LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Engineering Science

Software Engineering

LARGE SCALE DATA INGESTION PIPELINES FOR ENERGY INDUSTRY USING AWS

Atte Hemminki, 2021

Examiners: Professor Jari Porras

M. Sc. Matti Turunen

ABSTRACT

Lappeenranta-Lahti University of Technology LUT

School of Engineering Science

Software Engineering

Atte Hemminki

Large scale data ingestion pipelines for energy industry using AWS

Master's thesis, 2021

56 pages, 27 figure

Examiners: Professor Jari Porras

M. Sc. Matti Turunen

Keywords: Big data, IoT, AWS

Data is becoming increasingly important to industries. Industries can't just use data to be more profitable, but they rather depend on it. This Master's Thesis answers questions on what are the characteristics of industrial IoT data, and how the data ingestion can be done using AWS managed services. There is an abundance of industrial IoT data. Moreover, data volumes will continue increasing in future as transition towards more frequent sampling rates is ongoing. Cost mitigations are needed with big data volumes, and e.g. descriptive naming might come with a hefty cost over time. As a solution, AWS provides scalable managed services to build fault tolerant data ingestion pipelines in a serverless manner. It is also advisable to design components and pipelines in a way that those can be easily replaced with better ones - AWS is evolving quickly and you want to get the most out of it.

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT

School of Engineering Science

Tietotekniikan koulutusohjelma

Atte Hemminki

Suuren mittakaavan datan sisäänottoputken rakentaminen energiateollisuudelle käyttäen AWS-alustaa

Diplomityö 2021

56 sivua, 27 kuvaa

Työn tarkastajat: Professori Jari Porras

M. Sc. Matti Turunen

Hakusanat: Big data, IoT, AWS

Datasta on tulossa yhä tärkeämpää teollisuudelle. Teollisuuden menestys on riippuvaista datan käytöstä, eikä sitä voida enää ajatella vain yhtenä tapana tulosten parantamiseen. Tämä diplomityö vastaa kysymyksiin siitä, mitkä ovat teollisen IoT-datan ominaisuuksia ja kuinka datan kerääminen voidaan toteuttaa käyttämällä AWS:n hallinnoimia palveluja. Teollista IoT-dataa on runsaasti. Datan määrä lisäksi jatkaa kasvuaan tulevaisuudessa, kun siirrytään entistä tiheämpiä mittaustaajuksia kohti. Kustannussäästöjä tarvitaan käsiteltäessä suuria datamääriä. Esimerkiksi kuvaileva nimeämiskäytäntö saattaa tulla ajan myötä kalliiksi. AWS tarjoaa skaalautuvia hallittuja palveluita vikasietoisten tiedonkeräys putkien rakentamiseen palvelimettomalla tavalla. Komponentit ja putket kannattaa myös suunnitella siten, että ne voidaan helposti vaihtaa parempiin, sillä AWS kehittyy nopeasti ja siitä kannatta ottaa kaikki hyöty irti.

ACKNOWLEDGEMENTS

I would like to thank you – my colleagues at work, for serving me with countless discussions and learnings. Those really have excelled my work. Also big thank you for both of my thesis examiners, for all the help and support with this thesis. Especially to Matti for all the insightful discussion around the subject.

Big thank you also to my sister for pushing me for the last mile(s), and most importantly Janika, for all the support and encouragement at home. It have made finishing the work possible.

In Helsinki, 4.12.2021

Atte Hemminki

TABLE OF CONTENTS

1 INTRODUCTION 5

1.1 GOALS AND DELIMITATIONS 5

1.2 STRUCTURE OF THE THESIS 6

2 INDUSTRIAL DATA 7

2.1 INDUSTRIAL DATA SOURCES 7

2.2 USE CASES FOR INDUSTRIAL DATA 8

2.2.1 Reporting and descriptive use cases 8

2.2.2 Predictive maintenance 10

2.2.3 Operational optimization 11

2.3 STORING DATA 12

2.4 DATA VOLUMES 13

2.5 REDUCING DATA VOLUMES 16

2.6 SENDING DATA 17

2.7 DATA CHARACTERISTIC 19

2.8 DATA LAYERS 20

2.9 USING PUBLIC CLOUD PROVIDERS 22

3 ASSET DATA PLATFORM - CASE FORTUM 25

3.1 EXISTING MQTT DATA INGESTION 26

3.1.1 Overview to current architecture 26

3.1.2 Architecture from three environment perspective 28

3.1.3 High level fault tolerance 30

3.1.4 Fault tolerance between components 31

3.1.5 Monitoring and alarming *33*

3.2 ARCHITECTURE IMPROVEMENTS 34

3.2.1 Unifying data pipelines in all environments *34*

3.2.2 Technical raw backups from IoT Core *35*

3.2.3 Standard data format in Kinesis Stream *36*

3.2.4 Improve usability of data stored in S3 ..... *37*

3.2.5 Dead-letter-queues for Lambda functions *38*

3.3 COST CONSIDERATIONS 39

3.3.1 Main pipeline *40*

3.3.2 Raw data pipeline *44*

4 DISCUSSION AND CONCLUSIONS 47

4.1 DISCUSSION 47

4.2 CONCLUSIONS 49

4.3 FUTURE WORK 50

REFERENCES 51

LIST OF SYMBOLS AND ABBREVIATIONS

ADC	Application Delivery Controller
ADP	Asset Data Platform
AWS	Amazon Web Services
AZ	Availability Zone
CSV	Comma-Separated Values
DCOM	Distributed Component Object Model
DCS	Distributed Control System
DLQ	Dead-Letter Queue
DST	Daylight Saving Time
EC2	Amazon Elastic Compute Cloud
E2E	End to End
HDD	Hard Disk Drive
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
IIoT	Industrial Internet of Things
JSON	JavaScript Object Notation
KKS	Kraftwerks Kennzeichnungs System
KMS	Amazon Key Management Service
MDM	Master Data Management
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
OS	Operation System

OPC	Object Linking and Embedding for process control
PCP	Public Cloud Provider
REST	Representational State Transfer
RPO	Recovery Point Objective
SFTP	Secure File Transfer Protocol
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol
SQS	Amazon Simple Queue Service
S3	Amazon Simple Storage Service
TLS	Transport Layer Security
UTC	Coordinated Universal Time

1 INTRODUCTION

Data is becoming increasingly important to industries. Industries can't just use data to be more profitable, but they actually depend on it - it's more of a matter of survival. With purposeful use of data, industries can potentially have big savings when optimizing areas like manufacturing, producing, and maintenance. Some benefits can be gained just by using batch operations, but if one wants to pick all the fruits, it is essential to have access to near real time data. [1].

When speaking of industries, we are typically facing large amounts of data. It is not rare that a single industrial sensor can record data in a millisecond level. Usually, in one production facility there can be thousands of these sensors. And when we look at this situation from organizational perspective, which might own hundreds of production facilities, handling data isn't evidently always a trivial task. [2].

To accommodate all this data, organizations need rather large solutions to ingest, process, and store it. This whole process should be flexible enough for adding new data sources easily, and for carrying out new use cases when invented. Yet at the same time, it's at least equally important that solutions are durable. All of the data that is coming from specified sources should be captured. The solutions also need to scale easily to different load sizes and to be as maintenance-free as possible. By that, resources for data-related innovation can also be released. As a solution for these needs, there is Public Cloud service providers and services they offer. They are redundant and scalable "out of the box", and they don't require any maintenance on hardware or operating system. [3].

1.1 Goals and delimitations

This Master's Thesis provides answers to following research questions:

- RQ1: What are the characteristics of industrial IoT data?
- RQ2: How industrial data ingestion can be done using serverless services of a Public Cloud provider (PCP) Amazon Web Services (AWS)?

Answers for the research questions are found by examining an industry case and related research. All of this is done from an energy industry perspective. We will discover on how

Fortum, a large Finnish energy company, is creating and operating its industrial data platform. However, despite of the focus on the energy industry, most of the subjects can be generalized or utilized also for other industries (common issues in data ingestion and AWS architecture) as well. Although in Fortum's case the whole case project and its architecture is introduced, the main focus of this work will be in the data ingestion pipeline. Advisable strategies for implementing a particular ingestion pipeline is being found, as well as discussion of ups and downs, and possible limiting factors of the design.

1.2 Structure of the thesis

The thesis will begin with a discussion on how industrial data can be used. After, we will dive deeper into getting familiar with the characteristics of it. Then, we will discuss the benefits of using PCP and their services in industrial data platforms. Lastly we will get to know how Fortum is building a new platform for its operational asset data in AWS, and have a discussion about different architectural challenges and related costs. The thesis will be finished up with discussion, conclusions, and future work.

2 INDUSTRIAL DATA

Industrial internet has been described as one of the great innovations in industrial revolutions alongside with steam, electricity and internet [4]. Industrial internet connects machines into centralized system which then can be used to make advanced de-centralized decisions either by human operators or machine-driven analytics. The term “Internet of things” (IoT) was first introduced in 1999 [2] and we are currently in the middle of the breakthrough period of it. Since IoT was first introduced as an industrial term, its meaning has since been broadened to many other domains as well and nowadays it is usually more clear to use term industrial internet of things (IIoT).

2.1 Industrial data sources

Industrial data can encapsulate a large variety of different data sources. Usually most of the industrial data is originating from different kind of sensors, actuators, valves, and other measurement devices. These components are then usually connected with local automation systems or supervisory control and data acquisition (SCADA) systems. These “raw” data sources include the actual measurements, and quite often also some calculated values. For example, one could simply calculate the average flows in certain pipe or some more complex calculations such as emission calculations. These calculated results may stay in some sort of process information system. [2].

On top of these device originating data sources, industrial data can include other data sources. For example other kind of asset related data such as maintenance data. Maintenance data can provide information about what kind of maintenance activities different components/devices are going through and what kind service history those components are having. Maintenance related data is important in many ways. It can be used in finding the optimal operation and maintenance plans and also for finding weaknesses that different devices have. Knowledge can be then used for improving the designs in future. [5].

In addition to data that is originating from the organization itself it is common to merge some data from external sources. One usual example would be weather data which can often bring useful insight on the industrial process. Of course these external data sources are usually

combined with actual data. This can happen later on the data ingestion process. Oftentimes external data sources can also be queried at later time. Since external data sources are not usually the most critical thing to handle with. [5].

2.2 Use cases for industrial data

Industrial data can be consumed by many different use cases. Some use cases have been there for a long time and some have become more accessible due reliable networks, cheaper data processing and data storage.

2.2.1 Reporting and descriptive use cases

Probably the mostly used and well known use case for industrial data is reporting. Reporting can be done manually by querying database of interest but nowadays automatic reporting of some form is usually applied. Traditionally reporting is an activity where data is always gathered over some period of time from history. This means that data does not need to be real time data. Reporting activity collects a junk of data, which can be called as batch, processes it and gives results as output. Results can be anything from numeric values and texts to visual representations. Purpose of automatic reporting is to produce valuable information for businesses and enable them to make better decisions. [5]

Traditional reporting is able to deliver important knowledge for business from the past but it can’t deliver real time information. This is because traditional periodic reporting relies on database querying which becomes relatively expensive if it is done constantly for large data sets or/and requires multiple queries. [1].

By looking at the bigger picture we can also see some other restrictive factors in traditional reporting. As mentioned in the previous section, industrial sector is normally operating in multiple distributed locations which may have separate applications, systems, and networks. Since it’s typically restricted to make direct database connections from other than the local private network and reporting might be needed to be carried out across all of the distributed units, we have a small obstacle ahead of us. Traditional approach to overcome this would be batching up data into csv or xml files. Those files would be then transferred via secure file transfer protocol (SFTP) or by other method into centralized server. In the central location

data from all units needs to be then read and combined in a uniform way before final results can be discovered. [5].

With more novel approaches it's possible to bring data to central location much faster and with less overhead. This can be achieved by using for example Simple Object Access Protocol (SOAP), Representational State Transfer (REST), Websocket or MQTT. Moving out from data batching and closer to real-time data transfers will enable new kind of use cases. It is easy to think of the benefits a simple real-time reporting application where data is kept up to date can offer. But having up to date information constantly available can also bring completely new innovations that would not exactly serve the purpose if data flow would be slow.

One example of this kind of descriptive data visualization that raises situational awareness significantly is Fortum Cockpit. It is an application which is showing data from multiple data sources on top of 3D model of power plant. For example, it is presenting measurement values from devices and measurement points from all over the production site, open workorders and their location and convenient searching capability over related documents. In the following figure there is a screen capture of Cockpit presenting a subsystem of one of Fortums O&M contracted power plants.

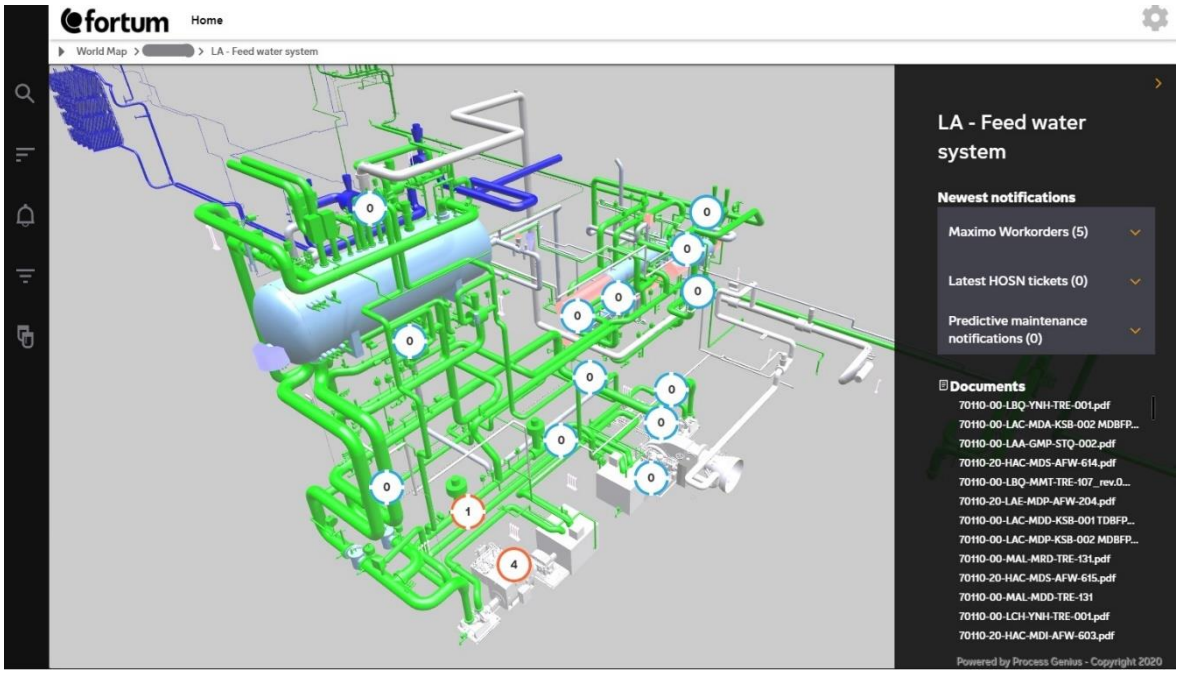

Figure 1. Fortum Cockpit presenting datapoints in 3D model.

