabilities of all cloud platforms, covering enterprise IT needs in a full range from IaaS to PaaS and SaaS. The report mentions resiliency and commercial complexity as some of the Azure's downsides. Microsoft has put much effort into improving resiliency with critical services, such as Azure Active Directory, but many are still concerned about the real-world impacts of outages with such critical services. Microsoft also has very complex licensing and contracting in addition to a complex account management structure.

This chapter of the thesis presents a brief technical background for each Azure service utilized in the PDI. The services include computing, storage and messaging services in addition to the services that enable the edge functionality of the PDI. The chapter is organized in a way where each subchapter covers a service or a couple of closely related services. Some of the services are visualized with Azure Portal, which can be used to view and manage applications running in Azure.

4.1 Azure Functions

Azure Functions is Azure's serverless "compute-on-demand" solution. In serverless model, the cloud infrastructure provides all the resources required by an application to run without the need for maintaining the server infrastructure. Azure Functions allows implementing blocks of code, called functions, that will run when certain events, called triggers, occur. For example, a function application can be triggered on HTTP requests, files uploads or messages from a message queue. (*Introduction to Azure Functions* 2022) Some example function application triggers are illustrated in Figure 4.1.

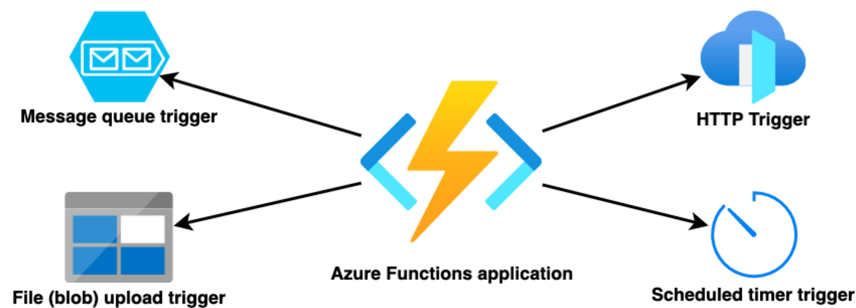

Figure 4.1. *Example Azure Functions triggers (*Introduction to Azure Functions *2022).*

Azure Functions offers development kits for multiple languages that can be used to implement functions. The currently supported languages include JavaScript/TypeScript, C#, F#, Java, Python and PowerShell. Azure Functions also provides scalability to function applications. When a function receives increasing amount of requests, Azure Functions can automatically spawn more function instances to meet the demand and drop the instances automatically when the requests fall. (*Introduction to Azure Functions* 2022)

4.2 Azure Storage

Azure Storage is Microsoft's cloud storage platform that offers high availability, massive scalability, durability and security in a managed cloud environment. Azure Storage can be used to store a variety of data objects that are accessible from anywhere in the world via a REST API. Azure Storage also offers client libraries for various programming languages that can be used to develop applications that utilize Azure Storage. Azure Storage's functionality consists of the following data services: Azure Blob Storage, Azure Queue Storage, Azure Table Storage, Azure Files and Azure managed disks. (*Introduction to Azure Storage* 2022)

Azure Storage's services are accessible using an Azure storage account. Storage accounts provide unique namespace for each Azure Storage within Azure, which is why storage account names must be unique. Azure Storage REST API also utilizes

storage account names as subdomains in addition to the fixed domain that includes the wanted data service. For example, Azure Blob Storage is accessible from the REST API endpoint `https://<storage-account-name>.blob.core.windows.net`. (*Storage account overview* 2022)

From the thesis viewpoint, the most important Azure Storage services are Azure Blob Storage and Azure Queue Storage. Azure Blob Storage enables sending data to cloud in batches and Azure Queue Storage is used to achieve asynchronous and durable communication between application components.

4.2.1 Azure Blob Storage

Azure Blob Storage is a cloud object storage solution optimized for storing massive amounts of unstructured data. Example use cases for Azure Blob Storage can include storing data for analysis, storing files for distributed access or serving documents and images. Azure Blob Storage consists of containers and blobs. An Azure storage account can have multiple containers which can contain multiple blobs. (*Introduction to Azure Blob Storage* 2022) Resources associated with Azure Blob Storage and their relations are illustrated in Figure 4.2.

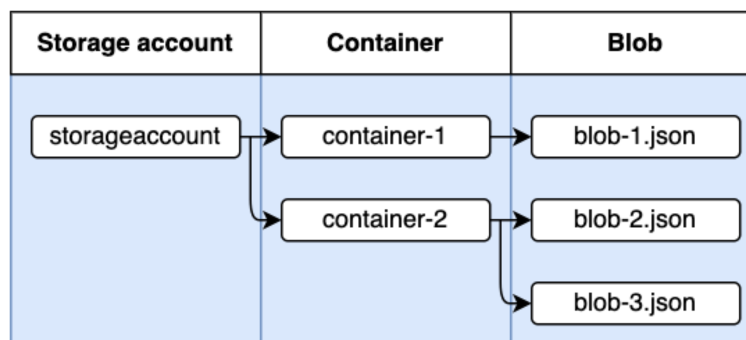

Figure 4.2. *Azure Blob Storage resource relations (*Introduction to Azure Blob Storage *2022, modified).*

In addition to containers and blobs, Azure Blob Storage also supports attaching metadata to them. Container and blob metadata are simple user-defined key-value pairs that can be attached to a container or a blob to provide additional information to the resource. For example, information about where the blob data originates from could be stored in metadata.

4.2.2 Azure Queue Storage

Azure Queue Storage is a messaging service that is commonly used to create a backlog of work to process asynchronously. An Azure storage account can contain multiple

queues and a queue may contain millions of messages up to the storage account capacity limit. Messages are typically used to represent a backlog of work for asynchronous processing and a single queue message can be up to 64 KB in size with "time-to-live" information is attached to it. Default time-to-live of a queue message is seven days, which means that the message expires after that and should not be processed. Azure Queue Storage resources and their relations are illustrated in Figure 4.3. (*What is Azure Queue Storage?* 2022)

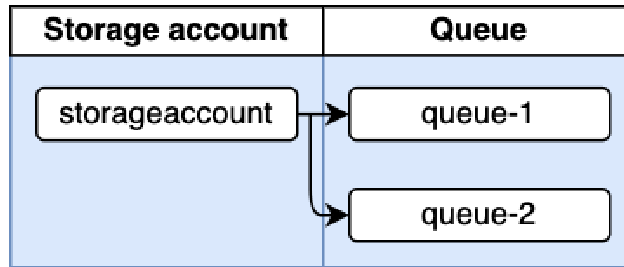

Figure 4.3. *Azure Queue Storage resource relations (*What is Azure Queue Storage? *2022, modified).*

If a queue has any messages in it, the messages can be received by clients. When a client receives a message from the queue, the message is not automatically deleted from the queue. Instead, the message becomes invisible to other clients for a visibility timeout and its dequeue count increases by one. The client that received the message is expected to delete the message from the queue if it has been successfully processed. The message becomes visible to other clients with the increased dequeue count if it has not been deleted before the visibility timeout. The client that receives the message with the increased dequeue count can determine if the message is considered valid based on the dequeue count. (*Get Messages* 2020)

As Chapter 4.1 mentioned, Azure Queue Storage can be used as function trigger. Azure Queue Storage can be exploited to create durable data pipelines with functions as data handlers that tolerate failures or the handler being offline. Azure Event Grid is a serverless and scalable broker that can be used to integrate application components using events (*What is Azure Event Grid?* 2022). With the exploitation of Azure Event Grid, a message can be inserted into a queue every time when a new blob is uploaded into a blob container. Durable data pipeline built with the mentioned Azure services is illustrated in Figure 4.4.

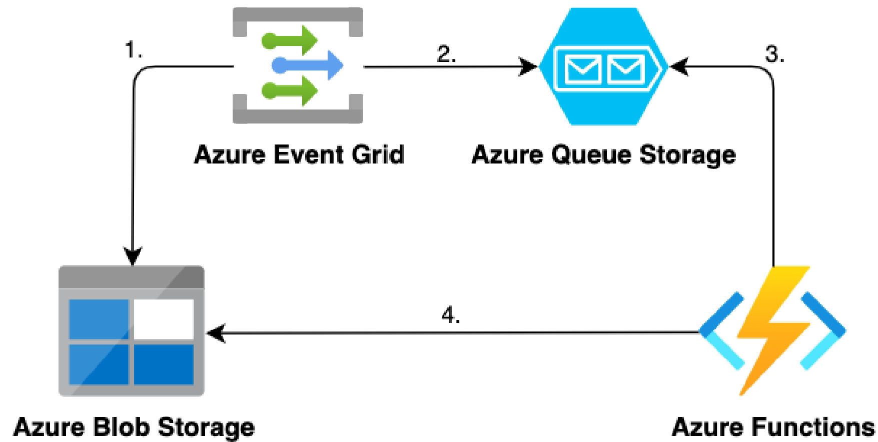

Figure 4.4. *Durable blob handler built with Azure services.*

In Figure 4.4, an event grid observes the blob storage (1.). Every time a new blob is uploaded, the event grid inserts a message that contains the blob identifiers to the queue (2.). The function is then triggered by the message in the queue (3.) and the function retrieves the blob based on the message (4.). If the function is offline, the messages keep accumulating and the function can start consuming the queue and blobs when it starts. If the function execution fails due to some unexpected error, the message becomes visible again after the visibility timeout and the function can try to process the blob again.

4.3 Azure IoT Edge & Azure IoT Hub

Azure IoT Edge is Azure's edge computing solution that enables deploying services, such as Azure services or own business logic, to on-premise environments. This allows performing computation tasks in the edge thus giving the ability to achieve the benefits of edge computing (see Chapter 2.4). Azure IoT Edge is comprised of three components: IoT Edge modules, IoT Edge runtime and cloud-based interface (*What is Azure IoT Edge* 2022).

IoT Edge modules are Docker-compatible containers that perform computation tasks in the edge. Modules are deployed to on-premise IoT Edge devices and are executed locally on those devices. Containerization allows deploying fully custom modules or packaging certain Azure services into modules. (*What is Azure IoT Edge* 2022) In addition to the container that contains logic, an edge module also includes module identity and module configuration. Module identities can be used to track multiple instances of the same module running on a device. Edge modules also have module twins, which are edge module specific configuration files that are accessible from the cloud interface. (Jensen 2019, ch. 2) Module twins and cloud-interfaces are examined later in this chapter.

IoT Edge runtime runs in each IoT Edge device and is responsible for managing the

modules of an edge device. IoT Edge runtime performs various management and communication operations, such as installing and updating modules, maintaining security, reporting module health to cloud and handling communication between modules, device and cloud. (*What is Azure IoT Edge* 2022) IoT Edge runtime is composed of two separate modules supplied by Azure: Edge Agent and Edge Hub. The Edge Agent is responsible for fetching module images from a container registry, starting modules and monitoring module states. If a module stops for some reason, the Edge Agent tries to restart the module to keep it running. The Edge Hub is primarily responsible for handling the communication. (Jensen 2019, ch. 2) IoT Edge runtime uses deployment templates to define module configurations. A deployment template contains information about each module that should be deployed to an edge device, which contains Docker images, Docker container options and environment variables of the deployed modules. (Jensen 2019, ch. 4)

Azure IoT Hub is Microsoft's preferred way to connect IoT Edge devices to Azure cloud. IoT Hub offers many key functionalities, such as managing devices, modules and twins. The IoT Hub also serves support for bidirectional communication that enables sending commands to edge devices and querying their state and metadata. (Stackowiak 2019, ch. 2) Furthermore, the IoT Hub also provides REST APIs that can be consumed by other services to manage and monitor edge devices. (Jensen 2019, ch. 2) The IoT Hub provides a cloud-based interface via Azure Portal. The interface can be used to perform all of the IoT Edge specific functionality. Figure 4.5 shows an example edge device list in the Azure Portal.

View, create, delete, and update devices in your IoT Hub. Learn more

+ Add Device Refresh Assign tags Delete

enter device ID Types: All + Add filter

Device ID	Type	Status	Last status update	Authentication type	C2D messages queued	Tags
iiroki-edge-device-demo	IoT Edge Device	Enabled	--	Shared Access Secret	0	

Figure 4.5. *Example edge device list visualized with IoT Hub and Azure Portal.*

As part of the IoT Edge functionality of IoT Hub, device and module twins can also be managed from the Azure Portal. Device and module twins are JSON documents that are created and maintained by the IoT Hub. Twins store metadata and properties of devices and modules. There are two kinds of properties: desired and reported. Desired properties are pushed to the device or module and reported properties vice versa. (Jensen 2019, ch. 2) An example module twin with the Azure Portal is seen in Figure 4.6.

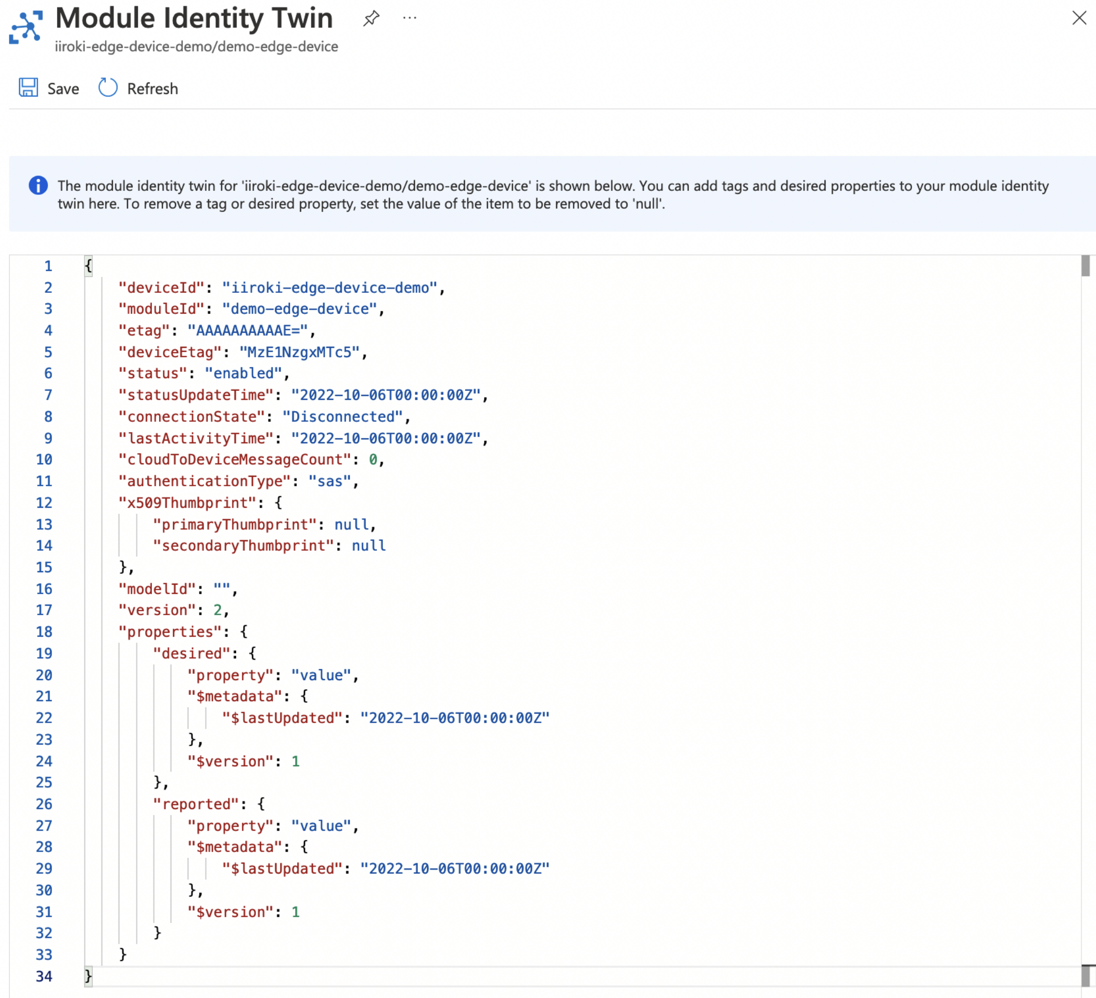

Figure 4.6. *Example edge module twin visualized with IoT Hub and Azure Portal.*

One common method of communication between edge modules and IoT Hub is direct methods, which represent a request-reply interaction similar to HTTP calls. Because of the request-reply pattern, direct methods are meant for communication that requires a confirmation of the method result. Direct methods can be invoked on an edge module by providing the edge device ID and the module ID. Edge modules can choose to listen for certain direct methods and execute their own logic on a direct method call. (*Understand and invoke direct methods from IoT Hub* 2022)

Merging everything together, IoT Edge and IoT Hub can be used to enable connection from the edge to Azure. Figure 4.7 illustrates an example environment where the real devices are connected to Azure using Azure IoT Edge. The example also emphasizes that the edge modules are the entities connected to the real devices.

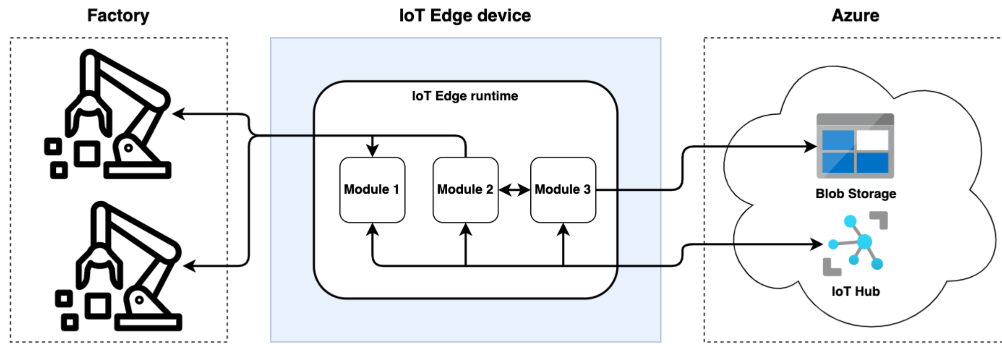

Figure 4.7. *Example Azure IoT Edge environment (*What is Azure IoT Edge *2022, modified).*

In addition to connecting an edge device to the IoT Hub, the edge modules can also be connected to other services besides the IoT Hub. In example from Figure 4.7, it can be seen that one of the edge modules is also connected to a Blob Storage located in the cloud. Being able to connect to other services besides the IoT Hub adds flexibility to custom edge module implementations.

4.4 Azure Event Hub

Azure Event Hub is Azure's big data streaming and event ingestion solution that has capability to process millions of events per second. Event Hub represent a "front door" for a data pipeline, which means it is located between event publishers and event consumers and allows decoupling the publishers and the consumers. Event Hub provides a distributed processing, which allows multiple consumers to process the incoming data. Some example scenarios where Event Hub can be used include live dashboards, archiving data and telemetry streaming. (*Azure Event Hubs — A big data streaming platform and event ingestion service* 2022)

Event Hub is a fully managed PaaS with little configuration or management overhead, which enables teams to focus on the business needs. Event Hub contains integration methods to many other Azure services. For example, Azure Functions include triggers for Event Hub events. Event Hub client library is also available in various languages and all the supported languages provide low-level integration, which creates a rich ecosystem for Event Hub. (*Azure Event Hubs — A big data streaming platform and event ingestion service* 2022)

5. PROCESS DATA INTEGRATOR: REQUIREMENTS

As the first requirement in Insta's requirement list (Appendix A) states, the high-level goal of PDI is to collect data from OPC UA to Azure. First step in process data integration is the data acquisition from OPC UA *Server*. The OPC UA *Client* that interacts with the OPC UA *Server* should be located as close to the data source as possible in the edge, for example in the factory where the data is gathered from. This allows edge computing to be utilized in data preprocessing before any data is sent to Azure. Example edge computing use cases in process data integration can include filtering or transforming the received data or already applying some algorithms to it before the cloud computing context.

PDI should support variety of different OPC UA data access methods. These possibilities might vary based on the capabilities of an OPC UA *Server*, but might include methods such as continuous data subscription or historical data access. PDI should also be configurable, which would allow enabling or disabling different features to match the capabilities of the OPC UA *Server*. Different OPC UA *Servers* might also use different security measures, which the edge solution built around an OPC UA *Client* should support.

When PDI sends data to Azure, the data should already have been processed to some extent in the edge and only the data that can produce business value should be included. The data is then transformed to an unified data schema in Azure and the transformed data can then be dealt with proper procedures, depending on the application.

The high-level objectives of PDI can be summarized into the following points:

1. An OPC UA *Client* located in the edge extracts data from an OPC UA *Server*.
2. Edge computing is utilized in data preprocessing before the data is sent to Azure, so only the data that has business value is sent.
3. The data is transformed to an unified data schema which allows it to be integrated with data from different sources.
4. PDI supports controlling what data to collect and in which manner (e.g., frequency, batches or streaming).
5. PDI is configurable in a sense that the same implementation is compatible with various OPC UA *Server* configurations (e.g., *AddressSpaces*, security measures).

6. PDI provides a good foundation for IIoT applications that use OPC UA as data source.

To achieve the objectives and answer the research questions, a requirement list for the PDI architecture is established. The requirements are based on Insta's requirements (Appendix A) and the research questions (Chapter 1.1), which can be used to specify the functional and non-functional requirements of the architecture. The functional requirements cover the required features of PDI and the non-functional ones cover the implementation requirements. The requirements set for the PDI architecture are listed below.

R1: Environment
The information exchange method to access the industrial data is OPC UA and the cloud platform where the data should be integrated is Azure. The PDI architecture is able to utilize services and components provided by Azure, including Azure IoT Edge. The OPC UA *Client* implementation used by the edge implementation is NodeOPCUA (*NodeOPCUA* n.d.).

R2: OPC UA compatibility and modularity
The edge implementation should be able to connect to OPC UA *Servers* with variety of different security measures and user authentication should also be supported. There is no unified way to represent a system with the OPC UA *Server AddressSpace*, which is why the edge implementation should support various *AddressSpace* structures and hierarchies. The features of the edge implementation can be enabled or disabled to match the capabilities of the OPC UA *Server*. For example, if the OPC UA *Server* does not support history reads, there is no need to have the history feature enabled.

R3: Data flow control
Data from OPC UA *Servers* can be extracted in two different methods: continuous data subscription and history data reading. With both methods, data source OPC UA *Nodes* and sample rate must be configurable. Subscription data can be configured to be delivered to Azure in two ways: batches or streaming. The timespan from which the history data is read can also be specified. History data can always be delivered in batches.

R4: Multiple OPC UA data sources
PDI must be able to extract data from multiple OPC UA *Servers* simultaneously. OPC UA *Nodes* are unique only inside their own *AddressSpace*, which is why the data source of incoming data must be identifiable.

R5: Edge computing
Only the data that can produce value is sent to Azure and the rest should be filtered out using edge computing. If possible, the data should already be preprocessed to some

extent which reduces the processing load in cloud.

R6: Management API

PDI should provide a management API to display its current state and control its behavior. At minimum, the API can be used to view the data sources including the data nodes within them and control the data flow. The management API should be a web API located in Azure.

In addition to the requirements 1-6, the presence of a data transformer should also be noted. The transformer transforms the incoming data into an unified data schema which makes the data coming from OPC UA compatible with the data originated from other sources. PDI only aims to function as a foundation for IIoT applications, which is why the unified data schema is not part of the thesis because it is heavily application dependent and can vary between applications. The presence of the transformer is noted in the architecture, but there are no requirements for it.

6. PROCESS DATA INTEGRATOR: ARCHITECTURE

The chapter represents reference architecture of PDI that aims to satisfy the requirements specified in Chapter 5 and thus achieve the objectives set for it and provide answers to the research questions defined in Chapter 1.1. When referring to the requirements from the previous chapter, the short format of R<number> is used (e.g., R1, R2). The reference architecture is initially examined as a whole and the focus then shifted into the architecture of the integration method between OPC UA and Azure, the IoT Edge implementation. After the edge implementation, the database schema and the data models between various components in the architecture are introduced. The last entity in the architecture, an API to control the PDI, is also examined from the viewpoint of the services it provides. When the architecture has been introduced, a feature review is performed. The feature review aims to find out how the features suggested by the architecture satisfy the requirements set for it and thus the research questions.

6.1 Overall architecture

The high-level reference architecture of PDI consists of three main sections: OPC UA *Servers*, Azure IoT Edge implementations connected to them and an Azure architecture connected to each IoT Edge implementation. The OPC UA *Servers* were already examined to the required extent in Chapter 3, which is why they do not require separate examination in this chapter. The OPC UA *Servers* only act as the starting point of the data flow to which the IoT Edge implementation should be capable to be integrated with. The edge implementation consists of custom modules implemented for the purpose in addition to prebuilt modules provided by Azure. IoT Edge is responsible for extracting the data from an OPC UA *Server*, possibly preprocessing it and transferring the data to Azure. This chapter only briefly examines the IoT Edge implementation from an external viewpoint and it is more thoroughly examined in Chapter 6.2. The Azure architecture is responsible for processing and transforming the data in cloud and consists of multiple Azure services, including computation, storage and messaging services. The overall reference architecture of PDI is illustrated in Figure 6.1, which also contains unique identifier numbers of the resources present in the architecture. The resources and their brief descriptions are in Table 6.1 and the table uses the same numbering as the architecture figure.

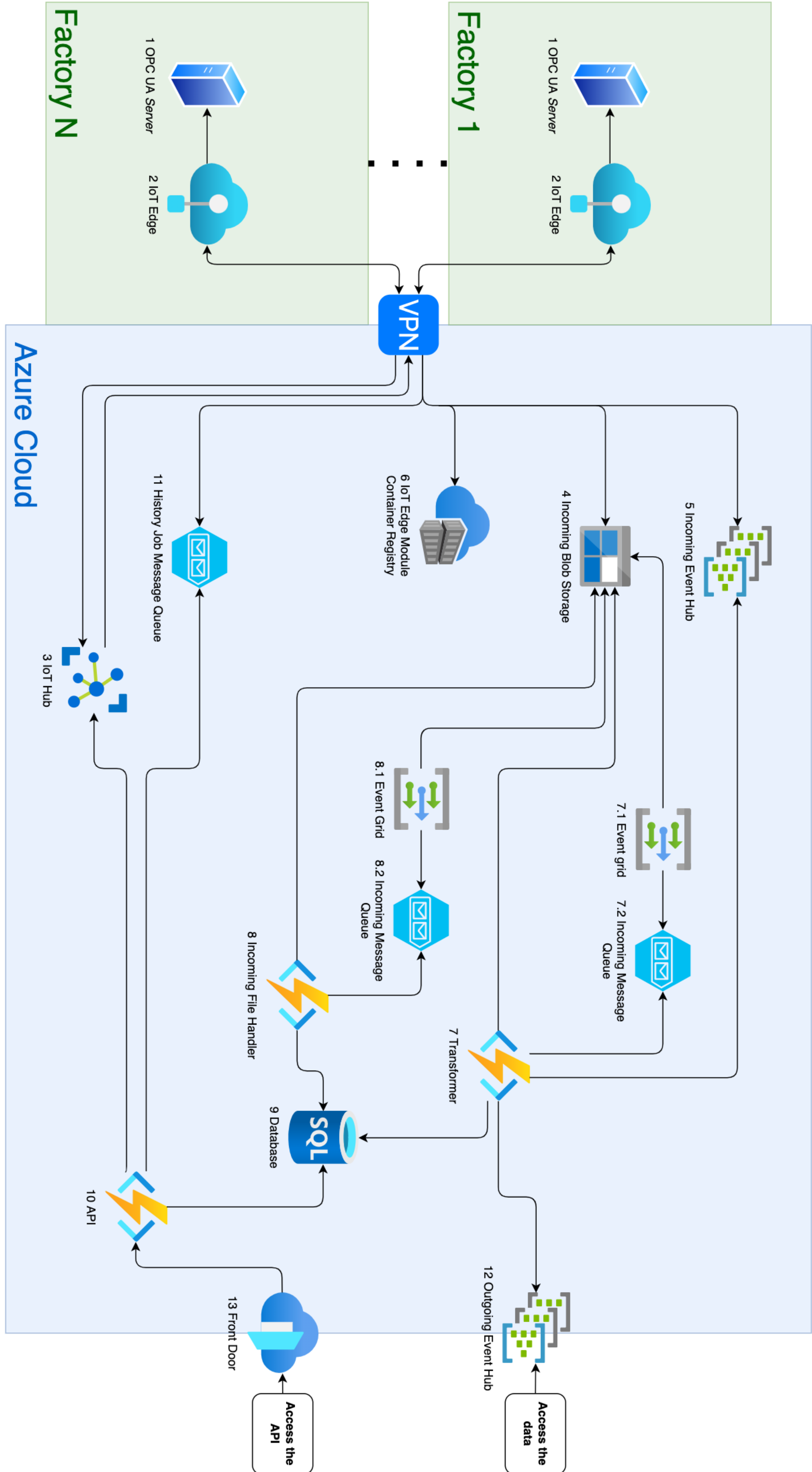

Figure 6.1. *PDI reference architecture.*

Table 6.1. *PDI reference architecture resources and their descriptions.*

N	Name	Description
1	OPC UA *Server*	A data source of PDI which the IoT Edge implementation extracts data from.
2	IoT Edge	An IoT Edge implementation of PDI which consists of multiple modules. Further examined in Chapter 6.2.
3	IoT Hub	An IoT Hub that is utilized to access the edge modules and configure their module twins.
4	Incoming Blob Storage	A Blob Storage that stores the incoming data batches.
5	Incoming Event Hub	An Event Hub that is utilized to stream data from the edge to Azure.
6	IoT Edge Module Container Registry	A Container Registry that contains the edge module images.
7	Transformer	A Function App that transforms the incoming telemetry data to an unified format.
7.1	Event Grid	An Event Grid used in triggering the Transformer.
7.2	Incoming Message Queue	A Queue Storage used in triggering the Transformer.
8	Incoming File Handler	A Function App that handles the incoming data used to update the PDI state.
8.1	Event Grid	An Event Grid used in triggering the Incoming File Handler.
8.2	Incoming Message Queue	A Queue Storage used in triggering the Incoming File Handler.
9	Database	An SQL database that holds the PDI state. The database schema is further examined in Chapter 6.3.
10	API	A Function App that contains an API to view and control the PDI state. Further examined in Chapter 6.5.
11	History Job Message Queue	A Queue Storage that contains history jobs to be executed by the edge implementation.
12	Outgoing Event Hub	An Event Hub that can be used to access the telemetry data in the unified format.
13	Front Door	A Front Door that is used to access the API.