Cockpit is offering easier access point to the data by single modern user interface which is more convenient compared to using handful of separate applications. Values that are presented to user are updating without user actions and without constantly polling for changes in the values.

2.2.2 Predictive maintenance

From industrial point of view predictive maintenance or value-based maintenance is probably the most interesting use case because of the great potential for cost reductions it has. Nowadays in every industry, the production relies heavily on machines and automated processes, since failures and breakdowns will easily cause big economic impacts. Traditional time-driven preventive maintenance, which is leaning on lifecycle estimates for components, is not a bullet proof way to avoid production stops caused by breakdowns. Lifetime estimates can't fully take into account changing operation conditions, hidden defects caused by external factors or mistakes in production, assembly, or maintenance of the component. Unexpected failures are particularly inconvenient because fixing them usually takes longer than it would if it was scheduled beforehand. Spare parts are not always easy to get since there are often a special custom made components and machines in many industries. To avoid long production breaks industries often keep internal stock on production critical components and equipment. Unfortunately, keeping internal stock creates costs and some of the expensive equipment might never end up in use. [6].

If the unexpected failures are creating costs, so is the replacement of well-functioning equipment too. Following time-driven maintenance interval can result in unnecessary maintenance breaks and decommissioning of equipment that would still have had years of faultless operation ahead. [6].

In predictive maintenance measurement data from production processes is the key source of information. Data can be for example vibration or temperature measurement from bearings or basically any kind of measurement data from the process. This data can then be enriched with records of actual maintenance activities. This combination of data enables anomality detection in measurement and when there is enough history and different occurrences, those detected anomalies might get probabilistic explanation of the cause and so predict what is going to happen. In predictive maintenance it is normal practice to use machine learning

(ML) models for analyzing data. Creating successful predictive maintenance program is not typically a quick sprint since training accurate ML models need rather long history of data.

Predictive maintenance allows more reliable operations without unexpected maintenance breaks. It can be also used for avoiding too early equipment replacements and unnecessary maintenance breaks. Looking across a large fleet of production sites, predictive maintenance data can be utilized also for making wiser investments by prioritizing them with the knowledge of conditions from each site and expected scenarios and their impacts.

2.2.3 Operational optimization

Optimizing operations opens up great opportunity to reduce costs by using assets more efficiently. Traditional way for trying to optimize assets usage is done from the control room of production site. Despite the fact that control rooms have evolved a lot since early days, there still is dozens of monitors presenting values form sensors and cages. It is really hard for human operators to make optimal decision from those. [6].

By utilizing the historical data and creating optimization models it is possible to propose better optimized operation plans. In some cases those optimized proposals can be then recommended to human operators. Nevertheless, normally the ultimate goal would be a system that would optimize itself, all automatically. In this scenario humans would only monitor that everything runs smoothly. Achieving this goal means that real-time data is required. [7].

Operational optimization can be done in smaller units and across large fleet. For example Airbus which have increased the number of sensors in its new A350 (2020 model) to 28000 from the predecessors sensor count of 6000, is now able to save millions in yearly fuel consumption by performing automatic inflight optimization with data coming from the sensors [7].

Another example is Fortum E2E H&C Digital Twin which is an ongoing project at Fortum. The purpose of Digital Twin, among other capabilities, is to optimize heat production of district heating network in Espoo, the second largest city in Finland. Digital Twin is running a dynamic simulation model of Espoo district heat network using Apros® District, which is constantly getting real-time measurement values from the actual network and some

complementary data (e.g. weather forecasts and measurements). By utilizing the up-to-date network simulation model and an optimization algorithm, Digital Twin is then able to make optimized plans on how the district heating network should be operated. [8].

2.3 Storing data

Industrial processes are producing a huge amount of data from which only small percentage is typically being stored. For some use cases less frequent data is totally fine, when to others more frequent data flow is essential. Fortunately, storage prices have constantly been decreasing over the years which is enabling the storage of even denser data in economical way. The following figure is presenting projection of average price per GB for HDD US dollars by the Coughlin Associates.

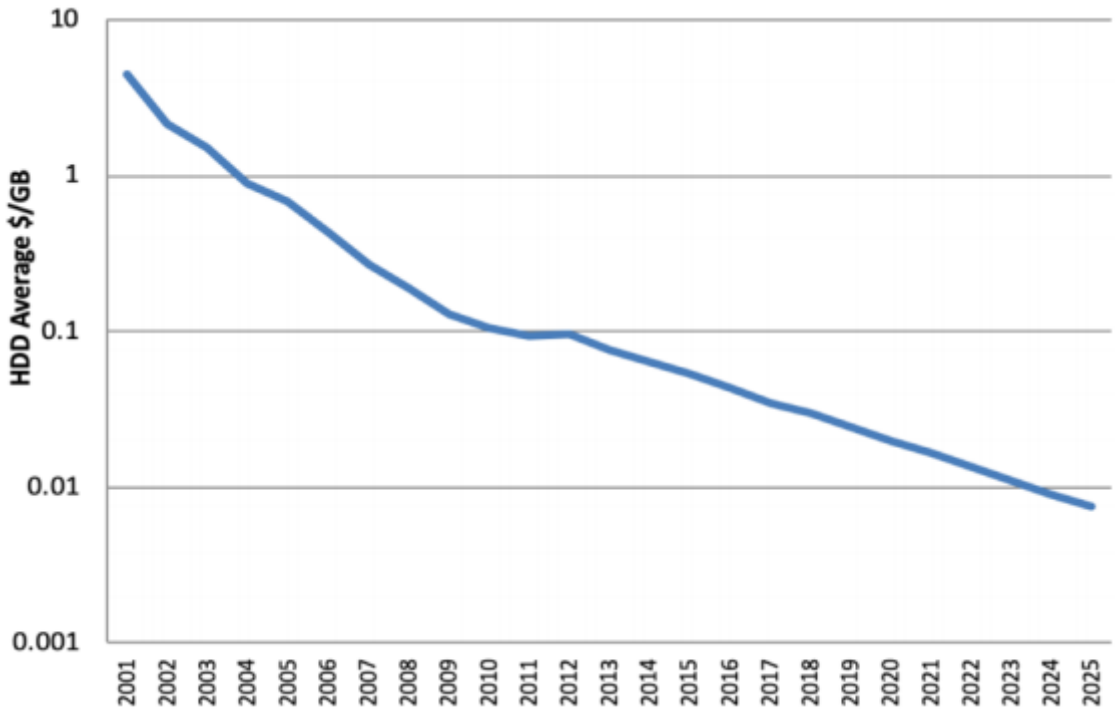

Figure 2. Projection for average price per GB of HDD in US Dollars [9].

As we can see there is a significant decrease in the average price of HDD. From 2004 the price has come down to one tenth in 2010. It is also expected that the price in 2023 is going down again to one tenth from 2010 prices.

2.4 Data volumes

As presented earlier, cost of storing data has drastically decreased over the years which makes storing more data economical. Transition from hour or minute level to real time data usage is ongoing and data volumes are getting increasingly larger. But how much data then might be coming in? When we are discussing about industrial processes which are using some sort of distributed control system (DCS) or SCADA, communication is normally utilizing OPC, or the newer OPC-UA standard. OPC (Object Linking and Embedding for process control) first released in 1996 is specifying a common language to be used in real-time communication between control devices and client applications build on Microsoft operating systems which at the time dominated the industrial automation field [10]. OPC-UA is referring to Unified Architecture and was released in 2008. It is an updated version from original OPC standard and it is enabling communication across multiple platforms, not just Windows based DCOM (Distributed Component Object Model). [11]. Following figure is clarifying how OPC server based architecture can look like.

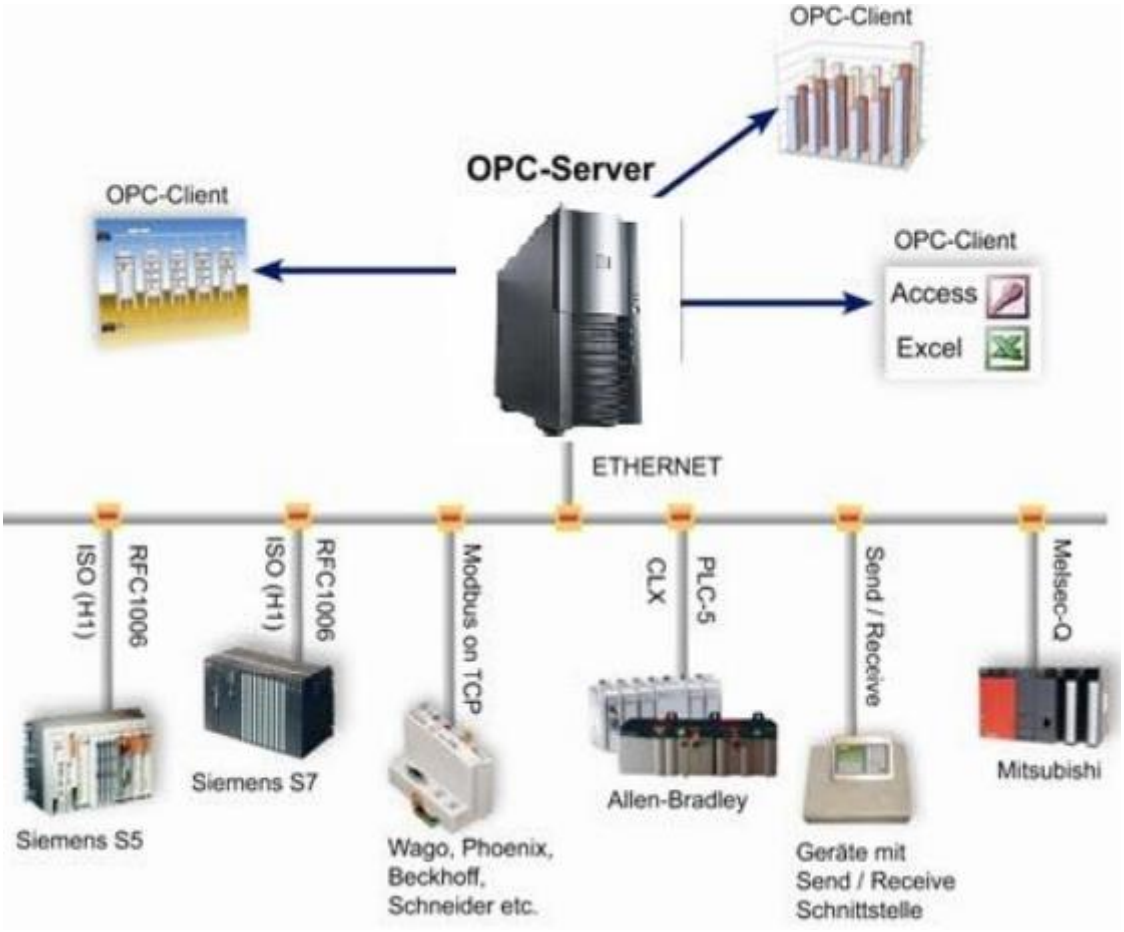

Figure 3. OPC server based SCADA architecture [11].

OPC-UA messages from devices are usually interpreted in Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format when transferring to downstream applications. [12]

One message in JSON could look like the following.

```
{
  "id": "Wind_NNNNN_OPCUA.NNNNDCS.DA.NNNNN.WTG05.WCNV01.GriA3",
  "v": 1973,
  "q": true,
  "t": 1600066667701
}
```

Previous JSON object is actually extracted from one Fortum wind farm OPC UA server, from which we will have more discussion later in Fortum case. “Id” is identifying particular

object to some specific device in the windfarm. Value attribute “v” is for the value. Quality attribute “q” is specifying if the measured value should be trusted. Timestamp “t” is the timestamp of the event in unix time milliseconds in Coordinated Universal Time (UTC).

In industrial production plants there can be anything from a few hundreds to tens of thousands data signals like the JSON example earlier. In its JSON form, the size of one message is 125 bytes (example JSON presented earlier). If we imagine a single production facility which has 1000 signals, with one second sampling rate this would mean that every hour approximately 420 Mega Bytes (MB) of data is produced. Message and data cumulation from 1000 signals is presented in the following figure.