Table 6.1 contains the resources present in the PDI reference architecture that can be utilized to deliver an entity that performs the tasks set for it in Chapter 5. The reference architecture also takes durability into account by suggesting the data pipelines to be built in a durable way as described at the end of Chapter 4.2.2. If the incoming Event Hub or

Blob Storage receives the data, it is guaranteed that the data will be processed at some point even though a function that processes the data is down for some reason.

The Incoming Blob Storage contains multiple containers to be able to differentiate the type of incoming data. It contains separate containers for tags, telemetry and history telemetry. This allows the Event Grids (7.1 and 8.1) to be connected to a certain container type, which makes it possible for different resources to only listen for changes in container they are interested in. The Incoming Blob Storage containers are illustrated in Figure 6.2. The containers contain JSON blobs compressed with gzip for smaller file sizes and their exact JSON schemas are defined in Chapter 6.4. The blobs are named using Universally Unique Identifiers (UUID) which prevents file name collisions.

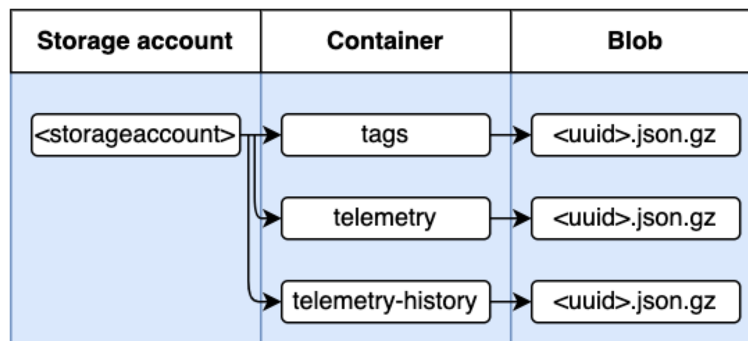

Figure 6.2. *Incoming Blob Storage containers.*

The History Job Message Queue also contains multiple message queues. Each IoT Edge implementation connected to an OPC UA *Server* has its own message queue and the edge implementation listens for the queue for history jobs to execute. This allows separation of history jobs targeted for each data source. The History Job Message Queue queues are illustrated in Figure 6.3. The content of a single history job message is also defined in Chapter 6.4.

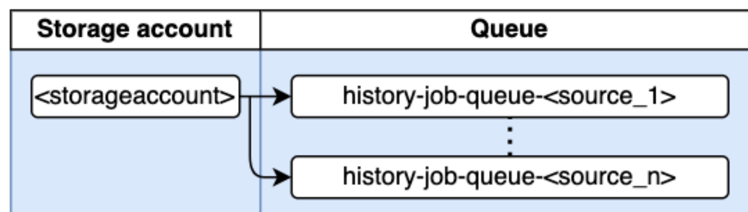

Figure 6.3. *History Job Message Queue queues.*

Using separate containers and message queues to route data and messages gives the resources that consume data (e.g., Function Apps, IoT Edge implementations) a guarantee that the container or message queue they're connected to only contains a blob or message meant for them. Allowing multiple consumers to process data at the same

time by connecting them to different containers or message queues give the architecture the ability to process more data simultaneously.

6.2 IoT Edge architecture

The IoT Edge (2) implementation is responsible for communicating with the data source OPC UA *Servers*. The architecture of the implementation utilizes the modular nature of IoT Edge by separating the OPC UA functionalities into independent modules and each module has their own responsibilities. The OPC UA functionality of the IoT Edge is divided into three parts to achieve the required functionality:

1. Reading the OPC UA *AddressSpace* and discovering the OPC UA *Nodes* that data can be gathered from. From now on, these data nodes will be called "tags" and refer to the OPC UA *Variables* whose values can be read.
2. Managing OPC UA *Subscription*, creating OPC UA *MonitoredItems* and publishing value changes for the subscribed tags.
3. Reading tag history values with OPC UA *HistoryReadRequests*.

The tasks above will be handled by three custom IoT Edge modules called **OpcUaTagFinder**, **OpcUaSubscriptionManager** and **OpcUaHistoryReader**. As they are independent IoT Edge modules, each module is connected to the OPC UA *Server* and manage their own OPC UA *Sessions*. Independent OPC UA *Sessions* enable deploying a custom combination of IoT Edge modules to match the capabilities of the OPC UA *Server*, which corresponds to the modularity part of R2. For example, not every OPC UA *Server* supports OPC UA *HistoryReadRequests* which is why certain IoT Edge implementations might not want to deploy OpcUaHistoryReader. As specified by R1, the OPC UA modules use NodeOPCUA library as their OPC UA *Client* library. The library provides various options when establishing an OPC UA *Client* including the security measures (*OPCUAClientOptions | NodeOPCUA* n.d.). The library can be utilized to ensure compatibility to various OPC UA *Servers* with different configurations and capabilities, which corresponds to the OPC UA compatibility part of R2. Managing the module configurations is described at the end of this chapter after the IoT Edge architecture has been introduced.

In addition to the custom modules, Azure Blob Storage IoT Edge module provided by Azure is also utilized in the IoT Edge architecture. The module is called **EdgeBlobStorage** as it is deployed as an edge module to the IoT Edge runtime. All the OPC UA modules use EdgeBlobStorage to publish their data to Azure as the module can be configured to upload its content to Incoming Blob Storage (4) located in cloud. The OPC UA modules can connect and upload data to EdgeBlobStorage locally and the module will upload its content to cloud over time. This prevents data losses when the connection

to cloud is restricted by storing the incoming data locally. After the connection to cloud is restored, the accumulated data will be uploaded to cloud afterwards. The IoT Edge architecture also contains the system modules defined in Chapter 4.3 called **EdgeAgent** and **EdgeHub**. The architecture containing the IoT Edge modules and their relations to cloud resources is illustrated in Figure 6.4.

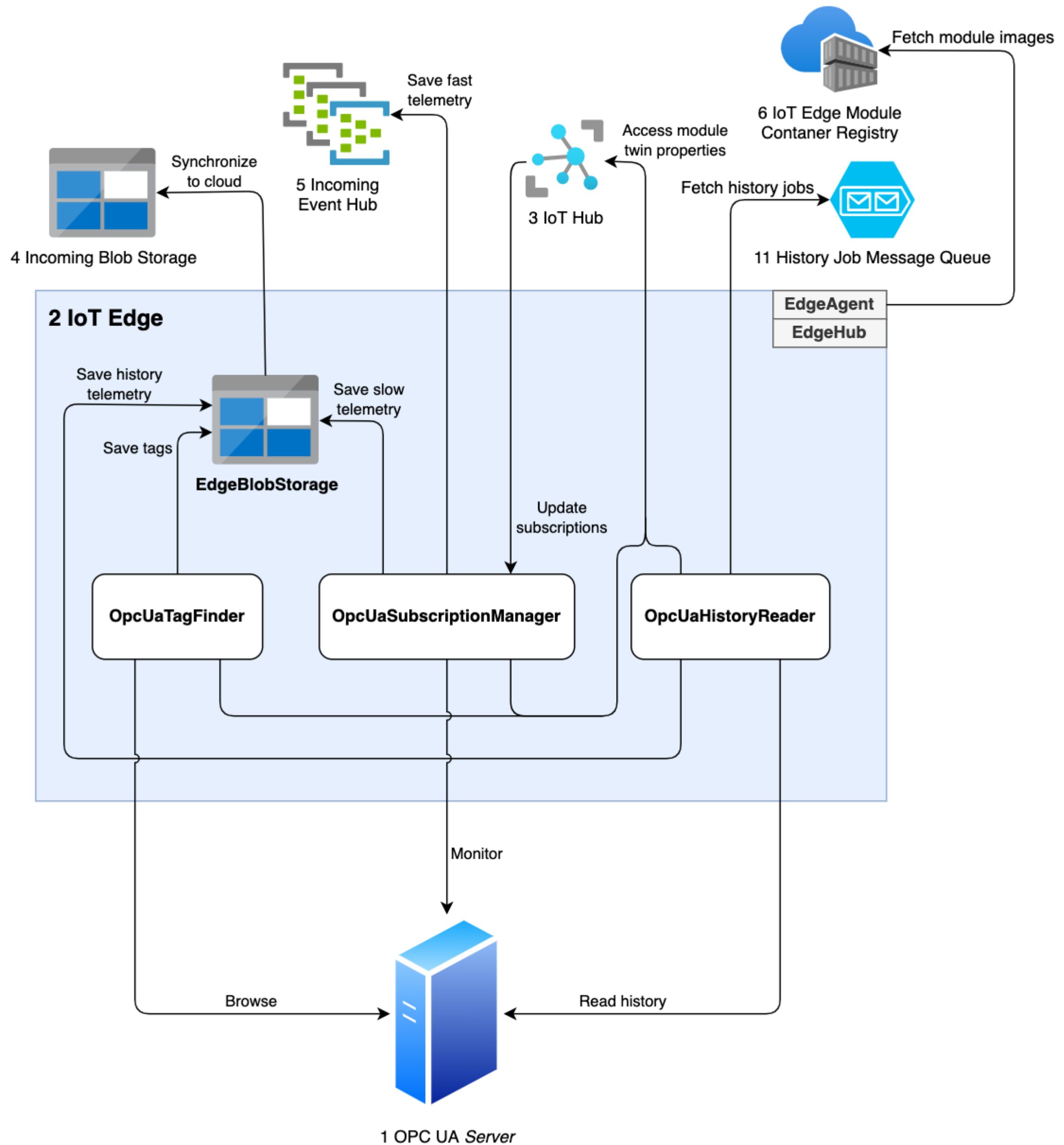

Figure 6.4. *PDI IoT Edge modules and their relations to cloud resources.*

OpcUaTagFinder utilizes OPC UA *Browses* to browse through the OPC UA *AddressSpace* and the browsing is used to produce a list of tags that can be subscribed to. The list of tags is sent to "tags" container of EdgeBlobStorage which uploads the list to cloud. The tag list JSON format is specified in Listing 6.1. The OPC UA *AddressSpace* might contain internal tags that cannot be used to produce value in cloud, which is why not every tag should be published to cloud. OpcUaTagFinder should be configurable in a way that it

can be specified which tags should be included or excluded. At minimum, the root OPC UA *Nodes* that act as starting points for the OPC UA *Browses* should be configurable in order to specify which hierarchies should be included in the produced tag list. To go one step further, a regular expression could be attached to a root *Node* and only the *Browse* results that match the regular expression would be accepted to the final tag list. This can greatly reduce the tag amount in cloud which leads to faster database operations. As OpcUaTagFinder is responsible for filtering and specifying the tags that will be available in cloud, this corresponds to R5 by handling the processing in the edge.

OpcUaSubscriptionManager creates an OPC UA *Subscription* which will be used to monitor tag changes in the OPC UA *Server* using OPC UA *MonitoredItems*. The subscribed tags are managed using direct methods from IoT Hub (3), which can instruct OpcUaSubscriptionManager to add, update or delete *MonitoredItems* in the *Subscription*. State of the *Subscription* should also be persisted in the edge which enables restoring the state on module startup. OpcUaSubscriptionManager utilizes two delivery paths to publish tag value changes: fast and slow. The fast path corresponds to streaming delivery method of R3 by delivering subscription data to cloud using Event Hub (5), which makes the changes accessible in a cloud rapidly by enabling data streaming. The slow path utilizes "telemetry" container of EdgeBlobStorage to publish data to cloud, which fulfills the batch delivery method of R3. The slow path should not send the received data instantly, but to cache them for a while and send the data in bigger batches to decrease the amount of network operations. For example, the fast streaming-based delivery method is best suited for data whose new state should be accessible in cloud as soon as possible and the slow batch-based delivery method is best suited for data that targets analytics and does not require to be delivered instantly to cloud.