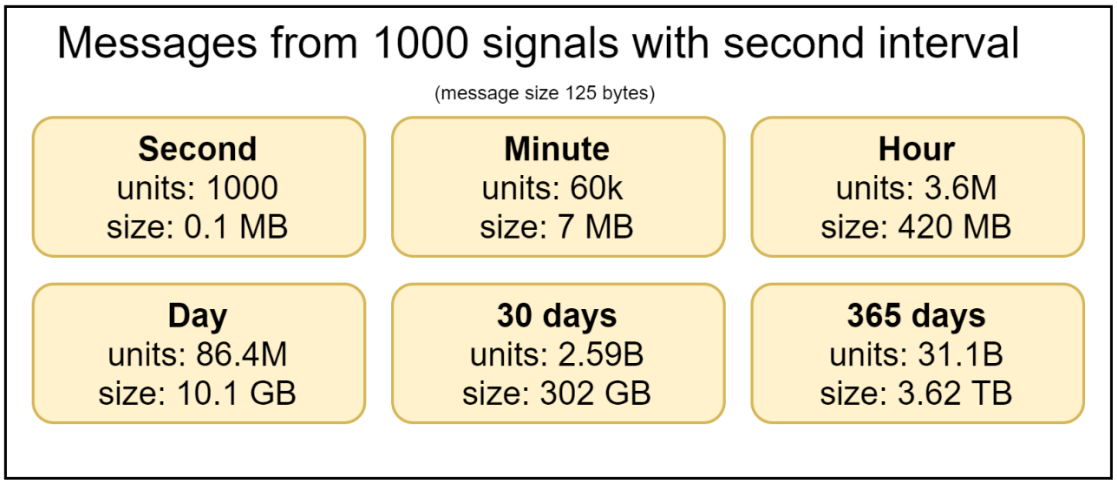

Figure 4. Message and data cumulation from 1000 signals.

Respectively 100 similar plants would produce 100 000 messages every second and about 1 TB of data every day. Some use cases might benefit from even shorter time interval, say 0.1 second interval, which would yet again multiply data volumes by ten.

In addition to the OPC data signals, also aggregated data from site can be sent to the centralized system. Fortunately, aggregated data is usually in higher resolution such as 1 minute or 10 minute intervals. Depending on production site, aggregated data can still represent the lion’s share of the data overall, if there are much more aggregated datapoints. Aggregated data is collected from historian database on site. Data itself can normally be presented in similar JSON format as OPC data.

Other related data, like maintenance data, have only a small share in overall data volume in comparison to OPC and aggregated data. This is mainly because, for example maintenance activities for different assets are created manually by human operators.

2.5 Reducing data volumes

Transferring, processing, and storing data takes resources and creates costs. Unnecessary data can also have a negative impact to performance of the end application e.g. by increasing latency on queries and data streaming. This is why some approaches for decreasing data volumes should be thought of.

The easiest way to reduce data volumes is simple, only store what is needed. This can be achieved by careful design process early on, but often it is easier to store more and there is good rationalize behind that, of course. We can't really tell if the data would be valuable in years to come. It is possible that after two years a new innovation around the subject is found and we would need two years of historical data in order to get new machine learning application trained. This is when it comes to the decision whether we should store a bit of data or not and in what sampling rate, every 10 minutes or every second. We should always carefully think that the accuracy in which we are storing data is meaningful. For example, if we are storing water inlet temperature with 7 decimals, we are way over the accuracy of the measurement device. Recording as many decimals only brings noise to the downstream processing and should be avoided.

We can also reduce the amount of data by sending and storing only values that have changed. Generally thinking it does not provide any value to store every second that the temperature in a location has stayed the same. We should be able to figure out the current temperature by searching the latest received record. However, this approach requires that we can trust that all the messages are being received and successfully processed. If some information is lost on the way, we should have a method for gaining the information that the latest value isn't actually correct. Adding this kind of functionality to source system on site may be a challenging job depending on the system capabilities. Another complexity adding factor is that an organization might have multiple sites which have different source systems and finding generalized solution, that is easy to take in use everywhere might be near to

impossible. So, if source system is supporting an option to send only changed values it's good take in use. Otherwise, careful thinking on achieving trust on the data is needed.

When messages are sent to centralized system it's useful to send multiple readings at once which lowers the total number of messages transferred. Human readable JSON payloads are simple to work with, it makes developing and debugging much easier. Messages could also be compressed in order to reduce message size.

2.6 Sending data

Breaking the silos is not always a straightforward thing. In industrial environment we have many restrictions along the way, before we have received data from all sources into a centralized place. In order to protect low level automation and control systems, we can never have direct connections from cloud to data sources. Several firewalls and checkpoints are keeping cloud and low-level system separated. If one would want to send data by using flat file integration from low level systems towards cloud, we would need to have jumping servers in between for security purposes. The following figure is illustrating how communication between different network layers are handled in Fortum.

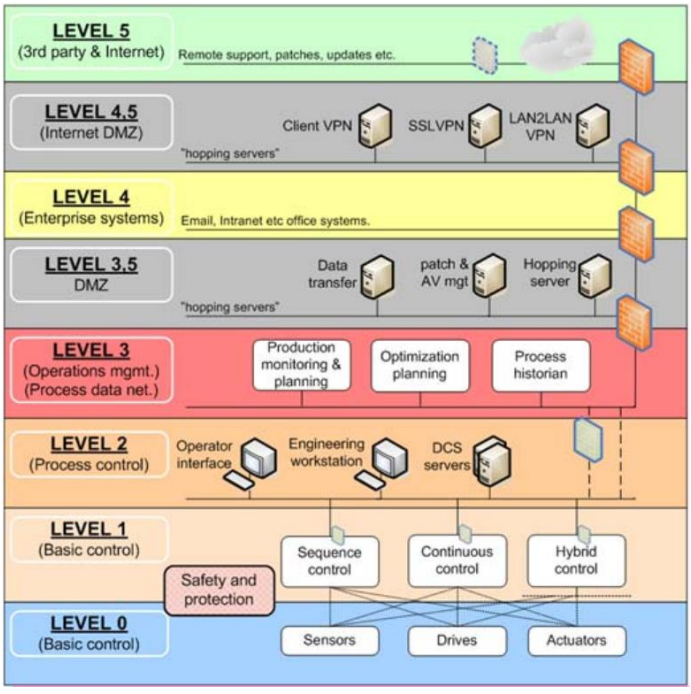

Figure 5. Communication between different network levels [13].

The main idea behind the model is cutting/stopping connections at each border. Model is based on ISA-99 and ANSI/ISA 95 standards. [13].

Industrial IoT data sending could happen from level 3, with IoT Gateway device. It can be configured to send data in specified sampling rate and dead-bands (accuracy). One example of IoT Gateways is Kepserverex. There is a high level figure about Kepwareserverex below.

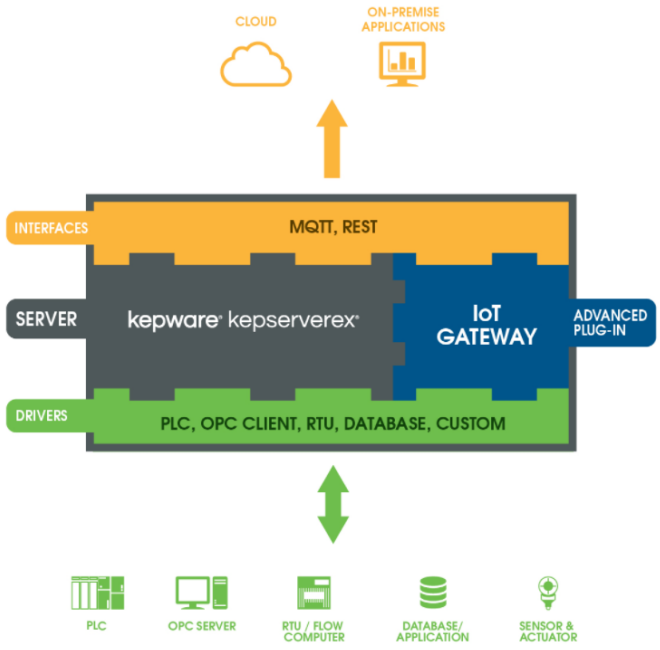

Figure 6. Kepware Conceptual diagram [14].

IoT Gateway device would be connected to OPC UA Server in level 3 or in level 2 and would then send data towards cloud using Message Queuing Telemetry Transport (MQTT) protocol. MQTT is becoming a de-facto protocol in IoT field. It is lightweight protocol that is widely used and supported. [15]. On top of firewalls between each layer there is also application delivery controller (ADC) which is checking all the contents before sending messages forward towards cloud. Networking may also bring some limitations to message sizes. For example it is not always possible to send full message size that MQTT protocol supports through ADC.

There is a sample message payload below from one of Fortum wind parks. Message payload uses JSON format.

```
{
 "timestamp": 1600066668486,
 "values": [
  {
   "id": "Wind_NNNNN_OPCUA.NNNNNDCS.DA.NNNNN.WTG05.WCNV01.GriA1",
   "v": 2159,
   "q": true,
   "t": 1600066658658
  },
  {
   "id": "Wind_NNNNN_OPCUA.NNNNNDCS.DA.NNNNN.WTG05.WCNV01.GriA3",
   "v": 1973,
   "q": true,
   "t": 1600066667701
  }
 ]
}
```


2.7 Data characteristic

Data that is collected from multiple different systems and production sites can often be inconsistent. These inconsistencies need to be taken care of in order to effectively use data. Different production sites may be located in different time zones. Some production sites might use standard UTC time and others might be running in local time. In Nordics, some sites might also change their time following Daylight Saving Time (DST) when others are only using for example the Winter time. It's also possible that one site is running in incorrect time, so in other words their clock is not correctly synchronized even though synchronizing should be basic function. Incorrect timestamps can lead into faulty interpretation of data which can be crucial for many use cases.

Another point is how timestamps are formatted. In the best case scenario all timestamp are formatted in a same way following for example ISO 8601 standard [16] or as unix timestamps. ISO 8601 format has two benefits, it's human readable and it's also revealing the time zone. Unix timestamp in other hand is just a simple number. Example timestamps from both are presented below.

ISO 8601 timestamp
1997-07-16T19:20:30.45+01:00 (YYYY-MM-DDThh:mm:ss.sTZD)

Unix timestamp (millisecond)
1606601133

Some legacy systems might also use even more exotic timestamp formats. From data ingestion point of view, it would be ideal if all timestamps would be converted into same format and UTC time before sending it to data ingestion pipeline, but sometimes performing conversions in the sending end can be difficult. Ingestion pipeline that is capable of handling different conversions internally is more flexible and it can speedup rolling in new data sources.

Another thing to be taken into account is encoding. Source systems in different locations might or might not be using same encoding. Again, flexibility is added if data ingestion pipeline can handle different encodings. Usually, pipeline should produce uniform encoding. Utf-8 encoding is good choice for that.

Sites seldomly have universal naming conventions. Organization might have multiple naming standards in use due some acquisitions or other reasons. Even if some naming standard is used, it doesn't guarantee that naming is done similarly. One example of such standard is Kraftwerks Kennzeichnungs System (KKS) [17]. There is often some site specific interpretation of the standard. This means that we could end up having tens of different names for same thing. From data scientist point of view, it would be much easier if they could deal with conceptual names, say turbine generator is always turbine generator. Conceptual naming can be handled outside the data ingestion pipeline itself. By using master data management (MDM) tools such as Semarchy xDM, conceptual names can be mapped with site specific names and data scientist can process data using conceptual names.

2.8 Data layers

Data is often being layered in different layers which have their own characteristics. There can be several different storage layers which have their unique processing activities in between. Data layering strategies can vary a lot depending on what tools are used, where the data is coming from, how it looks and the independent choices made in the development

process. For example, Apache Spark uses following methodology in data layering where data quality is constantly increased when it flows from source applications towards consumer applications.

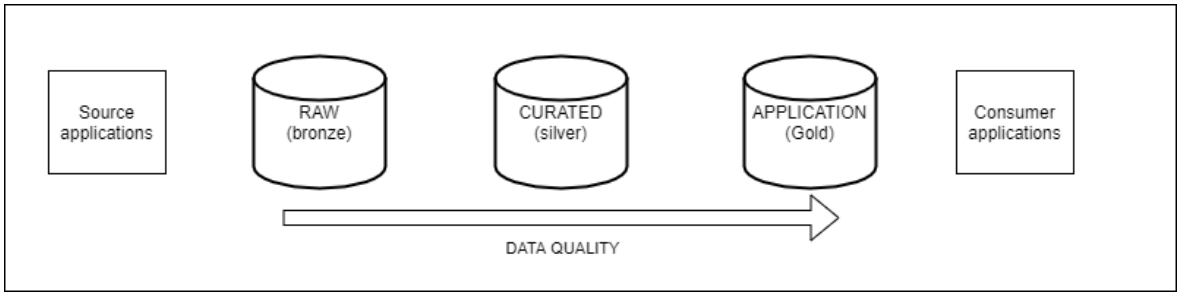

Figure 7. Data layering.

Raw layer receives raw data from source systems. Data in raw layer is typically the same it is when received, no transformations are done before the data is added to raw layer. Raw data layer can be stored as a backup, in case processing in the later stages is failing or an error that would cause some data to be lost is found in the processing. There can be also a number of intermediate layers before the data is actually stored in the raw layer. For example, if one uses file based ingestion, files could be first put in a "landing" layer from where the data would be then moved into the raw layer. Between the landing layer and raw layer, some small transformations could be executed. For example, data could be converted into general format, e.g. transforming csv files into json files, but these transformations should be rather minimal and those can be also done when moving data to standard layer. Data in the landing layer is not usually something that one would want to preserve, but instead a retention time would be set, after which the data is removed.

When data is moved from raw layer to curated layer there are more transformations happening. Firstly, data would be typically partitioned by the value time, when in raw layer partitioning is done by the receiving time. Secondly, deduplication is done here. Duplicate records would be dropped here if such records are received from source. Transformations at this step would include the following: convert timestamps to UTC time, formatting timestamps to selected format, converting datatypes, changing attribute names of the data, mapping contents in selected way, parsing ids etc. Typical users of curated data are data scientist which is why layer should be designed while keeping their needs in mind.

Application layer is for business oriented use cases. Data can be aggregated and transformed for application layer according to what is suitable in consumer application perspective. Usually, application layer data is being served from faster data storage than would be needed in raw and curated layers. This could mean for example that data is stored in timeseries database and relevant history is kept in memory for better performance.

2.9 Using Public cloud providers

The use of public cloud providers such as Amazon Web Services (AWS), Microsoft Azure or Google Cloud is getting more and more popular over the traditional way of purchasing own servers and maintaining them in dedicated data centers. According to markets and markets infrastructure as a service (IaaS) market was expected to grow from 2015 market size of 15.79 Billion USD to 56.05 Billion USD in 2020 [18]. Growth estimation seems to be somewhat accurate as Gartner reported that IaaS market in 2019 reached 44.46 Billion USD and was increased by 37.3% since 2018. The forerunner of IaaS market was AWS with an impressive market share of 45.0 %, second being Microsoft with 17.9 % market share. [19].

Growth in the popularity of using PCPs is tightly related to modern needs and the benefits that PCPs can provide. Modern applications are often using more resources than a few decades ago. Almost everything is being recorded and if something isn't there is most likely a plan to do so. This recorded data is then processed in order to provide business value. Increasing volumes and need for more complex and fine grained processing sets a demand for resistant, highly available, scalable and secure infrastructure. This is what PCPs are providing out of the box with no upfront fees and investments. [20].

Using PCPs is reducing the time needed for maintaining the infrastructure, as many things are automated and/or those are a responsibility of service provider. The boundary can be for example on operation system (OS) level, when customer of PCP is responsible for making updates and security patches to the OS and application on top of that. Need for maintaining the infrastructure can be further reduced when using serverless components. With serverless components user is typically freed from any kind of updating and patching activities. All patching is then on service providers responsibility. [20].

In the following figure there is shared responsibility model that AWS is applying. It's showing the boundaries of responsibilities between AWS and the customer for virtual machines and other infrastructure services space is divided.

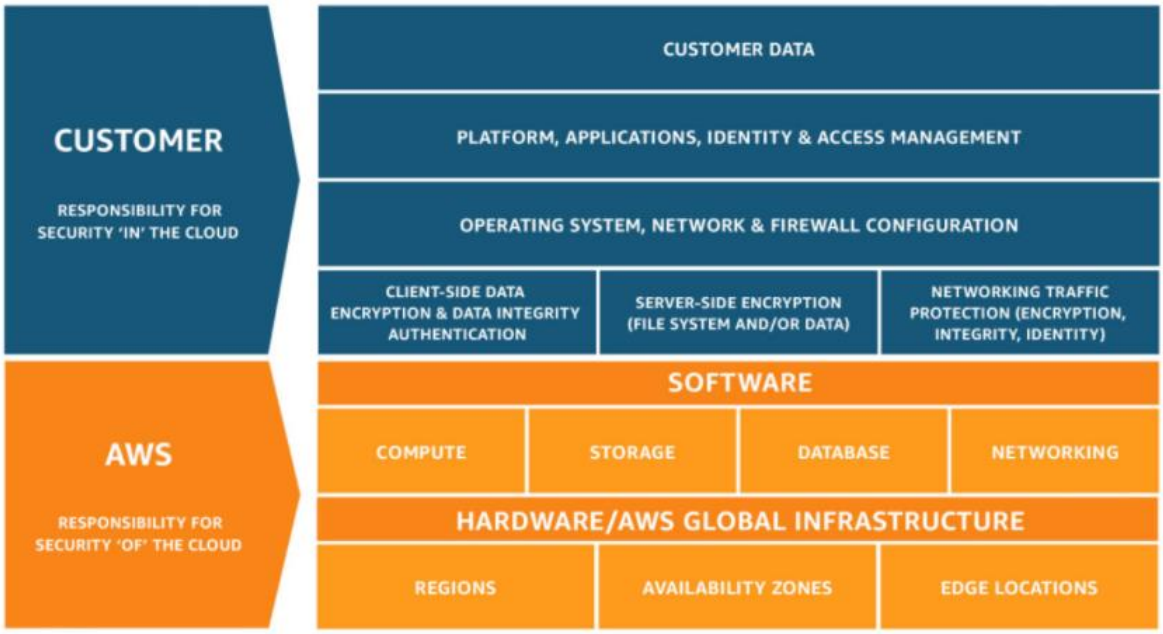

Figure 8. AWS Shared responsibility model [20] .

When using AWS managed services, customer responsibility is decreased further. Following figure shows how AWS and customer responsibilities are divided with different service spaces.

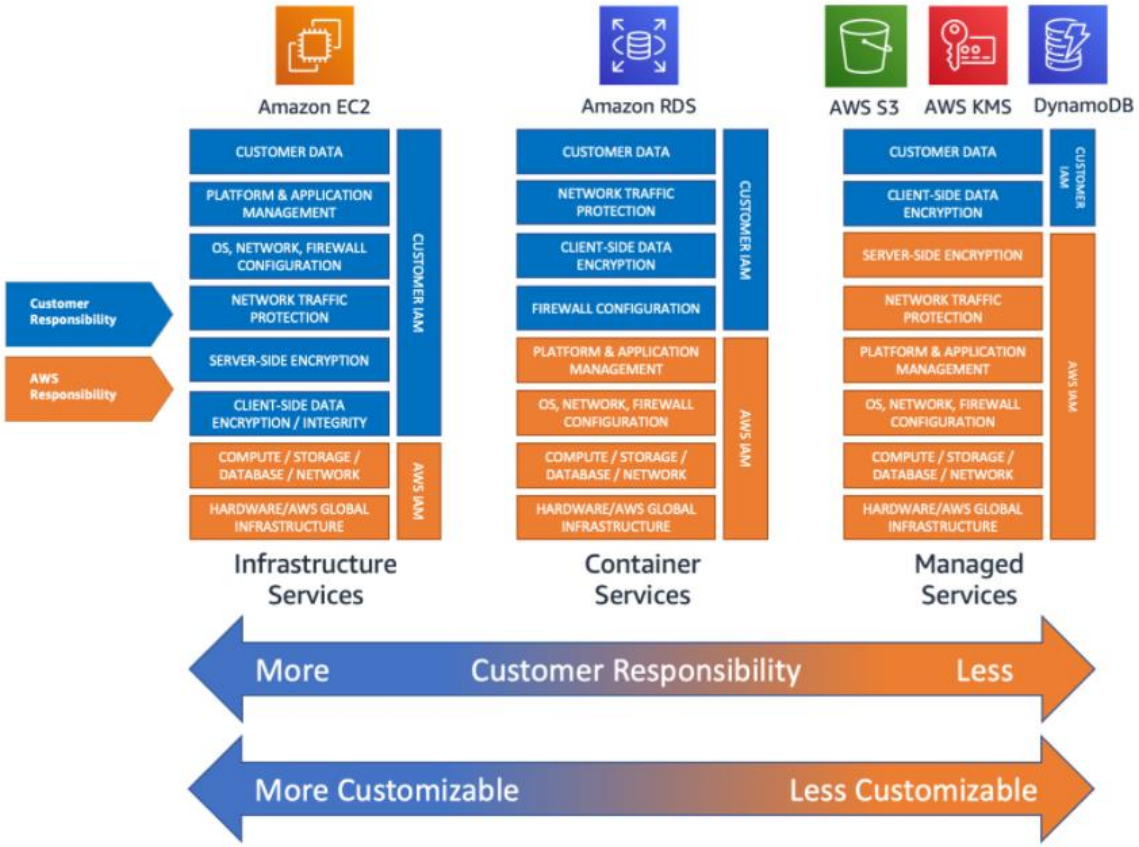

Figure 9. AWS Shared responsibility for different service spaces [21].

When using infrastructure services, there still are many things that customer will need to take care of. When moving towards container services and managed services, customer will need to perform less and less operations to manage the infrastructure.

3 ASSET DATA PLATFORM - CASE FORTUM

Asset data platform (ADP) is built within a project that was launched at Fortum in 2020. It is a centralized platform for all operational asset data inside Fortum. The platform is ingesting data from multiple Fortum's business domains (hydro, wind, heat, solar, recycling). From each business domain, there will be multiple source systems from which the data is transferred into the ADP. These source systems include e.g. TOPi -process information and optimization system, IBM Maximo – Asset management system, Project Wise – document management system, and various underlaying DCSs and SCADA systems. Data from these sources can be ingested into ADP by using several different messaging protocols which suits each data source system the best. When data has been ingested to the platform, it is processed and served to end application, again leveraging multiple protocols. These various end applications and business services are the ones actually delivering the business value. Below there is a high level figure on how ADP is positioned.

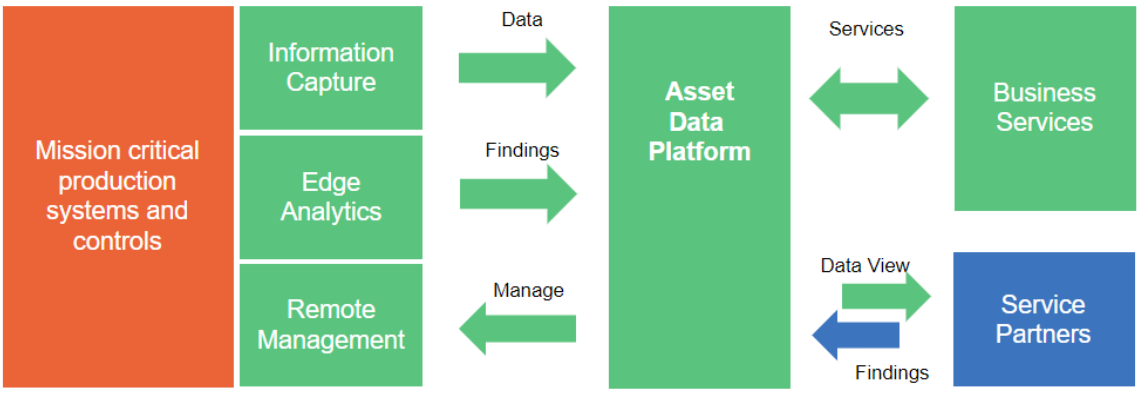

Figure 10. ADP in high level [22].

Grasps of starting the ADP project are coming from earlier style of siloed development for single specific use cases. ADP is supposed to gather those small projects under a one single platform using unified data structures and reusable components. This will lower the time needed for operations and maintenance as a whole, and release time for new development. In addition, there will be less duplicate data to be transferred, stored, and processed as multiple applications can use same source data from a single place.

In this case study we will focus on the data ingestion side of the platform, and to be more precise: MQTT data ingestion. MQTT data ingestion is used for the most critical data ingestions (large volume, near real time use cases). We will first make precise review on what is already done in those siloed projects, and then propose improvements. Lastly, we will make some cost estimations of the pipeline, as it will always be one of the key factors when assessing solutions.

3.1 Existing MQTT data ingestion

Basis for the architecture that is used in ADP, is a project started in 2019. It was first iteration of trying near real time data intake and it was the most evolved siloed solution. In this section, we will be focusing on the data ingestion part of the solution, MQTT data ingestion, as vast majority of data is coming in via MQTT messages.

3.1.1 Overview to current architecture

Data from several plants is sent by MQTT Clients running on Kepware servers at the plants towards AWS IoT Core. Every change that Kepware OPC Clients are receiving from OPC Servers are published with MQTT client to AWS IoT Core. Between Kepware and internet there is ADC (F5), which is checking the contents of all messages before letting them go through and also other firewalls have to be opened (security requirements). This is presented in the following figure.

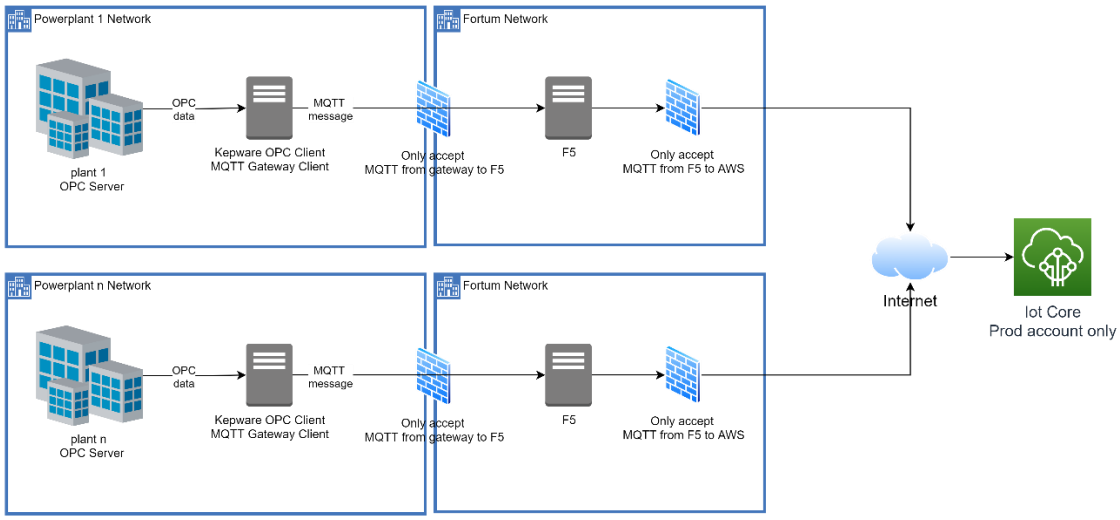

Figure 11. Existing MQTT data ingestion from powerplants.

All communication is encrypted with Transport Layer Security TLS 1.2. which is a default in AWS IoT Core. Server and client are both authenticated by X.509 certificates.

When data lands into IoT Core it is caught by IoT Core Rule. It is then triggering a rule to trigger Lambda function containing the message as a payload. IoT Core Rule is encoding the message as base64 encoded string. This was done for added flexibility for the data senders. Downstream AWS components can't handle other than UTF-8, but it was considered to be easier to convert different character sets on cloud side. This allowed getting data inflow to be done much faster than it would have been otherwise. The below figure shows how data is flowing in production account.