Each subscribed tag should contain information about sample rate and the delivery path. Sample rate is used to instruct *MonitoredItems* about wanted sampling intervals, which can be used to reduce the amount of data received from the OPC UA *Server*. For example, every value change of a tag might not be significant, which makes it more useful to receive value changes in a minute intervals. When OpcUaSubscriptionManager receives values changes from the OPC UA *Server*, it uses the delivery path information of a subscribed tag to decide which delivery path the change should be published to. By controlling the OPC UA *Subscription* in the described way and publishing the values with the specified delivery method, OpcUaSubscriptionManager aims to fulfill the data subscription part of R3. The OPC UA *Server* which OpcUaSubscriptionManager is connected to might also yield "bad" values from time to time. In this context, the value being "bad" means that the OPC UA *Subscription* received a null value due to some OPC UA *Server* configuration error. To correspond to R5, OpcUaSubscriptionManager should always validate the incoming data and filter out the bad values so only the valid data is sent to cloud. Some data transformations also take place in the edge,

as OpcUaSubscriptionManager transforms the data sent to cloud to a simple format introduced later in Listing 6.2, which strips many OPC UA specific fields not necessary required in cloud. For example, boolean values can be converted into ones and zeroes before sending the data to cloud.

OpcUaHistoryReader provides the ability to to retrieve history values for tags by executing history jobs from History Job Message Queue (11). History jobs contain information that OpcUaHistoryReader uses to make OPC UA *HistoryReadRequests* to the OPC UA *Server*. At minimum, there must be the tag's identifier (OPC UA *NodeId*) and the timespan to fetch history values from to populate the mandatory fields of a *HistoryReadRequest* (see Listing 6.3). The Queue Storage acting as the history job queue can contain numerous jobs, which OpcUaHistoryReader executes one by one. The history requests are split to history jobs with smaller timespan because the size limits of a single OPC UA response. Splitting the requests into smaller jobs also makes it easier to keep track of the history fetching status since History Job Message Queue contains the jobs yet to be consumed. After a job has been executed and history values have been received from the OPC UA *Server*, OpcUaHistoryReader inserts the values into a cache. Eventually, the cache is sent to "telemetry-history" container of EdgeBlobStorage. Storing the history values in the cache and sending data to cloud in batches corresponds to R5 by reducing the amount of network operations by combining values from multiple OPC UA *HistoryReadRequest* into a single batch. OpcUaHistoryReader marks the job as successfully executed by deleting it from the history job queue. Execution failures cause the job dequeue count to increase and OpcUaHistoryReader can choose to mark the job as failed and move it into a poison queue after a certain amount of retries. OpcUaHistoryReader corresponds to the history part of R3 by processing the history jobs and publishing the history values in a persistent way by utilizing History Job Message Queue. For example, OpcUaHistoryReader can be utilized in a situation where a tag has not been subscribed but its data is required afterwards. If the OPC UA *Server* supports *HistoryReadRequests*, the missing data can be filled afterwards by creating history jobs for the missing data and waiting for OpcUaHistoryReader to consume them.

It was previously stated that the modules manage their independent OPC UA *Sessions* to be able to function independently. The independent *Sessions* can also be exploited to reduce the amount of OPC UA service calls to the OPC UA *Server* as the existing *Sessions* can be used to perform OPC UA service calls, as illustrated in Figure 3.7. This functionality corresponds to R5 (edge computing) by managing the *Sessions* in the on-premise edge environment, which greatly reduces the amount of messages exchanged between the edge modules and the OPC UA *Server* allowing the cloud components to perform OPC UA actions via the modules. For example, OpcUaHistoryReader uses the same *Session* to perform all the OPC UA *HistoryReadRequests* from the history jobs.

As stated before, The OPC UA modules must be compatible with various OPC UA *Server* configurations and capabilities. Deployed modules and their configurations can be configured to match the OPC UA *Server* needs using IoT Edge deployment templates and IoT Hub module twin properties. The deployment template contains the deployed modules and their environment variables for a specific IoT Edge device. Some configuration properties of an edge module should be in the environment variables, but they cannot be changed later without a new deployment. Module twin properties on the other hand can be configured in Azure Portal or using IoT Hub API and can be applied to the modules without new deployments. At least the following properties of all deployed OPC UA modules should be configurable to ensure OPC UA *Server* compatibility: OPC UA *Server* URL, OPC UA *SecurityMode* & *SecurityPolicy* and OPC UA *User Authentication*. In addition to the mentioned properties, the edge modules can utilize additional properties to alter their functionality. This further contributes to the OPC UA compatibility of R2 by providing a method to manage the OPC UA settings.

To address R4 and support multiple OPC UA *Servers* as data sources, each OPC UA *Server* acting as a data source should be connected to their own IoT Edge implementation dedicated to the OPC UA *Server*. The IoT Edge modules are distinguished from each other with unique device-module name combinations, as the module names must be unique within an IoT Edge device. When IoT Edge modules publish data to cloud, they should include their device and module names inside the metadata sent with the actual data. For example, the modules that use EdgeBlobStorage to publish data to cloud should include their identifiers in the metadata fields of a blob as described in Chapter 4.2.1.

6.3 Database schema

This chapter describes the persisted schema in Database (9), which is an SQL database. The schema contains the resource hierarchy and the most important fields in order for PDI to provide the required functionality, but in reality there could be more fields attached to the schema. In short, Database contains the state of PDI, which includes every known IoT Edge device and module in addition to the content produced by the modules and the user-defined data flow configuration. To correspond to R4 and support multiple OPC UA *Server* data sources, Database schema also utilizes the device-module hierarchy from IoT Edge. The content from OPC UA *Servers* is also considered immutable and thus it can only be modified by the IoT Edge modules, which is why an additional layer of user-modifiable information is included in its own table. The schema of Database including the names and relations of the resources is illustrated in Figure 6.5.

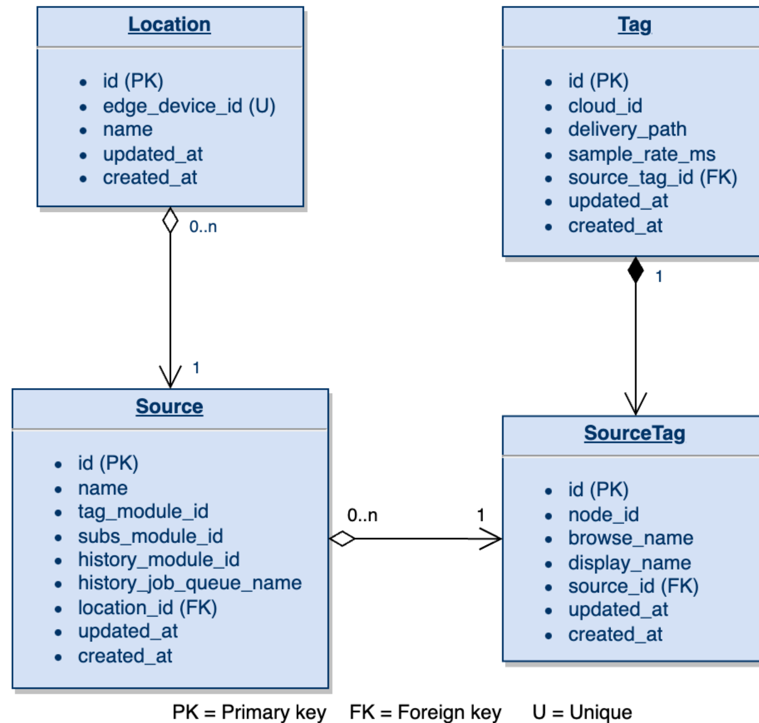

Figure 6.5. *Database (9) schema.*

As illustrated in Figure 6.5, **Location** has the highest rank in the hierarchy. Location represents an IoT Edge runtime and an IoT Edge device within it by specifying the ID of the IoT Edge device (`edge_device_id`). **Source** represents a single OPC UA *Server* and contains the IoT Edge module configuration by specifying the module IDs of the following modules: OpcUaTagFinder (`tag_module_id`), OpcUaSubscriptionManager (`subs_module_id`) and OpcUaHistoryReader (`history_module_id`). In addition to the modules, Source also contain the name of the history job queue (`history_job_queue_name`) in History Job Message Queue (11) in order to insert history jobs into the correct queue meant for the Source. The IoT Edge module IDs of a Source and the IoT Edge device ID of a Location are exploited to identify the source of incoming data, which fulfills R4 by providing a method for identifying multiple OPC UA *Server* data sources from each other (see example in Figure 6.6). A single Location can contain multiple Source rows, which allows multiple OPC UA *Server* specific IoT Edge module combinations to locate within a single IoT Edge device. This further corresponds to R4 as it provides additional flexibility on how and where the IoT Edge modules are deployed as every OPC UA *Server* module combination does not require its own IoT Edge device. In general, Location and Source table rows are manually inserted because of their stationary nature.

SourceTag (Figure 6.5) represents a single tag (OPC UA *Variable*) within a Source (OPC UA *Server*) and it contains the basic OPC UA *Node* information: *NodeId* (`node_id`), *BrowseName* (`browse_name`) and *DisplayName* (`display_name`). SourceTag rows

filtered with a Source's ID (`source_id`) represent all the tags within the OPC UA *Server* that PDI can subscribe to receive data from.

Tag represents a tag that has user-defined subscription information attached to it. Tag row uses SourceTag table information for the tag identification purposes and the information present in Tag table for the subscription details. The subscription information includes the fields required by the subscription IoT Edge module to manage its tag subscriptions: delivery path (`delivery_path`) and sample rate in milliseconds (`sample_rate_ms`). As there is no promise that SourceTag fields are unique between different OPC UA *Servers*, an additional cloud identifier field (`cloud_id`) is also included in Tag. The cloud identifier can be used to provide information for Transformer (7) to which identifier the incoming data should be associated with. For example, multiple OPC UA *Servers* can contain tags (SourceTag) with the same identifiers and a user can map these into different identifier in cloud to separate them from each other.

In conclusion, there is a clear separation of roles between the tables present in the schema. Location, Source and SourceTag tables are used to provide information about the connected OPC UA *Servers* and the possible data nodes within them. Tag table contains all the persisted information in order to fulfill the data subscription part of R3. Tag table should always contain the current data flow configuration that can be displayed to the users in order to provide up-to-date information about the data flow.

6.4 Data flow

This chapter describes the interface data models that various components use to exchange information with each other. At first, the JSON formats of the messages sent to the cloud by IoT Edge (2) and consumed by Transformer (7) and Incoming File Handler (8) are introduced. After the edge-to-cloud JSON formats, the cloud-to-edge JSON format of history jobs between API (10) and IoT Edge is introduced.

The data flow of PDI heavily depends on the IoT Edge device and module ID metadata to identify the OPC UA *Server* where the data is coming from. The device and module IDs are used to query Database (9) (see Chapter 6.3) to resolve the source OPC UA *Server* of the incoming data. The source resolving is illustrated in Figure 6.6.

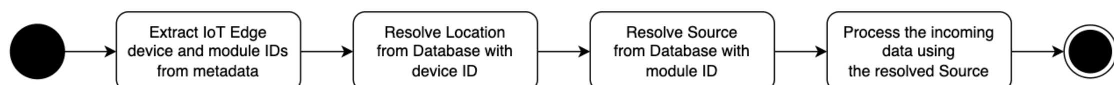

Figure 6.6. *Resolving the source OPC UA Server from Database (9).*

As illustrated in Figure 6.6, the source OPC UA *Server* of the incoming data is done be querying Location table with IoT Edge device ID and Source table with IoT Edge module ID. When the source has been resolved, the component processing the incoming data can

use the information to update the correct rows associated with the source in the database. The source of all incoming data is resolved using the same method which fulfills R4 by allowing to process data from multiple source OPC UA *Servers*.

After OpcUaTagFinder has finished browsing the OPC UA *AddressSpace* and filtered out the unwanted tags, the IoT Edge module uses the format defined in Listing 6.1 to send the produced tag list to Incoming Blob Storage (4). The format contains an array of tags that contains the information used for identification and display purposes. OPC UA *NodeId* (`nodeId`) of a tag is used as the identifying information and the tag's *BrowseName* (`browseName`) and *DisplayName* (`displayName`) can be used for display purposes as additional information in addition to the *NodeId*.

```
[
    {
        "nodeId": <string>,
        "browseName": <string>,
        "displayName": <string>
    },
    ...
]
```

Listing 6.1. *JSON format of a tag list produced by OpcUaTagFinder.*

Incoming File Handler (8) reads the tag list blobs from Incoming Blob Storage (4) and uses the blob contents to populate SourceTag table in Database (9). Incoming File Handler extracts the IoT Edge device and module IDs from the blob metadata and resolves the source OPC UA *Server* as illustrated in Figure 6.6. After the source has been resolved, Incoming File Handler uses the source information to populate and update the SourceTag rows associated with the source.

When OpcUaSubscriptionManager receives value changes from the OPC UA *Subscription*, the module publishes the changes in the format defined in Listing 6.2. The format contains an array of the bare minimum fields for associating telemetry values with a tag: OPC UA *NodeId* (`nodeId`), value (`value`) and timestamp of the value (`timestamp`). The format is the same in both, fast and slow, delivery paths. As for OpcUaHistoryReader, the same format is also used when publishing history values to Incoming Blob Storage.

```
[
    {
        "nodeId": <string>,
        "value": <number>,
        "timestamp": <string>
    },
    ...
```

```
]
```

Listing 6.2. *JSON format of telemetry produced by OpcUaSubscriptionManager and OpcUaHistoryReader.*

Transformer (7) receives telemetry from Incoming Blob Storage and Incoming Event Hub (5). As Transformer has access to Database, it only requires the *NodeId* in addition to the IoT Edge device and module IDs from metadata to be able to identify the OPC UA *Server* where the incoming telemetry is from and find the correct SourceTag row using the resolved data source information. When the tag has been resolved using SourceTag table, Transformer can further utilize the information present in Database to perform the transformation to the unified output schema.

History Job Message Queue (11) contains history jobs for OpcUaHistoryReader to consume and perform OPC UA *HistoryReadRequests*. The history job format is defined in Listing 6.3 and it contains the OPC UA *NodeId* (`nodeId`) and the timespan (`from` & `to`) to request history data from.

```
{
    "nodeId": <string>,
    "from": <string>,
    "to": <string>
}
```

Listing 6.3. *JSON format of a history job consumed by OpcUaHistoryReader.*

The history jobs are inserted to History Job Message Queue by API (10) to indicate that a user has requested history data to be read from specific tags. As illustrated in Figure 6.3, every source has its own history job queue. OPC UA *NodeIds* are OPC UA *Server* specific which is why history jobs must be inserted to the correct queue associated with the OPC UA *Server* where the history data should be read from.