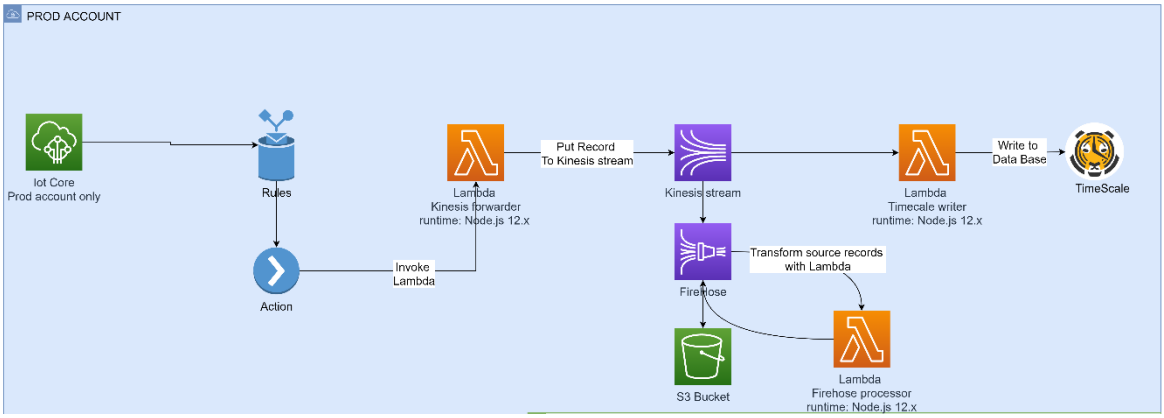

Figure 12. Existing data flow in PROD account.

First Lambda function "Kinesis forwarder" converts the message to UTF-8 and then writes it to Kinesis Data Stream. Kinesis stream is attached to two consumers. Kinesis Firehose is batching records, which are then being transformed with Lambda function "Firehose processor" to a standard format, and finally Firehose writes transformed data batches to S3 files containing lines of JSON objects.

Other Kinesis Data Stream consumer is Lambda function "Timescale writer", which is doing exactly the same transformations that are being done with "Firehose processor". After transformations, data is being written to PostgreSQL database with Timescale timeseries extension.

All configuration data on how each dataset should be transformed is stored in a DynamoDB table as can be seen in the below figure.

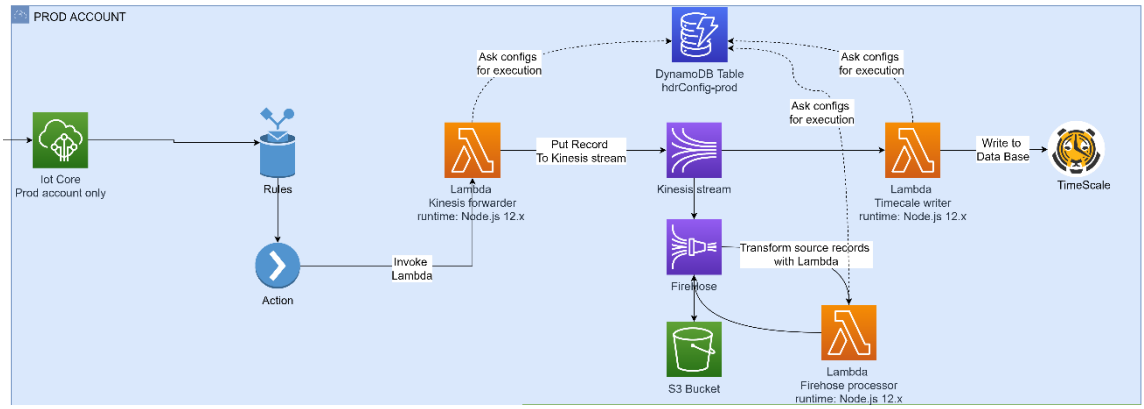

Figure 13. Existing data flow in PROD account, with DynamoDB.

All three Lambda functions now need to make queries to the DynamoDB table, as some transformations to the data are done in every Lambda function. Especially inefficient is doing same transformations in both Timescale writer and Firehose processor Lambdas. Even though DynamoDB is used for centralized place for holding configurations, it's not any kind of bottleneck in the processing. Configurations actually needs to be queried only when Lambda cold starts (container for running the code is initialized). This is done possible by saving configurations into Lambdas memory as a global variable, from where it can be then retrieved on warm starts.

Node.js is used as Lambda function runtimes, which is a good choice as it offers fast cold starts compared to compiled language like Java [23]. It is also easy to develop with tools like npm. All three Lambdas have fairly small package size as too many libraries are not used. Package sizes are varying between 208 kB to 1.6 MB.

3.1.2 Architecture from three environment perspective

In this subchapter we will put things into perspective by showing how the architecture looks in its current form in three environment development setup. In general the existing three environment infrastructure have been span up with Terraform, infrastructure as code software tool. Terraform makes it easy to deploy cloud infra to PCPs with command line interface. It also enables us to write reusable components and make easy deployments to different environments.

Current implementation is using three environments: development, test and production. For easing the downstream development, dataflow wanted to be replicated as its fullest to also

test and development environments. Customer wanted to do data replicated only in the cloud side to decrease load on sources. That need resulted in the following architecture.

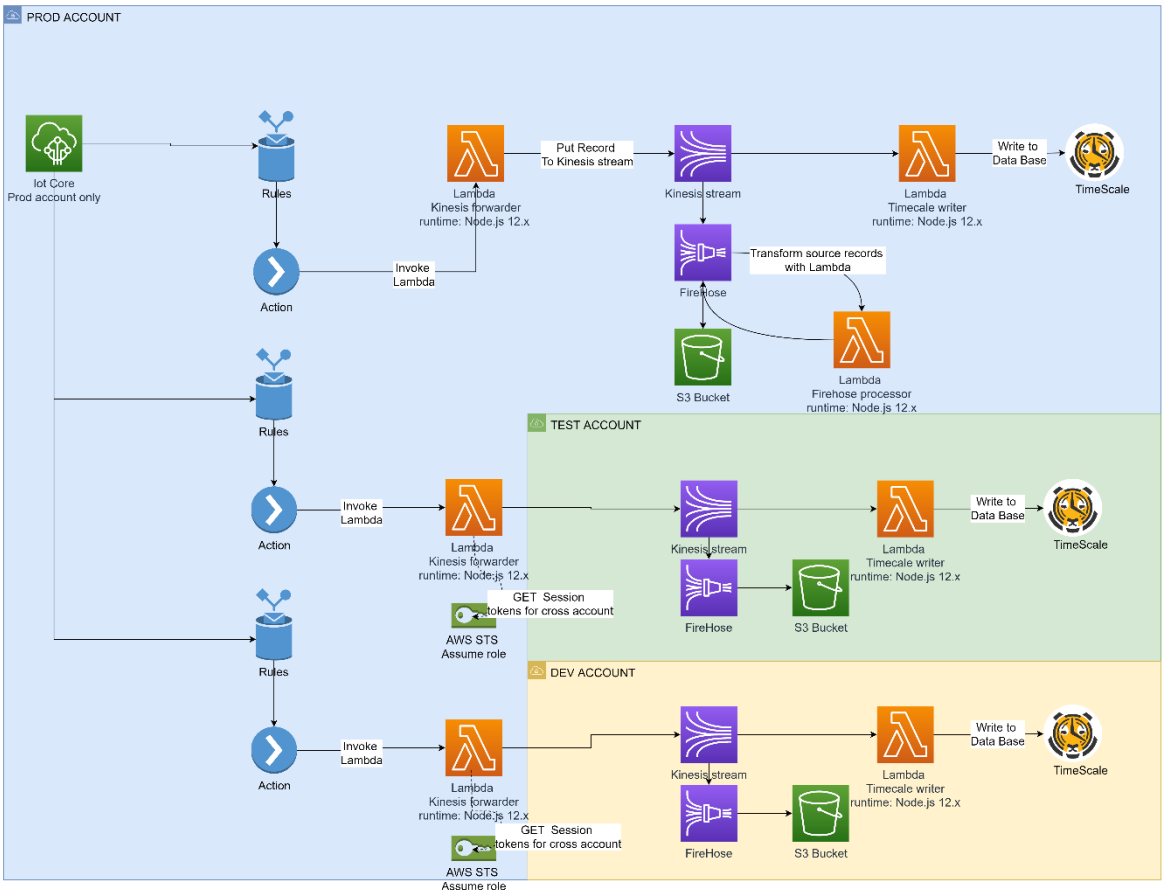

Figure 14. Existing architecture in 3 environment setup.

Note! Some details like DynamoDBs and Firehose processor Lambda (in test and development environment) have been left out from the figure for simplicity.

Although above architecture enables us to get replicated data streams to downstream processes, figure reveals us some down sides of chosen architecture. Firstly, production environment is different from the other environments. All environments are identical only at the Kinesis Stream. This means that all changes to parts that are upstream from that Kinesis Stream are deployed directly to production environment. Making mistakes there could result in data brakes and losses as even any kind of raw data is not being stored before

that. Also, in development perspective, having such setup is making implementation more complex.

3.1.3 High level fault tolerance

Customer Recovery Point (RPO) Objective was to have only minimal if any data losses. Existing architecture is fault tolerant by its nature as it has been built using AWS managed services. They typically offer fairly good service level agreements (SLA) which would be otherwise hard to reach with limited resources. SLA for IoT Core uptime is at the time of writing 99.9 % for each AWS region, which corresponds to 43 minutes and 49 seconds downtime monthly [24]. AWS Kinesis and S3 are providing the same 99.9 % uptime per AWS region [25][26]. AWS Lambda is giving highest uptime of 99.95 % from these main data pipeline components [27]. AWS DynamoDB which is used for holding configuration data AWS gives uptime SLA of 99.99% in single AWS region which could be increased to 99.999 % if Global tables were used [28]. PostgreSQL database with Timescale extension is running on single EC2 instance (Read replicates are in use). AWS gives SLA of 99.5 % uptime to individual EC2 instance which is the case in the current setup. For multi availability zone (AZ) setup, where instances are ran concurrently in multiple AZs AWS gives 99.99 % uptime. [29]. We need to note that SLA given to EC2 is only considering AWS managed part of the Database instance (see figure 8 AWS Shared responsibility model).

In the figure below, different uptime percentages are converted into corresponding downtimes.

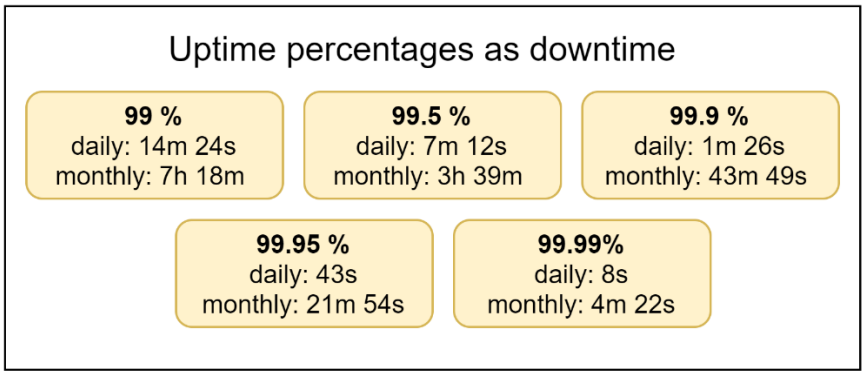

Figure 15. Uptime percentages as downtime.

If higher availability than 99.9 % at cloud side would be desired, easiest way would be to introduce new region to the architecture, as first component (IoT Core) is having 99.9 % uptime. However, when considering whole end-to-end data pipeline at the moment, weak links of availability (meaning here if the data is successfully even captured) are upstream from the cloud, and since it would be overshooting to take another AWS region into use.

3.1.4 Fault tolerance between components

Starting from upstream of the data pipeline we have IoT Core and Lambda connected as shown in the figure below.

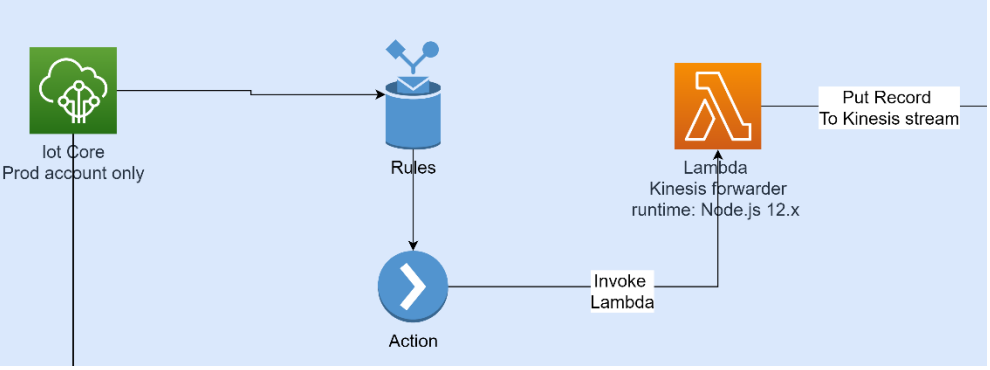

Figure 16. IoT Core to Lambda connection.

IoT Core lacks any kind of buffering capabilities when using Rules and Actions [30]. This leaves current architecture vulnerable to data losses when AWS Lambda service is not available even for short times. Luckily AWS Lambda is having the highest SLA uptime as seen in the previous paragraph. Also, it is worth of noting that architecture is handling well situations where Lambda is throttling. This is because Lambda invocations are done asynchronously and all the messages are being put into Lambdas internal event queue, where they can last up to six hours if concurrency limits are being hit. [31].

When AWS Lambda is invocated with asynchronous invocation Lambda function can be retried from its internal event queue up to two times. First retry attempt is done after one minute and the second one after two minutes. This means that current architecture starts losing data after three minutes if there is problems with running the actual Lambda function. This could result from Kinesis Data Stream being unavailable, Kinesis Stream maximum capacity being hit or simply errors in the function code because of update or invalid data. In

the current architecture we can really buffer data only after it has landed to Kinesis Data Stream. In the following figure there are some additional information to latter part of data pipeline. [31].

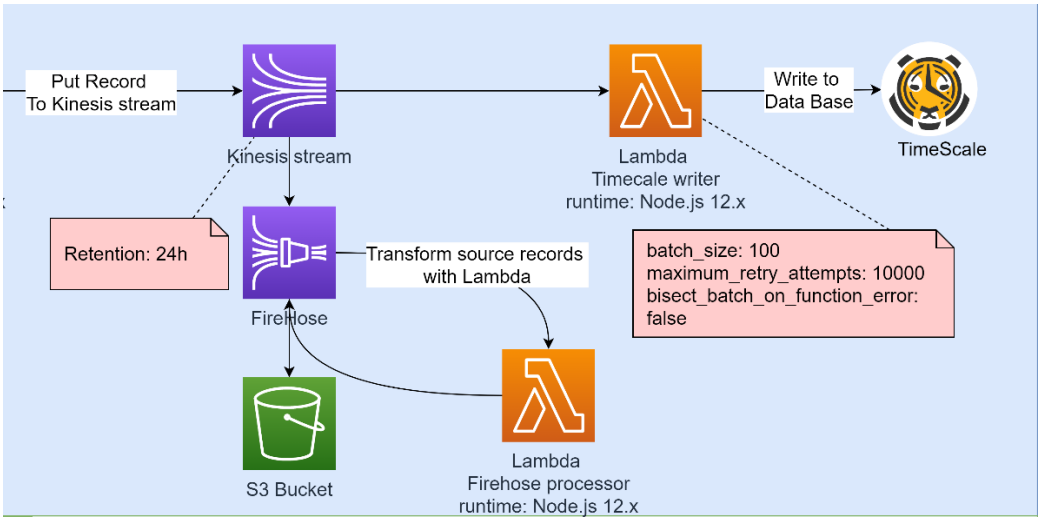

Figure 17. Kinesis Stream to data stores.

Kinesis Data Stream is acting as buffer in the architecture. It can hold messages until chosen retention period is reached, this can be up to one year. Currently it has been set to 24 hours. Kinesis Stream feeds Kinesis Firehose to transform and write objects to S3. Even if there is some availability issues with Firehose or S3 we can pretty much rely on that data will be stored because of the buffering capability of Kinesis Stream. If Firehose processor Lambda function is unavailable or there is any error with the execution itself, data will be stored in processing_failed folder of S3.

Timescale writer Lambda function is consuming Kinesis Data Stream in defined batch size, doing transformations and writing that batch of data to Timescale. In this case Lambda invocation is synchronous. If Lambda returns an error, it will be retried until maximum retry attempts or retention time is reached. When problems occur when function is writing to database, there is reasonable possibility that problems (e.g. temporary connection break) are solved before batch exceeds retention time or max retry attempts. In this case Lambda function is likely to be able to write the batch to Timescale, but if problem would rely on data content that would not be the case. For example, if one of the records in the batch is containing invalid JSON, whole batch of data would be missed from the database unless we

would have some extensive error handling inside the function. By setting “bisect batch on function error” to true, we could avoid the hassle as then every time function returns an error batch would be split into half. In the end we would able to successfully process all other records from the batch, except the one that was somehow erroneous. Of course, even then we might lose some valid data, as single record typically consist of multiple datapoints, in our case 100 datapoints. Also, it is to be noted that all of the data that we are missing with Timescale Writer Lambda can be retrieved later from S3. Anyhow, this will require some more processing and manual work to fill the data gaps in database. Such delay might also cause problems for some use cases. [32].

3.1.5 Monitoring and alarming

Current solution gathers a lot of log data to AWS CloudWatch from where many things can be monitored manually if wanted. In this chapter we will concentrate on automatic monitoring efforts which in this case will trigger alarms to defined Slack channel. Timescale database is having its separate monitoring strategy, which is not discussed here. Alerting efforts in the current solution are mainly about monitoring health of the pipeline, answering the question of “are we receiving data from source x and y” and then finding out possible errors in the pipelines. Some of the monitoring capabilities are built in downstream application (Grafana), where more detailed signal based monitoring is easy to build completely detached from the actual data pipeline.

From Kinesis Stream current pipeline actively monitors two things. Firstly, alarms will be triggered e.g. if Kinesis Stream starts to buffer data. This basically means that Timescale Writer Lambda is not able to keep up with the inflow of messages. Typical reason for this is that there is some slowness with the database. Another thing would be an unusually volatile burst of data or a lot of invalid records coming in for the lambda to process. Secondly, health of the whole pipeline is monitored by launching alarms if Kinesis Stream is not receiving data in 15 minutes period. If this alarm is triggered it means that there are some problems with upstream dataflow. Typical reason is network error or other reason that clients at the source are not able to send any data into the cloud.

From AWS Lambda, current pipeline triggers alarms if any of the lambda functions returns an error or timeouts. This is typically a good practice, but can sometimes lead into having

too many alarms popping up. Even if one Lambda run returns an error, data might be still flowing, as the retry of that Lambda might be successful. Thus single Lambda error is not always something that needs active actions.

Firehose Processor Lambda has also been set up to create custom Cloud Watch metric. It is recording from what powerplant each data batch had data from. Alarms are triggered if no data is being received from any of the powerplants in 15 minutes period. With this alarm, it is possible to find out if any individual powerplant is not successfully sending data to the platform.

3.2 Architecture improvements

In this chapter we are discussing on architecture improvements that would make data ingestion more reliable and easier to maintain. We will start with unifying pipelines in all environments, then going to technical raw data capture, to standard stream format, and improved S3 data usability, and improved error handling in Lambda functions.

3.2.1 Unifying data pipelines in all environments

We want to have identical data pipelines in all three environments starting right from IoT Core. This makes development easier and less prone to errors as we can test changes in test and development environments. For this need we can create new separate Terraform module which is not part of the actual data pipeline. Module will create two IoT Core rules to production environment (per MQTT topic). Those rules are attached with action to trigger Lambda function. Lambda function will then publish messages to IoT Core of another AWS account by assuming role which have been given the rights in the target account. Following setup is shown in the next figure.

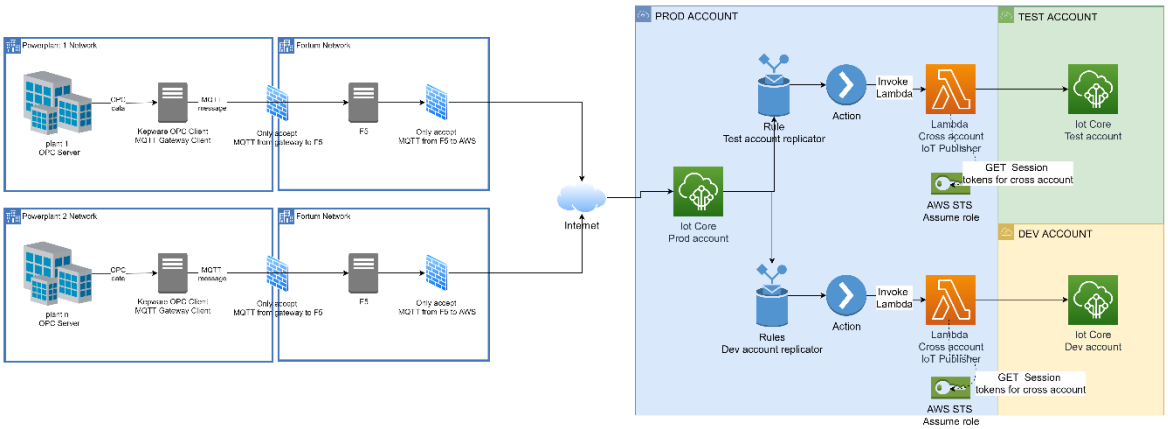

Figure 18. Cross account IoT Core publisher.

IoT Core rules of this module are by default set to be disabled and changes are ignored in Terraform lifecycle. This way developers can manually activate cross account IoT Publishing for desired data sources when it is needed. This will also decrease costs as we are not constantly doing same processing at all three environments.

3.2.2 Technical raw backups from IoT Core

To avoid permanent data losses that could occur due to failures happening before data has been successfully put into Kinesis Data Stream, we can create new IoT Core Rule with an action to write files into S3. With expected data volumes to ADP, we will create daily a few million files. Because of high number of files, S3 prefix (folder structure) is set to be time partitioned to minute level. It is good to note that the purpose of technical raw data backup is only to backfill some data gaps that are resulting from technical problems downstream before data is successfully stored to S3. Small file size and high number of files makes this bad candidate for other use. Following figure presents the architecture. MQTT messages are stored just as they are to S3 as JSON files.

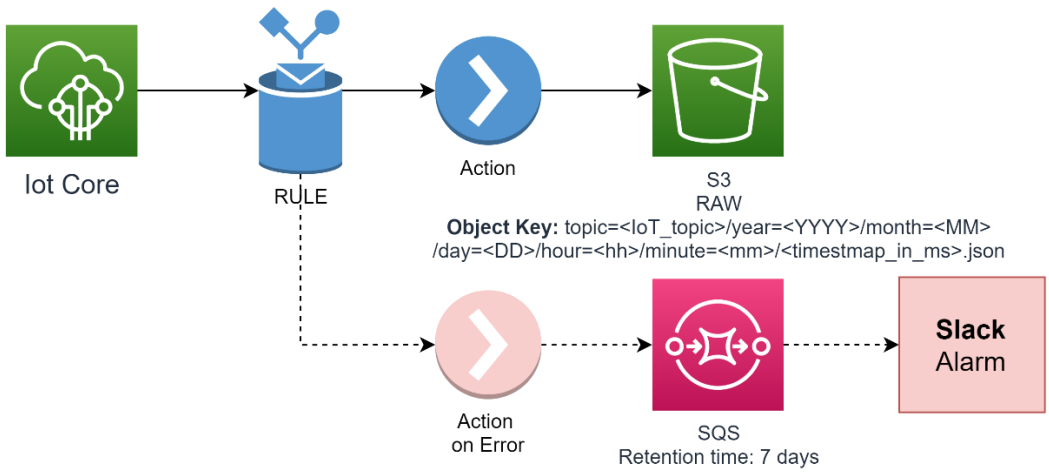

Figure 19. Technical raw backups from IoT Core.

For monitoring purposes we have introduced here "Action on Error" which is attached to the IoT Rule. If for any reason action to write file to S3 is failing, action on error is being triggered. Action on error will send the message to AWS Simple Queue Service (SQS) queue. Queue is monitored and if any messages are appearing to the queue an Alarm is triggered to Slack. SQS has been setup to hold the messages for seven days, which will give enough time for developers to debug the root cause and reprocess messages if needed. Action on Error behavior should be added to all IoT Rules, to increase visibility on failures.

3.2.3 Standard data format in Kinesis Stream

In order to decrease duplicate processing we can do all transformations before Kinesis Data Stream. This way we would have all data in standardized format already in Kinesis Stream. Same ideology can be applied throughout all source systems, not only bounded to data that is coming in with MQTT. Downstream processing from Kinesis Stream onwards is then not depending on the data source. Below is a sample of ADPs chosen "standard" data format with explanations.

```
{
   "signal_identifier": "nnnnn.opc.WTG19.WROT01.PthCylPres2",
   "value": 87.0,
   "source_quality_id": 1,
   "value_time": "2020-12-21T13:31:47.525Z",
   "update_time": "2020-12-21T13:31:48.906Z"
}

{
   "signal_identifier": "<source-system>.<data-pipeline>.<signal_location_code> ",
   "value": <value>,
   "source_quality_id": <source_quality_id>,
   "value_time": "<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>.<ttt>Z",
   "update_time": "<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>.<ttt>Z "
}
```

Signal_identifier is constructed from 3 parts which are making it globally unique. Source-system identifies production system, typically a power plant. Data-pipeline identifies the data pipeline from which data is acquired. Signal_location_code is unique identifier within a source system. It uniquely points to one location/measurement in source system. Value describes the value in that unique signal identifier. Source_quality_id indicates if measurement can be trusted according to the system that has done the measurement. "1" means that measurement is good, "0" that it is Bad and "-1" that it is unknown. Value_time is the timestamp of the measurement, meaning the moment when measured value was equal to the value presented in the data object. Update-time on the other hand describes the time when MQTT client sent the data towards ADP. Timestamps should be presented as UTC timestamps in the Kinesis Data Stream.

3.2.4 Improve usability of data stored in S3

There are 4 main things that will make data in S3 more usable (easier access with different tools, better performance, decreased cost). Firstly, as Kinesis Firehose nowadays supports Apache Hive compatible Amazon S3 custom prefixes let's take those into use. Practically this means that S3 prefix structure should include equal (=) signs. For example when in the earlier setup year and month in the prefix was stated as "2021/10", hive compatible format is "year=2021/month=10". This will make querying/fetching data with big data tools much more efficient. [33].

Secondly, let's start using larger files. It will again give performance boost when data is later use from S3 for analytical/data science purposes. As there is no time critical use defined for data put in S3, we could even wait longer time to batch up a larger data batch, without doing any harm to downstream processes. In our case data volumes are large and batching bigger files does not even mean extensive waiting. In the earlier architecture Firehose processor lambda was one of the limiting factors on file size as maximum size of Lambda request and response are 6 MB [34]. Now we don't have such limitation, as transformations are already done before data is put to Kinesis Stream. Finding good balance with having reasonable buffering time and file size may need some adjusting during the lifetime of data pipeline. As a rule of thumb, bigger file is better for downstream use cases and it is not likely that too big file size would cause any problems downstream as upper limit that Firehose can produce is 128 MB [35].

Third improvement for S3 usability is to start storing data in columnar data format called Apache Parquet. It will save space and thus queries will have better performance and reduced cost. Compression ratio of Parquet format compared to normal JSON data was about 1 to 7 when tested with sample of standard format ADP data [36].

Lastly we can change how AWS S3 server-side encryption is done. Earlier we have used normal object level keys from AWS Key Management Service (KMS) which will call KMS for every write and read request. This will start generating costs especially when we have large amount of objects, as KMS is billed by the amount of requests [37]. AWS has now introduced bucket keys which can reduce traffic form S3 to KMS by up to 99 %. This is important feature to take into use, as it can be only applied to new S3 objects. [38].