6.5 API

API (10) aims to fulfill R6 by providing a management web API to manage the functionality of PDI. API can be used to request the current state of PDI to provide information about the source OPC UA *Servers*, which abstracts the underlying OPC UA *Servers* in a way that the end-user does not require any knowledge about the server-specific settings, such as URLs and security measures. In addition to the current state, API can be used to control the data flow from IoT Edge (2) to cloud. In order to provide the required services, API utilizes IoT Hub (3), Database (9) and History Job Message Queue (11) to perform the required operations. To invoke API services, the requests to it should be made through Front Door (13) that is exploited to access API. The API description does not take the

actual implementation of API into account as the description only contains the services that the management API should provide. For example, API could be implemented as REST or GraphQL API as both can be utilized in providing the required functionality. API services and their brief descriptions are displayed in Table 6.2.

Table 6.2. *API (10) services and their descriptions.*

N	Service	Description
1	Get locations	Returns the locations from Database (Location table).
2	Get location by ID	Returns the location with the specified ID from Database (Location table).
3	Get sources	Returns the OPC UA *Servers* from Database (Source table).
4	Get source by ID	Returns the OPC UA *Server* with the specified ID from Database (Source table).
5	Get source tags	Returns source tags from Database (SourceTag table). Used mainly for returning the source tags associated with a specific OPC UA *Server*.
6	Get source tag by ID	Returns the source tag with the specified ID from Database (SourceTag table).
7	Get tags	Returns tags from Database (Tag table). Used mainly for returning the tags associated with a specific OPC UA *Server*.
8	Get tag by ID	Returns the tag with the specified ID from Database (Tag table).
9	Create tag	Creates a new tag to Database (Tag table). Invokes the tag subscription direct method via IoT Hub if the tag should be added into the edge subscription. See example in Figure 6.9.
10	Edit tag	Updates an existing tag to Database (Tag table). Invokes the tag subscription direct method via IoT Hub if the tag should be updated into the edge subscription.
11	Delete tag	Deletes an existing tag from Database (Tag table). Invokes the tag subscription direct method via IoT Hub to delete the tag from the edge subscription.
12	Request tag history	Creates history jobs for the requested tags for the specified timespan. The history jobs are inserted into History Job Message Queue.

The first half of API's services consist of providing the OPC UA data stored in Database. API provides the locations with IoT Edge devices, the sources (OPC UA *Servers*) with IoT Edge modules associated to them and the source tags within the sources. These resources are updated either manually or by IoT Edge and cannot be modified via API. The second half of the services correspond to the subscription part of R3 by providing methods for controlling the data flow. Tags extended from the source tags are used to define the data flow configuration and API provides the basic functionality to manage

them: creating, editing and deleting. When a tag has been modified, these services utilize IoT Hub to forward the changes to IoT Edge and OpcUaSubscriptionManager (see Chapter 6.6.3 for an example). The history part of R3 is corresponded by the history request service, which allows requesting history values for tags for a certain time period. API creates the history jobs based on the request and inserts them into History Job Message in the format specified in Listing 6.3, which will later be consumed by IoT Edge and OpcUaHistoryReader (see Chapter 6.6.5 for an example).

6.6 Feature review

This chapter goes through the features provided by the PDI reference architecture by combining the previously described functionalities into complete features in order to fulfill the requirements from Chapter 5. By providing the features described in the following subchapters, the PDI architecture should form a complete entity that provides all the functionalities set for PDI. The features consist of fetching the source tags from the OPC UA *Servers* into cloud, viewing the current state of PDI, subscribing for tag value changes in order to receive telemetry from the OPC UA *Servers*, processing the previously mentioned telemetry and reading history data from the OPC UA *Servers*.

6.6.1 Populating the source tags

Populating the source tags from the source OPC UA *Servers* into SourceTag table in Database (9) is the step in the process of extracting data from the OPC UA *Servers* into cloud. Populating the source tags utilizes IoT Edge (2), Incoming Blob Storage (4) and Incoming File Handler (8) in order to populate the source tags into Database which makes them available for further usage. The feature itself does not directly correspond to any of the requirements but acts as a prerequisite to the features described in the later subchapters. Figure 6.7 illustrates the flow of populating source tags from a single OPC UA *Server*. If there is multiple OPC UA *Servers* with OpcUaTagFinder IoT Edge modules, the source tag population is independently executed for all the OPC UA *Servers*.

As illustrated in Figure 6.7, the source tag population process starts with OpcUaTagFinder starting to browse the OPC UA *Server AddressSpace* from the selected OPC UA root *Nodes* defined in the module configuration. Every tag under the roots that matches the tag filters defined in the module configuration is added into a temporary tag list, which will later be used to produce the source tag list that will be sent to EdgeBlobStorage IoT Edge module in a format defined in Listing 6.1. EdgeBlobStorage uploads the source tag list into Incoming Blob Storage where it will be accessible for Incoming File Handler. When Incoming File Handler receives a fresh source tag list, it resolves its source OPC UA *Server* using the method illustrated in Figure 6.6. Contents of the source tag list is

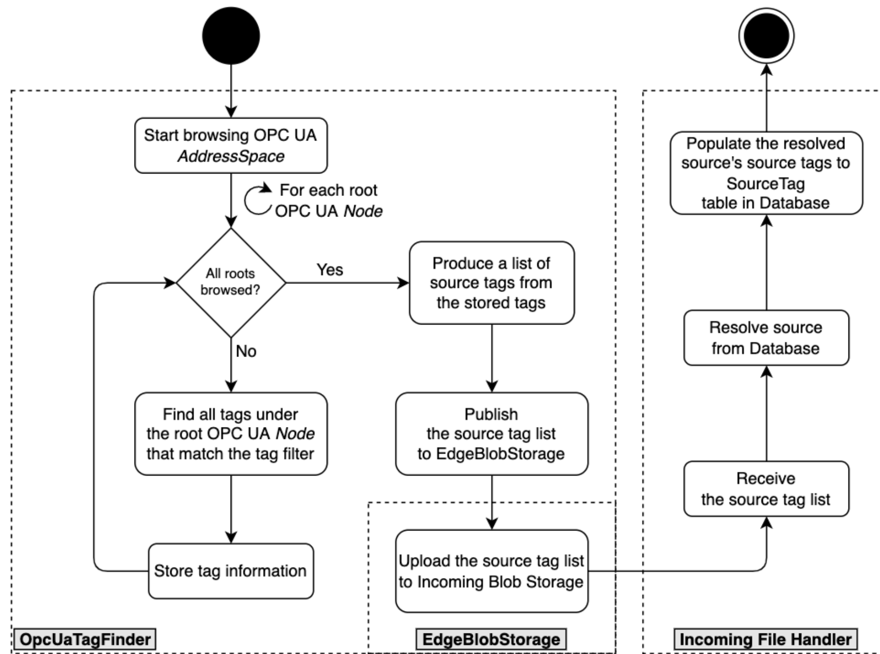

Figure 6.7. *Flow of populating OPC UA Server's tags into Database.*

then inserted into SourceTag table in Database using the resolved source information for every source tag included in the list.

6.6.2 Viewing the current state

To be able to control the data flow, one has to have knowledge about the current state of the data flow. This feature utilizes API (10) to request the state from Database (9) which can then be displayed to the end-user, which corresponds to R6 by providing the available resources and the data flow configuration. The following list contains the resources that the current state consists of and the API services (see Table 6.2) that can be used to obtain their state:

- Locations / IoT Edge devices (API: 1-2)
- Source OPC UA *Servers* (API: 3-4)
- Source tags in the source OPC UA *Servers* (API: 5-6)
- Tags that hold the data flow configuration (API: 7-8)

Fetching the mentioned resources can be done step-by-step, which allows fetching the resources that relate to another resource upper in the hierarchy (illustrated in Figure 6.5). For example, the source tags that belong to a source OPC UA *Server* can be requested from API after the available source OPC UA *Servers* are requested and one of them is specified in the source tag request. The flow of requesting the current state is illustrated in Figure 6.8.

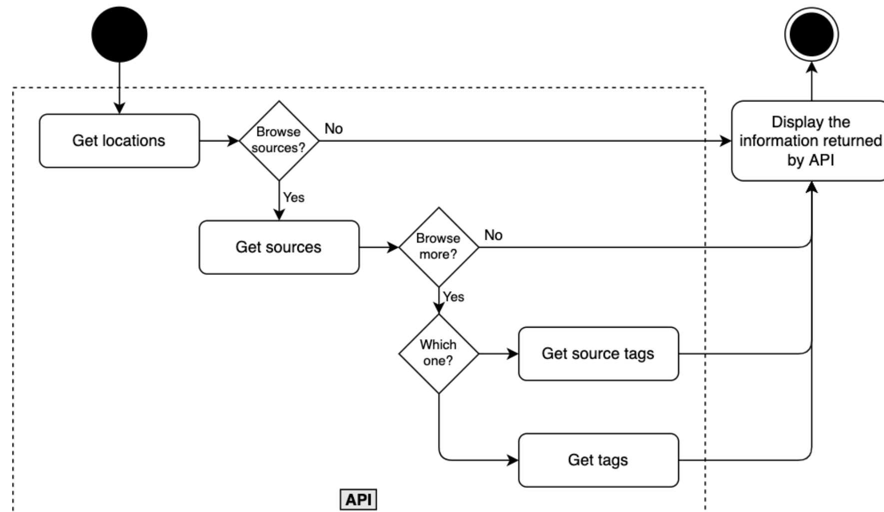

Figure 6.8. *Flow of viewing the current PDI state.*

Figure 6.8 illustrates that the first resources that should be requested are the location that contain the IoT Edge devices. This provides the client who made the request the locations, which can then be used to request the source OPC UA *Servers* that belong to them. When the OPC UA *Servers* are known, the available source tags or the tags with the data flow configuration can be requested. The source tags provide information about every tag in the OPC UA *Server* than can be subscribed to and the tags provide the current data flow configuration since they contain the tags that are present in the OPC UA *Subscription* in IoT Edge (2). The feature partially corresponds to the subscription part of R3 by providing the current data flow configuration with the tags and the available source tags that can be added to the OPC UA *Server's Subscription* (creating a new subscription is described in Chapter 6.6.3). The tags can also be used to request history data from the OPC UA *Server*, which partially corresponds to the history part of R3. History feature is further described in Chapter 6.6.5.

6.6.3 Creating a tag subscription

Data subscription feature of PDI enables managing the tag subscriptions in the edge OPC UA modules thus giving the ability to control the incoming data flow. The feature aims to fulfill the data flow control part of R3 by providing the required data flow management features. The feature consists of three services: API (10), Database (9), IoT Hub (3) and IoT Edge (2). The API service for new tag subscriptions (9 in Table 6.2) can be invoked to trigger the subscription flow, which in the end updates the OPC UA *Subscription* and starts producing data for the subscribed tag. The flow of updating subscription for a tag is illustrated in Figure 6.9.

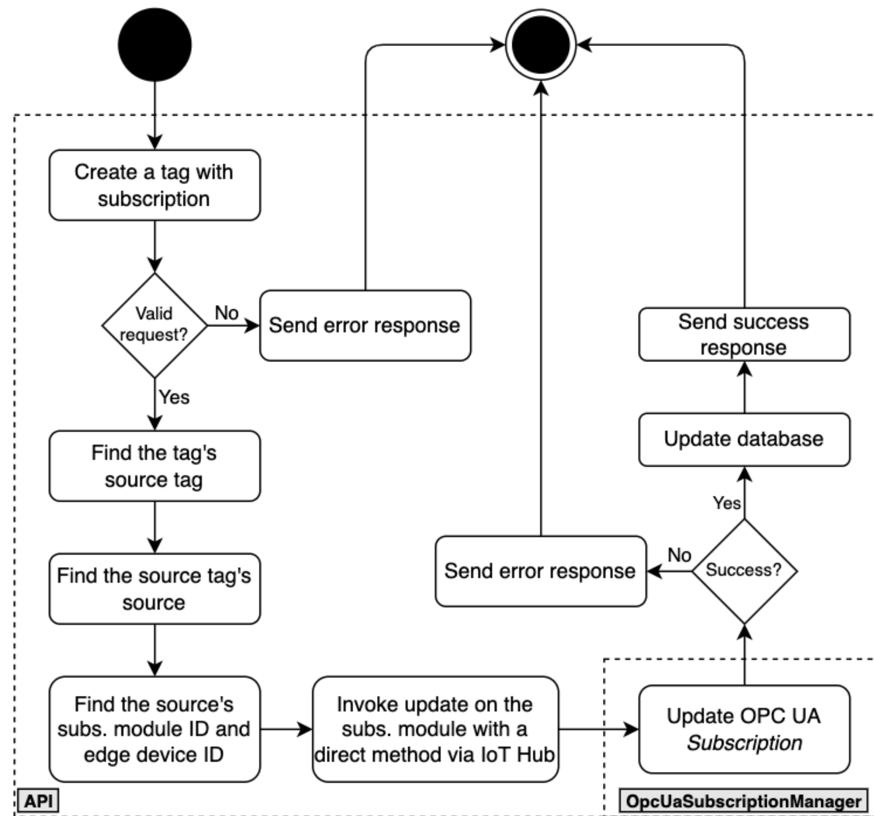

Figure 6.9. *Flow of creating a tag subscription.*

The flow starts by invoking the tag subscription API endpoint. First, the endpoint validates the request by checking that the request contains all the required information for creating a new tag from a source tag with valid subscription configuration. If the validation succeeds, SourceTag, Source and Location tables are used to resolve the information required to route and invoke a direct method that updates the OPC UA *Subscription* on the OpcUaSubscriptionManager of the source: OPC UA *NodeId* of the tag, edge device ID and subscription module ID. API makes a request to IoT Hub that is utilized to route the direct method to the OpcUaSubscriptionManager of the tag's source located in IoT Edge using the device and module IDs. OPC UA *NodeId* of the tag, sample rate and delivery path are included in the direct method request body which contains the actual information required to update the OPC UA *Subscription*. The direct method response contains information whether the update was successful, the tag subscription is updated into Database and API responds with a success response. If there was a failure in any step of the flow, API responds with an error response.

6.6.4 Processing telemetry data

When a tag in one of the source OPC UA *Servers* is successfully subscribed to (see Chapter 6.6.3), the OPC UA *Server* starts producing telemetry for the said tag. The

feature aims to provide a method for processing the telemetry produced by the source OPC UA *Servers*. As the telemetry can originate from various OPC UA *Servers*, the telemetry must be processed in a way that corresponds to R4 by supporting multiple OPC UA data sources. The telemetry processing utilizes IoT Edge (2), Incoming Blob Storage (4), Incoming Event Hub (5), Transformer (7), Database (9) and finally Outgoing Event Hub (12). Figure 6.10 illustrates how telemetry is processed after receiving it from an OPC UA *Subscription*.

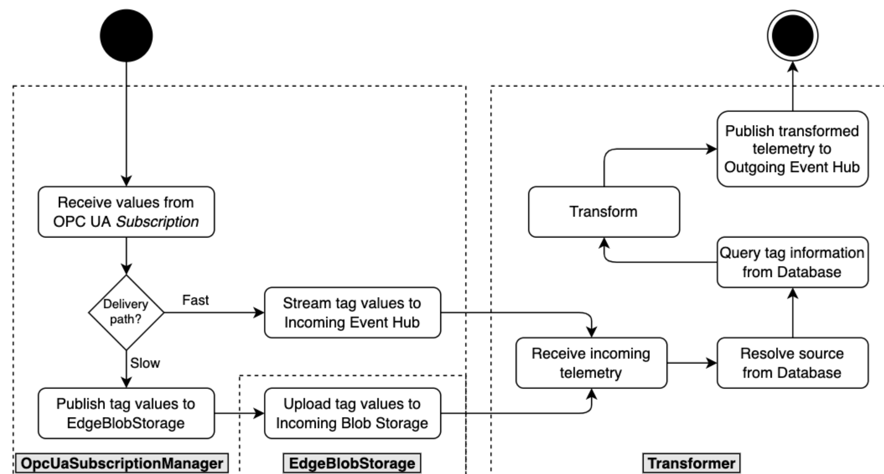

Figure 6.10. *Flow of processing OPC UA telemetry data.*

Processing of the received telemetry starts when OpcUaSubscriptionManager IoT Edge module receives a tag value update from the OPC UA *Subscription*. When a new value is received, OpcUaSubscriptionManager decides which delivery path the tag belongs to from the tag configuration and forwards the value to the delivery path specified for the tag in the format specified in Listing 6.2. In case of the fast path, the new value is streamed into Incoming Event Hub. The slow path sends the value to EdgeBlobStorage which uploads the value into Incoming Blob Storage. Either way, Transformer receives the incoming telemetry and starts to process it. As the telemetry can be from various OPC UA *Servers*, the source is resolved using the method illustrated in Figure 6.6. After the source has been resolved, Transformer can query the data flow configuration for the source from Database using the received OPC UA *NodeId* of the incoming telemetry. The data flow configuration for the received source tag is located in Tag table, which contains the information available for Transformer to use in the transformation, mainly the unique cloud identifier to distinguish the same OPC UA *Node* identifiers between different sources from each other. As stated before, the unified data schema used by Transformer is intentionally left undefined as it is heavily application dependent. However, Transformer performs the transformation to the application specified data schema using the data flow configuration that is persisted in Database. The transformed telemetry will be forwarded to Outgoing Event Hub, which provides a method for the business applications built around PDI to access the transformed telemetry data extracted from the various source OPC UA

Servers and transformed into the application specific data schema.

6.6.5 Reading history data

The features described in the previous subchapters have mainly corresponded to the subscription part of R3, leaving the history part without attention. This feature gives the functionality to fulfill the history part of R3 utilizing API (10), Database (9), History Job Message Queue (11), IoT Edge (2) and Incoming Blob Storage (4). The main concept for reading history data from the source OPC UA *Servers* consists of creating history jobs for a tag using API and OpcUaHistoryReader asynchronously consuming the jobs and producing the history values. The flow of reading history data from a source OPC UA *Server* is illustrated in Figure 6.11.

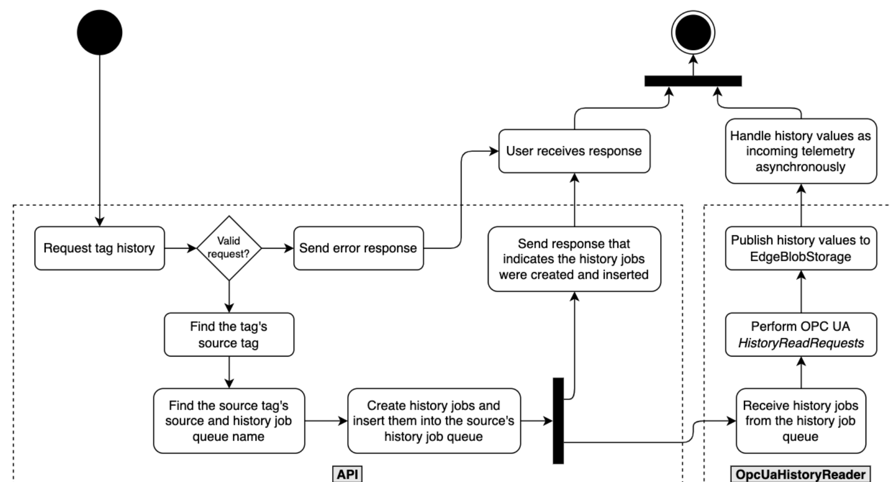

Figure 6.11. *Flow of reading OPC UA history data.*

As illustrated in Figure 6.11, the process of reading history for a tag starts by invoking the tag history request service (12 in Table 6.2). The request is then validated and a valid request results in the request being splitted into multiple history jobs with smaller timespans. The tag whose history is requested is also used to resolve the source and queue name in History Job Queue of the target OpcUaHistoryReader that will be responsible for processing the history jobs. When the history job queue is resolved and the history jobs are inserted to the queue, API provides a response that indicates that the jobs were successfully inserted to History Job Message Queue. OpcUaHistoryReader queries the history job queue assigned to it to receive information about incoming history jobs. When OpcUaHistoryReader notices that there are history jobs to process, the module starts executing OPC UA *HistoryReadRequests* based on the received jobs. The results from the requests are inserted into EdgeBlobStorage which uploads the received history data to Incoming Blob Storage. When the history data is uploaded into Incoming Blob Storage, the history data is treated as incoming telemetry the same way as telemetry from subscription as illustrated in Figure 6.10.

7. RESULT ANALYSIS

The results of the thesis that aim to solve the research questions specified in Chapter 1.1 are described in Chapter 6 in the form of PDI reference architecture. The research questions and Insta's requirements (Appendix A) were formed into six requirements (R1-R6) in Chapter 5, which the PDI architecture aims to solve thus providing answers to the research questions.

As the PDI reference architecture only focused on the previously mentioned six requirements, this chapter analyzes how the produced reference architecture can be traced back to satisfy the research questions. The produced architecture is also compared to the two reference OPC UA web integrations introduced in Chapter 3.5: OPC UA Web Platform by Cavalieri et al. (2019) and REST-based OPC UA Middleware by Habib et al. (2022). When addressing the research questions, the analysis also contains some downsides of the produced architecture compared to the other reference systems.

Before addressing the research questions, it should be noted that during the literature review no OPC UA to cloud integration with as much detail as the PDI reference architecture was discovered. The discovered OPC UA to cloud integrations addressed the topic on much higher abstraction level compared to the PDI reference architecture and many implementation details were left unpresented. The scope of the PDI reference architecture is also much larger than the discovered reference integration systems, which is why the comparison between the PDI architecture and the discovered reference systems are applied to the sections of the PDI architecture where the reference systems provide similar functionality. Also for clarity reasons, OPC UA Web Platform and REST-based OPC UA Middleware architectures are referred as "reference architectures" and the PDI reference architecture is only referred as "PDI architecture".

7.1 Architecture and required components

The first research question (RQ1) aims to solve which software components are required in order to integrate data from OPC UA *Server* to cloud, specifically to Azure. Chapter 6 introduces the PDI architecture with the high-level goal of integrating the data from OPC UA *Servers* into Azure, which provides an answer to the research question. As mentioned at the start of his chapter, the produced PDI architecture provides a detailed description

of the required components. The architectures by Cavalieri et al. (2019) and Habib et al. (2022) do not describe the required components in such detail which makes comparing them to the PDI architecture difficult.

The PDI architecture consists of the components specified in Table 6.1. Similarly to the two reference systems, the components of PDI can be roughly divided into two categories: asynchronous data flow components and synchronous API components for controlling the data flow. In addition to the previous categories, the edge components of the PDI architecture could also be considered as own category even though they are part of the data flow since they provide additional functionality layer compared to the two reference systems. The benefits of edge computing are further described in Chapter 7.4.

OPC UA Web Platform uses message brokers to handle asynchronous communication from the OPC UA *Servers* and gives roughly the same functionality as Incoming Event Hub (5) of the PDI architecture. REST-based OPC UA Middleware sends data to cloud using HTTP requests. In addition to the broker approach that utilizes data streaming, the PDI architecture also contains a slower and cheaper Incoming Blob Storage (4) approach that sends data from the OPC UA *Servers* to cloud in batches. The benefits of having multiple data delivery methods is further described in Chapter 7.3.

The API functionality of the PDI architecture focuses on controlling the data flow by controller the edge functionality, whereas the two reference systems map HTTP requests to OPC UA *Requests* directly. API (10) of the PDI architecture is implemented as an Azure Function App, which is also invoked by HTTP requests. Mapping the HTTP requests into OPC UA functionalities is a far more straightforward approach thus being easier to develop and maintain. On the other hand, IoT Edge component of the PDI architecture can be utilized to provide an additional layer of functionality to the data flow. Every architecture in the comparison maintains active OPC UA *Client Sessions* that are used to execute the requests coming from the API components. Exploitation of the previously established OPC UA *Sessions* greatly reduces the amount of required OPC UA *Requests* to fulfill the API requests as illustrated in Figure 3.7. As the PDI architecture handles the OPC UA communication between the OPC UA *Server* and the OPC UA *Client* located in the edge, it can also gain benefit from the decreased latencies from edge computing.

Even though the PDI architecture provides more features than the architectures by Cavalieri et al. and Habib et al., the more complex architecture causes some drawbacks. The existence of multiple logic and storage components results to more complex setup and development processes compared to the two reference systems. For example, the on-premise IoT Edge setup is an extra setup step compared to the two reference systems. Because of the complexity, the time required to setup and develop the system can also be much longer.

7.2 OPC UA compatibility

The second research question (RQ2) is about ensuring that the PDI architecture is compatible with variety of OPC UA *Server* configurations to be able to extract data from them only by configuring the implementation. The PDI architecture exploits IoT Edge (2) module configurations to ensure the compatibility to variety of OPC UA *Servers* with various *AddressSpace* models, security measures and capabilities. As described in Chapter 6.2, OPC UA *Server* URL, OPC UA *SecurityMode* & *SecurityPolicy* and OPC UA *User Authentication* should be configurable for every IoT Edge module to ensure the connectivity. The settings are passed to NodeOPCUA *Client* as specified in Appendix A. Module configurations can be managed by setting environment variables in IoT Edge deployment templates during deployments or by setting IoT Edge module twin properties. The module twin properties provide flexible module configurations as they can be configured in Azure Portal and the IoT Edge modules can apply the new twin properties without a need for new deployment. REST-based OPC UA Middleware only utilizes the OPC UA *Server* endpoint URL, which excludes the OPC UA security measures. If an OPC UA *Server* only supports signed or encrypted communication, REST-based OPC UA Middleware would be unable to connect to the OPC UA *Server*. OPC UA Web Platform on the other hand mentions the OPC UA security measures in the paper, but does not specify on how they should be configured to the OPC UA *Clients*.

OPC UA *Server* capabilities are also tied to the IoT Edge modules. The modules deployed for each OPC UA *Server* are defined by the capabilities of an OPC UA *Server*. For example, only the OpcUaTagFinder and OpcUaSubscriptionManager modules should be deployed for an OPC UA *Server* if no history data is required or available. The modules for each source OPC UA *Server* are also available via API (10), which provides the end-user information about the capabilities of a wanted OPC UA data source based on the deployed IoT Edge modules.

OpcUaTagFinder module is responsible for OPC UA *AddressSpace* compatibility by providing an abstraction of the available tags for each OPC UA *Server*. The module browses through the OPC UA *Server AddressSpaces* and produces a list of available source tags that can be accessed via API. The two reference systems map OPC UA *Browses* into HTTP requests, which makes the changes in *AddressSpace* models available instantly. Even though *AddressSpace* model changes would be infrequent, the PDI architecture requires OpcUaTagFinder to form the source tag list again to detect the changes. An OPC UA *AddressSpace* can contain an enormous amount of *Nodes*, which can become a problem with REST-based OPC UA Middleware since it reads all objects from the *AddressSpace* in intervals, making it less compatible with large OPC UA *Server AddressSpaces*. In future, the PDI architecture's source tag list abstraction could also be utilized to abstract tags from data sources. The current source tag model could be

extended to contain identifying information from other sources with minor changes. Other data sources than OPC UA would also require new IoT Edge modules to be compatible with the new data sources.

The current OPC UA connectivity of the IoT Edge modules exploits the more mature client-server model. In future, the publish-subscribe approach could also be considered when it gains maturity. Changing the OPC UA communication model to publish-subscribe would only require changes in the IoT Edge modules, leaving the rest of the PDI architecture untouched. This is possible because there is clear abstraction between the cloud infrastructure and the OPC UA connectivity of the IoT Edge modules, which would allow the new modules that use publish-subscribe to utilize the already existing communication methods to the cloud infrastructure. According to Aro (2021, p. 13), the publish-subscribe does not support OPC UA history features. Because of this, OpcUaHistoryReader IoT Edge module would still be required to perform OPC UA *HistoryReadRequests* and only the subscription modules could be converted to support the publish-subscribe communication model.

7.3 Data flow control

The third research question (RQ3) is about controlling the incoming data flow. The data flow can be divided categories by Industrial Big Data characteristics (five V's) presented in Chapter 2.2. The five V's are examined below from the viewpoint of the PDI architecture and the reference architectures by Cavalieri et al. and Habib et al.