3.2.5 Dead-letter-queues for Lambda functions

Use of Dead-letter queues (DLQ) for all Lambda functions decreases the need for doing data gap fillings from S3 and makes it possible to detect exactly when data pipeline is not ingesting all data. Let's use the Lambda function that is writing data to Kinesis Data Stream as an example. This is presented in the following figure.

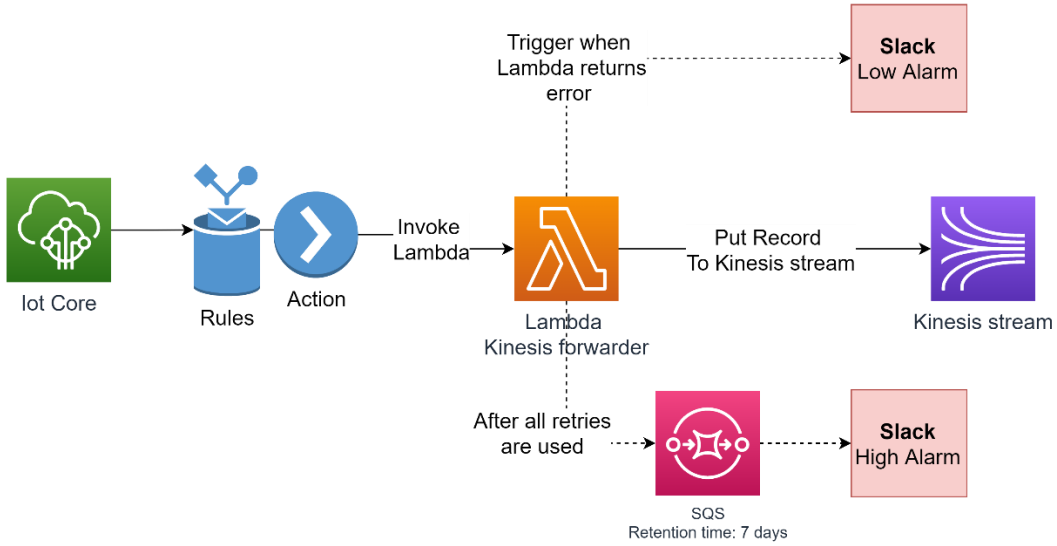

Figure 20. Dead-letter-queues for Lambda function.

Whenever Lambda function run returns an error, an alarm in Slack will be triggered. However, meaning of the alarm is changed from previous. Now it simply means that Lambda returned an error, there is no need for doing investigation on answering question “is some data missed from Kinesis Stream?”. Only if messages are landing to SQS (functioning as DLQ) we know that some data is missed from Kinesis Stream and immediate actions are needed to get that data processed. Again SQS has been setup to hold the messages for 7 days, which will give enough time for correcting actions.

3.3 Cost considerations

Cost is one remarkable factor when judging on what will be implemented and what should be kept alive. With high volumes of data, costs can quickly add up and thus it is helpful to thoroughly analyze cost factors of different components. In this section we will go through costs of running suggested improved architecture and see if there are any pitfalls in the design, cost wise. In the figure below there is simplified diagram of main pipeline (green box) and technical raw data pipeline (blue box) presented with cost components of each AWS resource.

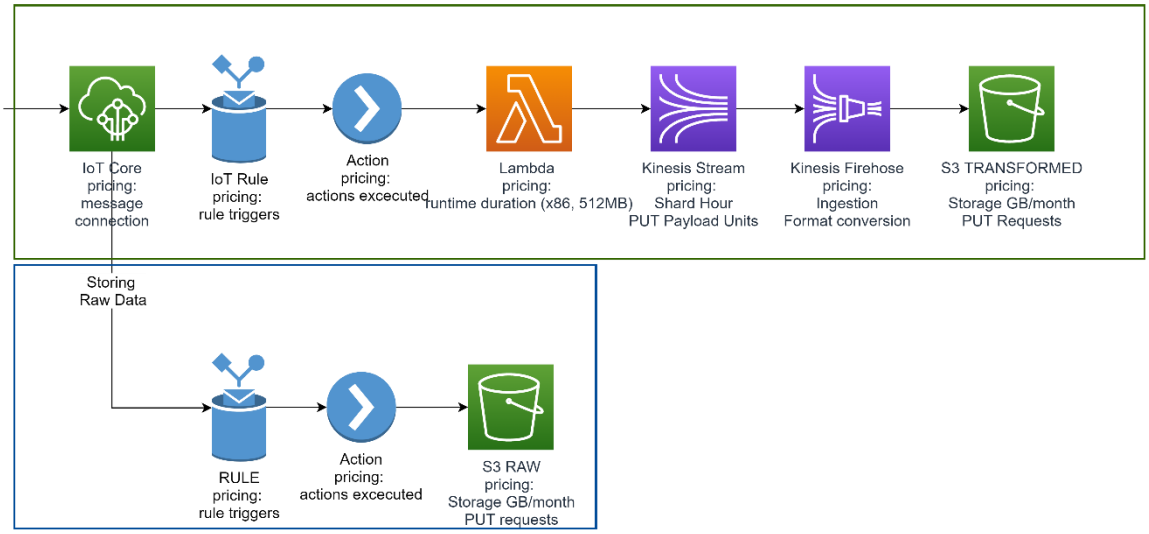

Figure 21. Main and raw data pipelines with cost components of each resource presented.

Pipeline costs are calculated in two ways: per million messages (where applicable) and for one wind farm with 21 turbines over 30 days period. In these calculations one MQTT message will consist of 100 measurement. Message size is 10.2 KB. From the wind farm we are receiving 3.24 million such MQTT messages every day. Prices are calculated for AWS "eu-west-1" region and AWS free tier is left out.

3.3.1 Main pipeline

First resource is AWS IoT Core, which is billed by number of messages sent in 5 KB increments and by connection minutes. In our case connection minutes represents the minority as there is only one device connected to IoT Core. Connection for one device for 30 days will cost under 0.01 dollars. Messages on the other hand are generating quite large bill, with the cost of 1 dollar per million messages. Thus single message of size 10.2 KB will be billed as 3 messages. [39].

IoT Core pricing for Rules and Actions is priced by million rules triggered / actions executed and with 5 KB message size increments. Both of those cost 0.15 dollars per million triggering/execution. In our case this means that single message will be billed by 3 Rules triggered and 3 actions executed. [39].

Lambda functions are priced by milliseconds of execution. When using x86 architecture and memory allocated to lambda function is set to 512 MB, price of millisecond is 0.0000000083 dollars. [40]. Our Lambda function running in Python 3.8 with 512 MB memory allocation runs in average in about 100 milliseconds. Data transfers for Lambda function is billed using EC2 pricing, for this Lambda function there is no data transfer expenses as data transfer "in" is free and also "out" transfer to Kinesis is free [40, 41]. Amazon CloudWatch is used for monitoring and logging purposes for Lambda function. Lambda function sends 8 metrics to CloudWatch which are billed monthly by 0.30 dollars each. Logs are priced as 0.57 dollars per GB ingested and 0.03 dollars per GB stored. In average this Lambda function is producing 0.5 GB of logs daily and logs are stored for 14 days. [42].

Lambda function also fetches transformation recipes or configuration data from DynamoDB. Cost impact of that will be low as Lambda function does not need to query DynamoDB table on every invocation. Configurations data from DynamoDB table can be stored in global variables in python code, which means that connections to DynamoDB needs to be established only on so called cold function starts. Because of this, we will need to provision only small number of DynamoDB Read capacity units (RCU) and minimal Write capacity units (WCU) to the table. Those are priced hourly at $0.000147 per RCU and $0.000735 per WCU. As of here we are storing just configuration data in DynamoDB the data volume is low, less than 1 MB. DynamoDB data storage price is 0.283 dollars per GB-month, so storage pricing in this case is not relevant. [43].

Kinesis Stream is priced by shard hour and PUT Payload Units. Single shard can ingest 1 MB/second or 1000 records/second and egress 2 MB/second and it is priced at 0.017 dollars/hour. For data coming from the wind park, we will need only one shard as we are using records aggregation in the feeding Lambda function. With the records aggregation we are able to send whole 100 measurements batch of data in a single "record". PUT Payload Units are counted in 25 KB chunks and the price for 1 million Units is 0.0165 dollars. This means that with records aggregation we are able to send whole 100 measurements data (18.4 KB after standardization done in Lambda function) batch using only single PUT Payload Unit. [44].

Kinesis Firehose is priced by data ingestion and format conversion. Both are billed in 5 KB increments. Because we are using Kinesis Stream as Firehose source we can expect that overhead towards 5 KB increments stays quite small. In calculations we are assuming that there is zero overhead. For data ingestion, the price is 0.031 $/GB and for format conversion it is 0.019 $/GB. [45].

Finally data will be put and stored to S3, which is priced by PUT requests and storage. For S3 Standard, price per GB stored for month is 0.023 dollars. For PUT requests price is 0.005 dollars per 1000 requests. If we set our firehose parameters in a way that our file size would be approximately 10 MB, and taking into account compression ratio (about 7/1 from standard format kinesis stream), number of PUT requests stays reasonably low. [46].

Following figure will present cost breakdown of the main pipeline for 1 million messages when time dependent factors are left out (CloudWatch metrics and store, DynamoDB, Kinesis Stream shard hours, S3 Storage).

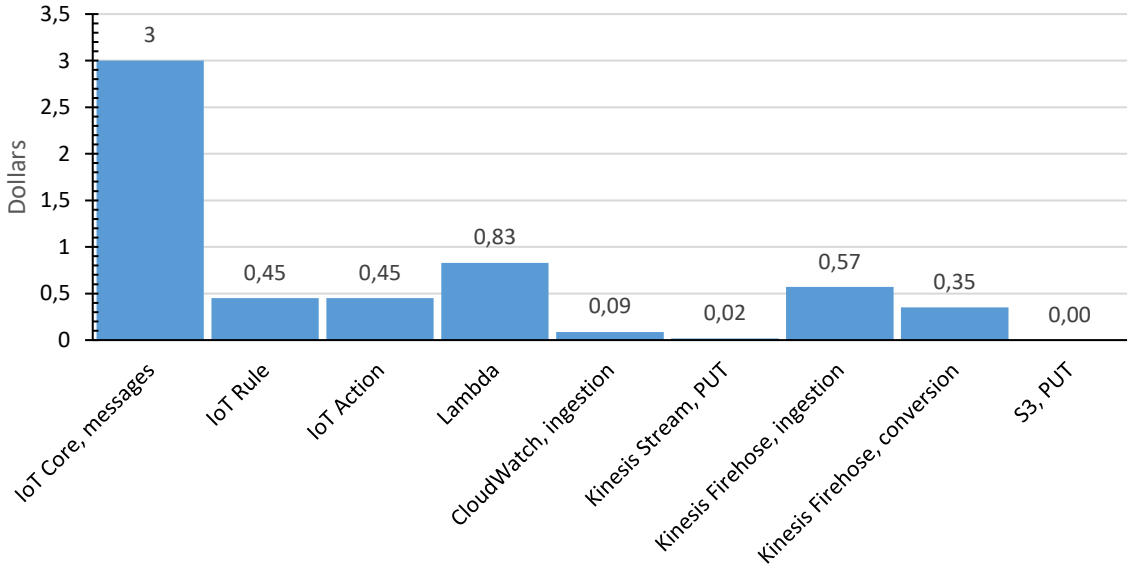

Figure 22. Main pipeline cost for million messages (include only event based cost components).

Total cost of 1 million messages is 5.76 dollars. As we can see from the figure, IoT Core message costs are generating over 50 % of total cost. In our case it would be easy to decrease IoT Core related costs (message, Rule and Action) by almost one third. Currently our

message size is really inconvenient from pricing perspective, as it is only 0.2 KB over two "full" messages of 5 KB.

When we expand our view to wind farm level and calculate costs for 30 days, including also time billed services, we will end up to total cost of 575 dollars per month (30 days). Below figure shows breakdown of AWS service costs from that situation.

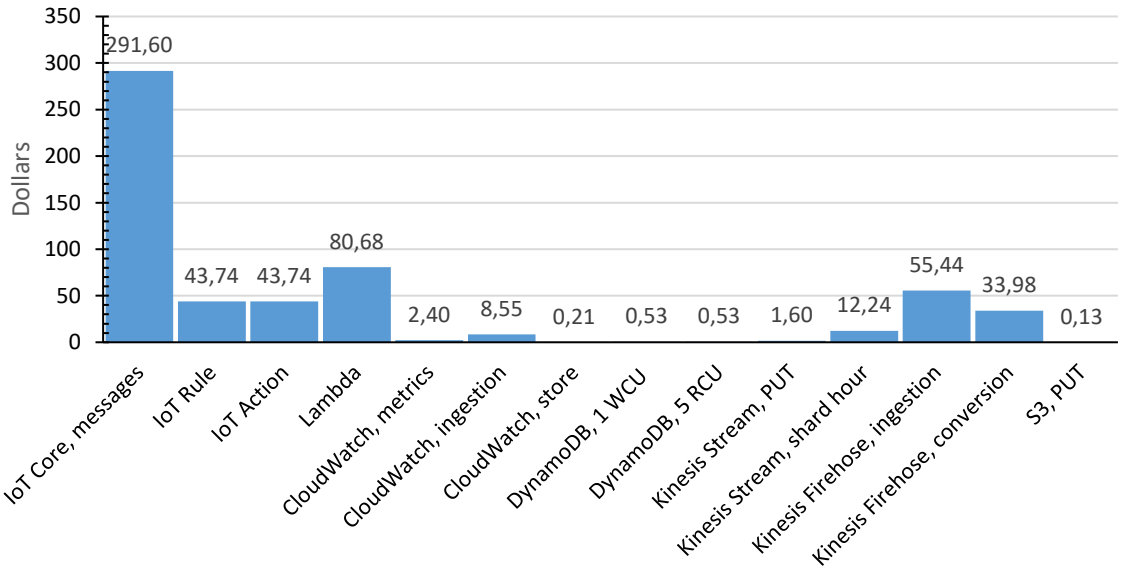

Figure 23. Main pipeline cost for one wind farm over 30 days.