"Value" characteristic is closely connected to the decision of what data should be collected from all data, as it refers to the ability to turn the data to actual business value. The PDI architecture allows the end-user to choose what data to collect from each of the source OPC UA *Servers*. OpcUaTagFinder has produced a list of available source tags for every OPC UA *Server* connected to PDI, which the end-user can request from API (10). From the available source tags, the end-user can choose which tags to subscribe for and invoke the API method for creating a new tag subscription (9 in Table 6.2). The new tag can be from any of the connected OPC UA *Servers* as the OPC UA *Server* where the tag originates from is resolved using the method illustrated in Figure 6.9. OPC UA Web Platform also provides a clear method of choosing the collected data by stating the wanted data set (OPC UA *Server*) and the OPC UA *NodeIds* in the POST request to `/data-sets/dataset-id/ monitor`. REST-based OPC UA Middleware however is used to collectively extract all the data from the OPC UA *Server*, leaving no room for the end-user to choose which OPC UA *Nodes* the data is collected as the cloud platform storing the data requests values for every object in the OPC UA *Server*. Some OPC UA *Servers* might have enormous amounts of OPC UA *Nodes* which makes scalability an issue with the collective approach. Also, no attention is paid to whether the data in the

OPC UA *Server* is actually meaningful from a business viewpoint. The two reference architectures only support subscription data, which gives an advantage to PDI. If the wanted data was not collected at the time, the data can not be collected via the reference systems afterwards which puts enormous pressure on reliability of the reference systems. If a tag has not subscribed before but its data is still required, the PDI architecture allows the end-user to request the wanted tag's history data afterwards using the feature described in Chapter 6.11, which is a requirement specified in Appendix A.

"Velocity" characteristic is about how quickly the data can be utilized after receiving it. Some applications only require big data batches from time to time, whereas some depend on the current data to be received in almost real-time. The PDI architecture aims to solve the problem by providing two different delivery methods: the slow path for data batches that utilizes Incoming Blob Storage (4) and the fast path for data streaming using Incoming Event Hub (5). The Azure Blob Storage based slow path provides a method for combining data received from the source OPC UA *Servers* into batches and transferring them into cloud for further processing and storing. For data that should be delivered to the receiving application quickly, the Azure Event Hub based fast path can be utilized to stream the data to cloud as soon as it is received from the source OPC UA *Servers*. For example, the slow path can be utilized for the data that targets analytics which requires big data volumes and the fast path for the data that targets some situation awareness applications. The PDI architecture utilizes API (10) to instruct the OpcUaSubscriptionManager module of IoT Edge (2) to decide the wanted delivery path for every subscribed tag, which is illustrated in Figure 6.10. "Velocity" is also somewhat connected to the "volume" characteristic since the data sample rate is closely connected on how fast the changes from the OPC UA *Servers* are received. The two reference architectures only provide their standard way for delivering the data, as OPC UA Web Platform uses streaming via a message broker and REST-based OPC UA Middleware reads data from the OPC UA *Server* into a database where it can be queried.

"Volume" characteristic is partially handled by the fact that more and more data is extracted from the source OPC UA *Servers* as the time goes on. However, the amount of data produced in a certain timespan can differentiate based on the sample rates. The PDI architecture requires that the sample rate is specified for each tag subscription, which can be used to configure the data volume sent to cloud. This also somewhat relates to the "value" characteristic as not every data point is required if the data is used to build a bigger picture. The independent sample rate for each tag can be utilized to gain frequent values for the tags that benefit from it, but also leaves room for the tags that do not require such frequent values. OPC UA Web Platform also requires sample rate to be included in the monitoring request, whereas REST-based OPC UA Middleware requests every value from the OPC UA *Server* in a predefined interval.

"Variety" characteristic is mainly handled by the source OPC UA *Servers*. OPC UA

AddressSpace, which the OPC UA *Servers* use to represent their content to OPC UA *Clients*, is a standardized model of the underlying system that can contain objects from heterogeneous sources. The PDI architecture and the two reference systems do not have to pay attention to the heterogeneity of the underlying data sources as long as the data is received via OPC UA communication. However, the PDI architecture provides an additional layer of interoperability with other data sources as it specifies Transformer (7) that can transform the incoming data to a format compatible with other systems. If PDI is just one data source in a larger context, it may be beneficial to transform the incoming data to an unified data schema required by the system that ingests data from PDI. For example, the transformed data can contain custom cloud identifiers (`cloud_id` in Figure 6.5) that can be used to differentiate the data produced by PDI from other systems.

"Veracity" refers to the validity and truthness of data as there can be some unreliability in data sources. None of the analyzed architectures take the validity of the incoming data into account and rely on the received data actually being true. If the data received from the OPC UA *Servers* contain false measurements, the false data is treated the same way as any received data by the systems. Eventually it is up to the end-user to recognize the false measurements since none of the analyzed systems provide any help on that matter.

7.4 Edge computing

The fourth research question (RQ4) investigates how the PDI architecture can exploit edge computing before sending data from the source OPC UA *Servers* into cloud. From all of the reference architecture, the PDI architecture is the only one that mentions the presence of edge computing within its architecture. The edge computing takes place within the IoT Edge (2) modules that are deployed for each source OPC UA *Server*. How the PDI architecture utilizes edge computing in its data flow is described below.

When OpcUaTagFinder browses an OPC UA *AddressSpace* in order to produce the list of available tags, the OPC UA *Browse* operations can produce lots of tags that serve no purpose in cloud. OpcUaTagFinder can be configured to only include tags that are located under specific root OPC UA *Nodes*, which can be used to exclude unnecessary folders from the OPC UA *AddressSpace*. If the tags in the *AddressSpace* have a consistent naming scheme, a regular expression could also be applied to the tags. For example, if all the OPC UA *Server's* internal tags are prefixed with the same string, they could all be excluded from the produced source tag list by exploiting the regular expression. Browsing through the *AddressSpace* can require lots of OPC UA *Browse* operations and the reduced latency between the OPC UA *Server* and the OPC UA *Client* located in the edge shortens the total amount of time required to browse through the *AddressSpace* and produce the source tag list.

OpcUaSubscriptionManager is responsible for managing its OPC UA *Subscription* in

the edge. The OPC UA *Client* of OpcUaSubscriptionManager receives notification for its *MonitoredItems* present in the OPC UA *Subscription* as responses to the *PublishRequests* it sends to the OPC UA *Server*. The more frequent the OPC UA *Client* requires notifications from the OPC UA *Server*, the more *PublishRequests* it has to send. If the request interval is short, OpcUaSubscriptionManager can exploit the smaller latencies from being located in the edge in order to keep up with the requested publish interval. This enables OpcUaSubscriptionManager to support shorter publish interval than the OPC UA *Clients* not located in the edge. OpcUaSubscriptionManager also filters and transforms data before sending it to cloud. The filtering can be utilized to filter invalid values from the incoming data before the data is sent to cloud, which reduces the data volume in the cloud. The transformation can be utilized to perform simple data transformations before sending the data to cloud. For example, lots of data fields are included in the notifications published by OPC UA *Servers* for the subscribed tags which serve no use in cloud and the published value fields can be decreased into the three defined in Listing 6.2, which reduces telemetry file sizes. The transformation can also be particularly useful in cases where the component that ingests data from the edge receives data from multiple sources. The ingesting component only needs to support one specific data schema, to which the IoT Edge modules from various sources would transform their data. This can be used to reduce the load in cloud which releases the cloud computing capacity to other tasks handled in cloud. As OpcUaSubscriptionManager supports different data delivery paths to cloud, the path where the incoming data should be delivered to is also identified in the edge by the module. The slow path also utilizes caching by managing a cache in the edge and inserting the incoming tag values assigned to the slow path to it. The cache is sent to cloud from time to time to reduce the amount of network operations and the overhead from handling multiple small files instead of one big batch, which also reduces write operations in cloud.

The state of the history requests to perform is stored in History Job Message Queue (11) and OpcUaHistoryReader consumes the queue that contains the history jobs assigned for it. OpcUaHistoryReader performs OPC UA *HistoryReadRequests* based on the information specified in the history jobs and removes the jobs from the queue as soon they are processed. Responses to the OPC UA *HistoryReadRequests* can contain large amounts of data for tags whose values have changed often and the OPC UA *Client* of OpcUaHistoryReader being located close to the source OPC UA *Server* in the edge can be used to achieve lower response times to the requests. When history is requested for a long timespan or many tags, the amount of history jobs required to describe the whole history request can be large. When processing large amounts of history jobs, the lower latency and response time to a single OPC UA *HistoryReadRequest* can lead to significantly lower total time to extract the history data from the OPC UA *Server* and fulfill the history request. OpcUaHistoryReader also utilizes a cache in the same way as

the slow path of OpcUaSubscriptionManager. The received history values are inserted into it and the history data from multiple OPC UA *HistoryReadRequest* is sent to cloud in a single batch file to reduce the amount of network operations and overhead from it between the edge and cloud.

7.5 Comparison summary

This chapter aims to provide a summary of the comparison between the reviewed architectures as described in Chapters 7.1 - 7.4. The difference in the level of abstraction in the reviewed architectures makes the comparison more difficult, but the comparison still provides a decent overview of the results. In addition to the topics mentioned in earlier chapters, the complexity of architecture is also added into the comparison since it differs between the reviewed architectures. The comparison between the architectures is summarized in Table 7.1.

Table 7.1. *Comparison summary between PDI reference architecture, OPC UA Web Platform (Cavalieri et al. 2019) and REST-based OPC UA Middleware (Habib et al. 2022)*

Topic	**PDI architecture**	**OPC UA Web Platform**	**REST-based OPC UA Middleware**
Architecture	Data flow, Control API, IoT Edge	Data flow, Control API	Data flow, Control API
OPC UA compatibility	Various OPC UA *Server* by configuring IoT Edge modules, Multiple OPC UA *Servers*	Various OPC UA *Server* configurations, Multiple OPC UA *Servers*	Configurable OPC UA *Server* endpoint URL, Single OPC UA *Server*
Data flow control	Select OPC UA tags and sample rates, Delivery path to cloud, History read	Select OPC UA tags and sample rates	Read all data from OPC UA *Server*
Edge computing	Preprocessing, Caching, Lower latency to OPC UA *Server*	-	-
Complexity	High	Low	Medium

Table 7.1 shows that all the architectures roughly consist of the data flow and control API components, with the difference being the existence of Azure IoT Edge implementation in the PDI architecture. The PDI architecture and OPC UA Web platform can both be configured to support various OPC UA *Server* configurations and can connect to multiple OPC UA *Servers* at once. REST-based OPC UA Middleware only support configuring

OPC UA *Server* endpoint URL, which affects OPC UA compatibility. The data flow control also contains differences between the architectures. Both the PDI architecture and OPC UA Web Platform support specifying the OPC UA tags to collect data from and the sample rate used to produce new values for the subscribed tags. Whereas OPC UA Web Platform always uses message brokers to deliver data, the PDI architecture also supports specifying the delivery path to cloud: cheaper and slower batches or faster data streaming. The PDI architecture also includes support for reading historical data from OPC UA *Servers*, which enables fetching data for tags afterwards if it is required. REST-based OPC UA Middleware does not support configuring the data flow as it always reads all data present in the OPC UA *Server*. Edge computing is only utilized by the PDI architecture, which is utilized by preprocessing of the data before sending it to cloud, caching in order to send data in batches to reduce network operations and to achieve lower latency between the OPC UA *Client* and *Server*. However, complexity of the PDI architecture makes it harder to implement and debug, which is definitely a downside compared to the other two systems. OPC UA Web Platform has the simplest architecture of the three, which is impressive taking account the functionality it provides.

8. CONCLUSION

This thesis introduced an OPC UA process data integration solution, PDI, that aims to bridge the gap between OPC UA and Azure by providing a configurable solution that should be compatible with various OPC UA *Server* setups. The thesis consists of three phases: the first being a literature review on the important topics from the thesis' viewpoint, the second producing a requirement list for the PDI reference architecture and the final third phase introducing the PDI reference architecture. After the PDI reference architecture was introduced, a result analysis was performed in order to examine how the architecture corresponds to the research questions of the thesis.

The first phase covered the topics of IIoT, OPC UA and Azure in order to achieve an understanding of the environment and used technologies and their possibilities. The phase reviewed IIoT architectures and the technologies associated with it to gain understanding about the environment where the PDI architecture would reside. OPC UA was reviewed from the technical viewpoint in order to achieve an understanding about its capabilities and the services it provides. The OPC UA review also introduced some state of the art integrations between OPC UA and web, which were later compared to the PDI reference architecture. As PDI aims to integrate OPC UA data to Azure, the literature review also reviewed Azure from the technical viewpoint and introduces the Azure services and components that the PDI architecture consists of.

The second phase established an enumeration of requirements for the PDI architecture based on the achieved knowledge from the phase one and the requirement list provided by Insta (Appendix A). The requirements for the PDI architecture from the second phase were tightly tied to the third phase, which introduces the PDI reference architecture with the goal of matching the requirements. The PDI architecture was first introduced by providing a high-level architecture description including the Azure components that are utilized to build the architecture. After the high-level architecture description, a few key areas of the PDI architecture were more closely examined, such as the edge architecture, persisted database schema, data flow between components and an API to manage the functionality of PDI. A feature review was performed after the whole PDI architecture was introduced, which aimed to demonstrate how the different components work together within the architecture to achieve the required functionality defined by the requirement list from the second phase.

The first research (RQ1) was what components are required in order to integrate that from OPC UA *Servers* into Azure. The PDI architecture and the state of the art architectures all consisted of two categories: the components that process the incoming data flow and the components that are used to manage the data flow via a management web API. Compared to the state of the art architectures, the PDI architecture described the architecture in more detail than the counterparts. All of the architectures maintained active OPC UA *Sessions* in order to reduce the amount of OPC UA messages exchanges between OPC UA *Clients* and *Servers* since the previously established OPC UA *Session* can be exploited to instantly perform the requests that produce the wanted outputs. In addition to the state of the art architectures, the PDI architecture also introduced edge components utilizing Azure IoT Edge. Both state of the art architectures specified a single method for delivering data, whereas the PDI architecture introduced two delivery paths for the incoming data: the slow path for batch data that utilizes Azure Blob Storage and the fast path for data streaming that utilized Azure Event Hub. The PDI architecture also contained components for reading history data from OPC UA *Servers*, which neither of the state of the art architectures supported. Even though the PDI architecture provided more features, its complexity makes it harder to implement and maintain.

The second research question (RQ2) was about ensuring that the PDI architecture would be compatible with various OPC UA *Server* configurations. The OPC UA compatibility of the PDI architecture is tied to the Azure IoT Edge modules, which contain the OPC UA *Clients* used to communicate with the OPC UA *Servers*. The modules contain settings for OPC UA *Server* URL, security measures (*SecurityMode* & *SecurityPolicy*) and *User Authentication*, which can be configured into the modules. The modules obtain their configuration from the deployment templates that specify the deployed IoT Edge modules into an IoT Edge device or from the module twin properties located in Azure IoT Hub. The module twin properties provide flexible module configurations as the properties can be changed easily from Azure Portal, which is a web GUI for Azure. For every OPC UA *Server*, there are modules for three main ares: tag finder, subscription data and history data. To match the OPC UA *Server's* capabilities, the deployed module configuration can be tuned based on the server, meaning that not every module should be deployed for each server. OPC UA *Server* compatibility is not illustrated with every state of the art architecture, as one only supported connecting to an OPC UA *Server* using the endpoint URL, which locks it into using predefined security measures.

The third research question (RQ3) was how can the incoming data flow be configured by specifying which data from all available data is collected and how the data is collected. There were differences between the PDI architecture and the state of the art architectures in the matter, since not every architecture allowed to specify which data to collect from the OPC UA *Server*. The PDI architecture and one state of the art architecture allowed to specify the exact OPC UA *Nodes* to collect data from and define sample rates for

them independently. One state of the art architecture collected data from the OPC UA Server by reading all available values in the *AddressSpace*, which left no room for choosing which data should be collected. As mentioned in the architecture conclusion, the PDI architecture allows to configure the delivery path for each subscribed tag. Every architecture benefited from the standardized OPC UA *AddressSpace* data model, which allowed to collect data from heterogeneous sources via an unified interface.

The final fourth research question (RQ4) was how can PDI benefit from edge computing during the data integration process to cloud. Neither of state of the art architectures mentioned edge computing, which led the PDI architecture the only to utilize it. The PDI architecture utilized edge computing with its Azure IoT Edge modules, which are located in the edge close to the source OPC UA *Servers*. As OPC UA communication requires lots of exchanged messages, the IoT Edge modules benefit from the lower latency and response time from the OPC UA *Clients* being located close to the source OPC UA *Servers*. The OPC UA *Clients* exchange lots of messages with the OPC UA *Servers* and the faster communication between the clients and servers reduces the time to perform the wanted operations. Also the network operations between the IoT Edge modules and cloud by combining the received values into a cache and uploading the cache to cloud as a large batch. The IoT Edge modules also filter, validate and transform the incoming data before sending it to cloud, which reduces the computing load in cloud.

In conclusion, the PDI reference architecture introduces a system that can be used to bridge the gap between OPC UA and Azure in a configurable manner. The PDI architecture is compatible with various OPC UA *Servers* out-of-the-box by configuring the Azure IoT Edge modules, which are responsible for communicating with the source OPC UA *Servers*. The management web API abstracts the underlying OPC UA *Servers* by allowing the end-users to configure the data flow from multiple OPC UA *Servers* without requiring any technical details about the OPC UA *Servers*. Compared to the state of the art architectures, the PDI architecture represents an architecture used to integrate data from OPC UA to cloud in more detail and provides more features than the counterparts, with the downside being the complexity of the PDI architecture.

In future, the PDI architecture could also be extended to support OPC UA publish-subscribe model. The history features of the PDI architecture would not available with the publish-subscribe model, meaning only the subscription features would be able to support it. The current PDI architecture is only used to collect telemetry data even though OPC UA *Servers* can contain more than just telemetry data, such as metadata for the OPC UA *Variables* where the telemetry is connected from in the form of OPC UA *Properties*. By implementing an extension to the tag finder module or by creating a new IoT Edge module, even more information about the collected data would be available in cloud. Also with minor modifications to the data flow in cloud, the PDI architecture could also be converted to support other data sources than OPC UA, since the IoT Edge modules are

responsible for communicating with the data sources. By implementing new IoT Edge modules for new data sources, PDI could become even more powerful tool for integrating process data from multiple sources into Azure.

REFERENCES

Aro, J. (Nov. 2021). *OPC UA PubSub Explained*. [Online, visited on 6.12.2022]. URL: https://www.automaatioseura.fi/site/assets/files/3103/13_opc_day_finland_2021_opc_ua_pubsub_explained_prosys.pdf.

Azure Event Hubs — A big data streaming platform and event ingestion service (July 2022). [Online, visited on 9.10.2022]. URL: https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about.

Bala, R. and Gill, B. (July 2021). *Magic Quadrant for Cloud Infrastructure and Platform Services*. [Online, visited on 26.9.2022]. URL: https://www.gartner.com/doc/reprints?id=1-2710E4VR&ct=210802&st=sb.

Boyes, H., Hallaq, B., Cunningham, J. and Watson, T. (2018). The industrial internet of things (IIoT): An analysis framework. eng. *Computers in Industry* 101, pp. 1–12. ISSN: 0166-3615.

Bruner, J. (2013). *Industrial internet*. eng. 1st edition. O'Reilly. ISBN: 1-4493-6826-3.

Buyya, R. and Srirama, S. N. (2019). *Fog and edge computing : principles and paradigms*. eng. 1st edition. Wiley series on parallel and distributed computing. Hoboken, New Jersey: John Wiley & Sons, Inc. ISBN: 1-119-52506-3.

Cavalieri, S., Salafia, M. G. and Scroppo, M. S. (2019). Integrating OPC UA with web technologies to enhance interoperability. eng. *Computer Standards and Interfaces* 61, pp. 45–64. ISSN: 0920-5489.

Classic - OPC Foundation (n.d.). [Online, visited on 15.9.2022]. URL: https://opcfoundation.org/about/opc-technologies/opc-classic/.

Evans, P. and Annunziata, M. (Jan. 2012). Industrial Internet: Pushing the boundaries of minds and machines. *General Electric*. [Online, visited on 7.6.2022]. URL: https://www.researchgate.net/publication/271524319_Industrial_Internet_Pushing_the_boundaries_of_minds_and_machines.

Ferrari, P., Flammini, A., Rinaldi, S., Sisinni, E., Maffei, D. and Malara, M. (2018). Impact of quality of service on cloud based industrial IoT applications with OPC UA. eng. *Electronics (Basel)* 7.7, pp. 109–. ISSN: 2079-9292.

Get Messages (July 2020). [Online, visited on 3.10.2022]. URL: https://learn.microsoft.com/en-us/rest/api/storageservices/get-messages.

Gilchrist, A. (2016). *Industry 4.0 The Industrial Internet of Things*. 1st ed. 2016. Berkeley, CA: Apress. ISBN: 1-4842-2047-1.

Girbea, A., Nechifor, S., Sisak, F. and Perniu, L. (2012). Efficient address space generation for an OPC UA server. eng. *Software, Practice & Experience* 42.5, pp. 543–557. ISSN: 0038-0644.

Gruner, S., Pfrommer, J. and Palm, F. (2016). RESTful Industrial Communication With OPC UA. eng. *IEEE Transactions on Industrial Informatics* 12.5, pp. 1832–1841. ISSN: 1551-3203.

Habib, K., Saad, M. H. M., Hussain, A., Sarker, M. R. and Alaghbari, K. A. (2022). An Aggregated Data Integration Approach to the Web and Cloud Platforms through a Modular REST-Based OPC UA Middleware. eng. *Sensors (Basel, Switzerland)* 22.5, pp. 1952–. ISSN: 1424-8220.

Hsu, P.-F., Ray, S. and Li-Hsieh, Y.-Y. (2014). Examining cloud computing adoption intention, pricing mechanism, and deployment model. eng. *International Journal of Information Management* 34.4, pp. 474–488. ISSN: 0268-4012.

Introduction to Azure Blob Storage (Aug. 2022). [Online, visited on 2.10.2022]. URL: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction.

Introduction to Azure Functions (June 2022). [Online, visited on 30.9.2022]. URL: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview.

Introduction to Azure Storage (Sept. 2022). [Online, visited on 2.10.2022]. URL: https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction.

Ioana, A., Burlacu, C. and Korodi, A. (2021). Approaching OPC UA publish–subscribe in the context of UDP-based multi-channel communication and image transmission. eng. *Sensors (Basel, Switzerland)* 21.4, pp. 1–19. ISSN: 1424-8220.

Jensen, D. (2019). *Beginning Azure IoT Edge Computing: Extending the Cloud to the Intelligent Edge*. eng. Berkeley, CA: Apress L. P. ISBN: 9781484245354.

Kavis, M. (2014). *Architecting the cloud : design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. eng. 1st edition. Wiley CIO Series. Hoboken, New Jersey: Wiley. ISBN: 1-118-82627-2.

Kohnhauser, F., Meier, D., Patzer, F. and Finster, S. (2021). On the Security of IIoT Deployments: An Investigation of Secure Provisioning Solutions for OPC UA. eng. *IEEE Access* 9, pp. 99299–99311. ISSN: 2169-3536.

Machiraju, S. (2019). *Hardening Azure Applications Techniques and Principles for Building Large-Scale, Mission-Critical Applications*. eng. 2nd ed. 2019. Berkeley, CA: Apress. ISBN: 1-4842-4188-6.

Mell, P. and Grance, T. (Sept. 2011). National Institute of Standards and Technology, ”The NIST Definition of Cloud Computing,” [Online, visited on 4.8.2022]. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

Mourtzis, D., Vlachou, E. and Milas, N. (2016). Industrial Big Data as a Result of IoT Adoption in Manufacturing. eng. *Procedia CIRP* 55, pp. 290–295. ISSN: 2212-8271.

MSV, J. (Feb. 2020). *A Look Back At Ten Years Of Microsoft Azure*. [Online, visited on 26.9.2022]. URL: https://www.forbes.com/sites/janakirammsv/2020/02/03/a-look-back-at-ten-years-of-microsoft-azure/?sh=15ee50214929.

Ning, H., Li, Y., Shi, F. and Yang, L. T. (2020). Heterogeneous edge computing open platforms and tools for internet of things. eng. *Future Generation Computer Systems* 106, pp. 67–76. ISSN: 0167-739X.

NodeOPCUA (n.d.). [Online, visited on 21.10.2022]. URL: https://node-opcua.github.io/.

OPC UA Online Reference (n.d.). [Online, visited on 18.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/.

OPCUAClientOptions | NodeOPCUA (n.d.). Version 2.3.2 [Online, visited on 3.11.2022]. URL: https://node-opcua.github.io/api_doc/2.32.0/interfaces/node_opcua.opcuaclientoptions.html.

Part 1: Overview and Concepts - OPC UA Online Reference (Nov. 2017). Release 1.04 [Online, visited on 16.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/Core/Part1/.

Part 2: Security Model - OPC UA Online Reference (Aug. 2018). Release 1.04 [Online, visited on 24.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/Core/Part2/.

Part 3: Address Space Model - OPC UA Online Reference (Feb. 2022). Release 1.05.01 [Online, visited on 16.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/v105/Core/docs/Part3/.

Part 4: Services - OPC UA Online Reference (Oct. 2021). Release 1.05.00 [Online, visited on 24.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/Core/Part4/.

Part 5: Information Model - OPC UA Online Reference (Feb. 2022). Release 1.05.01 [Online, visited on 22.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/Core/Part5/.

Part 7: Profiles - OPC UA Online Reference (Nov. 2017). Release 1.04 [Online, visited on 24.9.2022]. OPC Foundation. URL: https://reference.opcfoundation.org/Core/Part7/.

Pauker, F., Frühwirth, T., Kittl, B. and Kastner, W. (2016). A Systematic Approach to OPC UA Information Model Design. eng. *Procedia CIRP* 57, pp. 321–326. ISSN: 2212-8271.

Saeed, N. and Husamaldin, L. (2021). Big Data Characteristics (V's) in Industry. eng. *Iraqi Journal of Industrial Research* 8.1, pp. 1–9. ISSN: 2788-712X.

Sisinni, E., Saifullah, A., Han, S., Jennehag, U. and Gidlund, M. (2018). Industrial Internet of Things: Challenges, Opportunities, and Directions. eng. *IEEE Transactions on Industrial Informatics* 14.11, pp. 4724–4734. ISSN: 1551-3203.

Stackowiak, R. (2019). *Azure Internet of Things Revealed: Architecture and Fundamentals*. eng. 1st ed. Berkeley, CA: Apress L. P. ISBN: 9781484254691.

Storage account overview (July 2022). [Online, visited on 2.10.2022]. URL: https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview.

Understand and invoke direct methods from IoT Hub (July 2022). [Online, visited on 23.10.2022]. URL: https://learn.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-direct-methods.

Unified Architecture - OPC Foundation (n.d.). [Online, visited on 15.9.2022]. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.

Vranopoulos, G., Clarke, N. and Atkinson, S. (2022). Addressing big data variety using an automated approach for data characterization. eng. *Journal of Big Data* 9.1, pp. 1–28. ISSN: 2196-1115.

What is Azure Event Grid? (June 2022). [Online, visited on 3.10.2022]. URL: https://learn.microsoft.com/en-us/azure/event-grid/overview.

What is Azure IoT Edge (Mar. 2022). [Online, visited on 3.10.2022]. URL: https://learn.microsoft.com/en-us/azure/iot-edge/about-iot-edge?view=iotedge-1.4.

What is Azure Queue Storage? (Sept. 2022). [Online, visited on 3.10.2022]. URL: https://learn.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction.

What is Azure? (N.d.). [Online, visited on 26.9.2022]. URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/.

APPENDIX A: INSTA PDI REQUIREMENTS

Iiro Kiviluoma
Master of Science Thesis
27.9.2022

Process Data Integrator: Requirements

- Data is collected from OPC UA to Azure cloud computing platform.
- The OPC UA client implementation is NodeOPCUA.
- The implementation should be compatible with variety of OPC UA server configurations:
  - OPC UA security measures (security mode/policy, authentication)
  - OPC UA server address space models
- OPC UA data can be subscribed.
- OPC UA history data can be read.
- System admin can configure the following subscription properties:
  - Source: which OPC UA nodes the data is collected from
  - Delivery: data streaming or batches
  - Sample rate: number of data points per timespan
- System admin can configure the following history read properties:
  - Source: which OPC UA nodes the data is collected from
  - Sample rate: number of data points per timespan
  - Timespan: define start and end of the history read request
  - (History data can be delivered in batches)
- Data can be collected from multiple OPC UA servers at the same time.
- Data is transformed into unified schema in cloud.
  - The schema is undefined and not part of the thesis.

Pekka Savolainen
Director, Intelligent Industry
Insta Advance Oy