It is clear from the figure that time billed components are creating relatively small amount of costs in our case. From cost perspective IoT services are dominant, and as said earlier it would be easy task to cut that cost by almost one third. After that, optimizing Lambda function performance has big potential on cost savings. As of now the function is not performance tuned.

Costs that are coming from Kinesis Firehose are directly proportional to ingested data volume. This is good to note when one is deciding on naming conventions. In our case the naming convention introduced in section 3.2.3 is not optimal. Chosen keys are long, for example we could use simple "id" instead of long "signal_identifier". If we would only use 1-2 characters long keys, we would save about 5 KB per every 100 measurements. This would translate into 24 dollars saved of Kinesis Firehose costs every month for the wind farm data.

Last cost component of the main pipeline is the cost of data stored in S3. Storage cost of S3 is a cumulative cost component, as data will not expire after specified time period. Main pipeline will produce about 255 GB of Parquet formatted data to S3 every month. For S3 standard tier price per GB-month is set to 0.023$ [46]. Following figure will show how monthly price is evolving when data is cumulating to S3. Figure will also show price for data that would have been in JSON format.

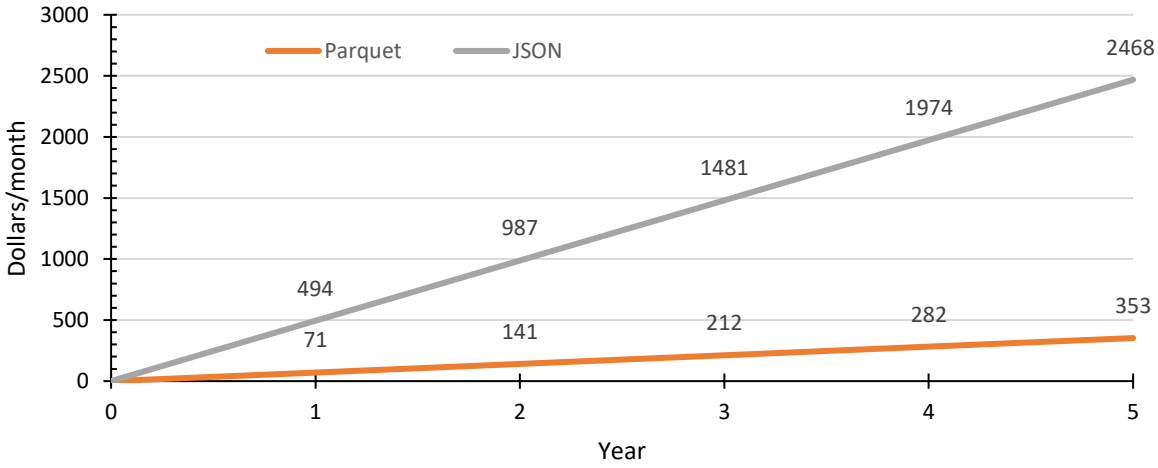

Figure 24. S3 Storage price cumulation for the wind farm data.

When data is stored in Parquet format, costs are staying in manageable level. After pipeline has been running for the first year, monthly storage cost will be at 71 dollars/month and it will continue increasing by about 6 dollars/month. If one would use JSON format instead, monthly storage cost would overrun other pipeline costs soon after first year of running the pipeline. After all, data can be moved into cheaper S3 storage tiers, if access needs to the data decreases over time. GB-month pricing for S3 Standard Infrequent Access is $0.0125, for S3 Glacier $0.004 and for S3 Glacier Deep Archive $0.00099 [46].

3.3.2 Raw data pipeline

For raw data pipeline, price of using IoT Rule and IoT Action is exactly the same as for main pipeline. S3 PUT requests will generate long bill in this case, as every message received in IoT Core will eventually trigger S3 PUT request. For million S3 PUT request cost is 5 dollars. Raw data pipeline cost breakdown for the wind farm data (30 days) is presented in figure below.

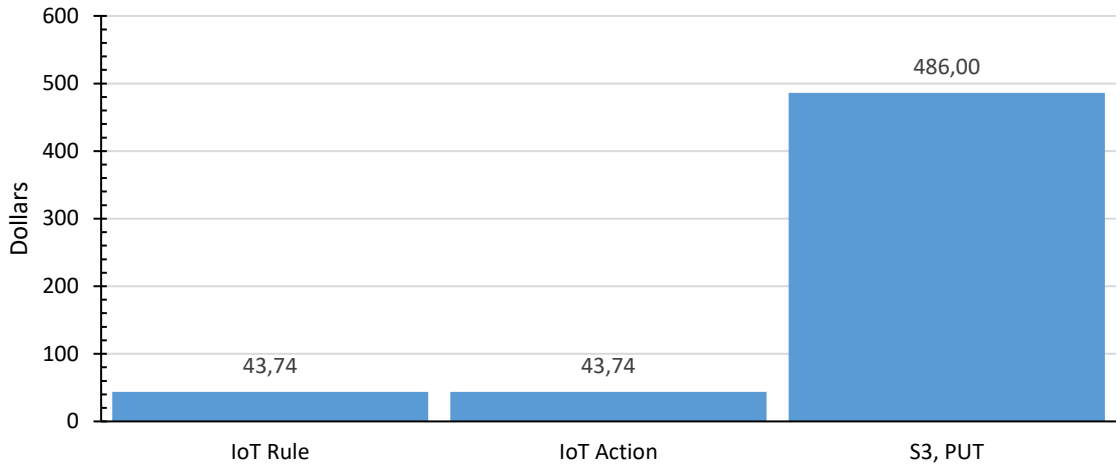

Figure 25. Raw data pipeline (direct S3 action) cost for the wind farm over 30 days.

Total cost of raw data pipeline is high, at 573 dollars/month for one wind farm data. It is about double the price of main pipeline, if IoT Core message cost is not counted to be solely cost of main pipeline.

From price perspective raw data pipeline is not in acceptable level. In order to reduce S3 PUT requests, we can use streaming approach and design raw data pipeline again. IoT Action offers possibility to directly put messages to Kinesis Stream. By utilizing that functionality, redesigned raw data pipeline would be as follows.

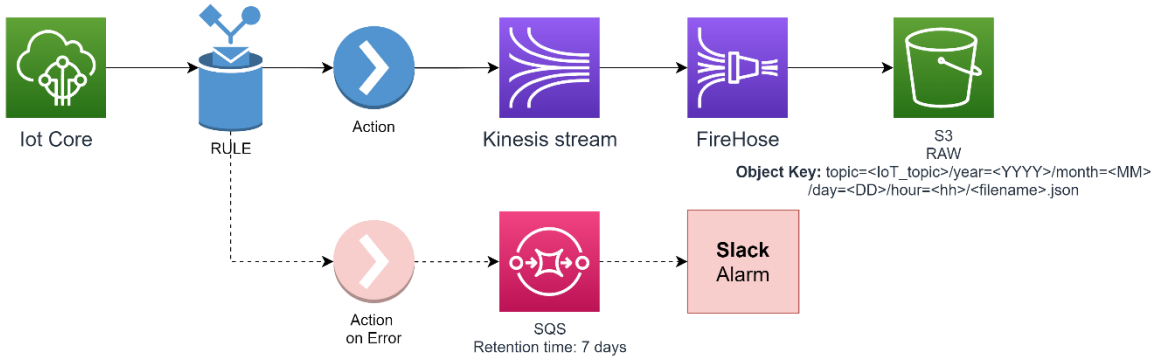

Figure 26. Redesigned technical raw data pipeline.

From cost perspective redesigned pipeline would look like in the figure below.

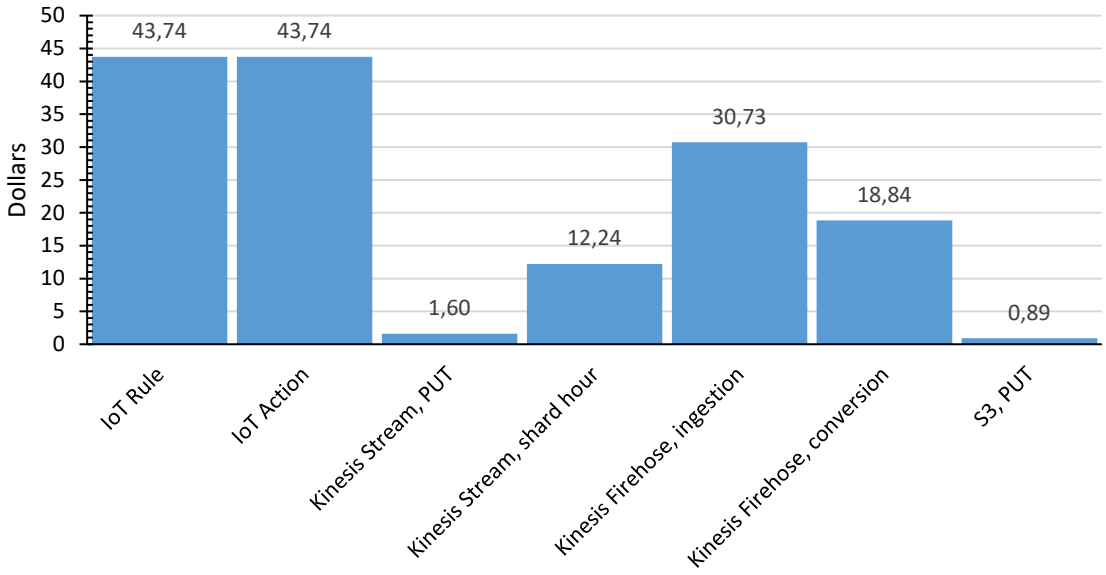

Figure 27. Redesigned raw data pipeline cost for the wind farm over 30 days.

With the redesigned pipeline, cost of raw data pipeline is more manageable at 152 dollars/month. Added benefit of new design is that when raw data is actually needed, it will be accessible in larger files, which improves performance.

Raw data is stored to S3 in JSON files. This means that the data volume will be increasing quickly. Setting suitable lifecycle policies to raw data is thus highly recommended.

4 DISCUSSION AND CONCLUSIONS

This chapter presents discussion about IoT data ingestion using AWS managed services and the main findings of the thesis work. Thoughts on the topic are discussed in the last paragraph as well as the development opportunities in the future.

4.1 Discussion

Using AWS managed services is an extremely fast and easy way to set up a scalable and reliable data ingestion pipeline. This can be done with zero upfront costs, which is a great benefit compared to other approaches, such as running pipelines in virtual machines, or containers in either AWS or on-premises data center. The ingestion pipeline in AWS is easy to equip with desired capabilities like buffering (Kinesis Stream), error handling (DLQs), and alarming. With managed services, even a small team is able to create highly available ingestion pipelines with minimal Recovery Point Objective.

When designing a large scale ingestion pipeline, it is good to keep in mind that costs will easily add up over time. Costs for a single wind farm may not come up to be significant, but when the same solution is taken into use for 10 or 50 similar wind farms, the total cost will definitely be noticeable. Reviewing drafted architectures in detailed level may reveal significant shortcomings in the design from cost perspective. Some of them require only small configuration changes. For example optimizing MQTT message size to nearest 5 KB, is having saving capability of about 20 % of main pipeline cost for one wind farm. Some, on the other hand, need complete redesigning, like happened with the first design of taking technical raw backups. S3 PUT requests are relatively expensive as creating one million objects costs five dollars. Redesigning raw data capture, from using direct S3 PUT request to streaming approach, can save about 70 % of raw data pipeline cost for one wind farm.

Many of AWS managed services are priced by the amount of data. Different measures to reduce data volume should be considered already at the source, like reducing noise in the data and selecting suitable sampling rate. However, it is good to note that with high data volumes also naming comes with a cost. How much can be seen reasonable to pay e.g. for descriptive naming? For example, using 1-2 character long keys compared to the use of descriptive key names, when data is standardized, would lower main pipeline cost by about

6 % for one wind farm (after MQTT message size is optimized). In addition to that, more costs would be created from downstream processes when descriptive naming is used. In ideal world, effective naming standards would have been applied to the data already at the source.

When storing data, it should be done in columnar format e.g. Apache Parquet. Monthly storage cost of storing data in JSON format would be nearly 500 dollars for one wind farm, after one year. However, when same data would be stored in Parquet format, monthly cost would be only 71 dollars. This is because of data compression benefits when using Parquet. Columnar format is also beneficial when data is actually used from the storage. Data can be read solely from specified columns, when using columnar format. This further decreases the amount of data that needs to be read from storage, giving performance and cost saving benefit.

In addition to what already exist, AWS is also evolving quickly. New valuable features are being brought to the AWS eco-system in fast pace. During the writing process of this thesis (almost two years), AWS has launched many useful features. For example, for S3 AWS has launched bucket keys, Lambda function pricing changed from 100 millisecond accuracy to one millisecond, and Kinesis Firehose enabled hive compatible partitioning of data. These are only some examples that were actually used in Fortum's case. Kinesis Firehose now enables data content based dynamic partitioning which is one of the latest new features released. It would give a lot of flexibility on how data is organized in S3 bucket. This would also enable the creation of consumption optimized partitions easily using Kinesis Firehose. Gaining knowledge on the real costs of this new feature would however require some testing, as the pricing is a little bit ambiguous. The cost is based on GBs delivered to S3, number of objects, and JQ processing hours. The cumulation of processing hours is not clear at a glance. [45].

AWS has also launched a new Graviton2 processors, which are using ARM-based processor architecture. These processors can also be used for Lambda functions, and AWS is promising up to 34 % better price/performance with Graviton2. It would be nice to take this widely in use for all Lambda functions, if the benefit is really there. [47].

When using AWS serverless stack, it is good to be open for trying out and evaluating new features. Components and pipelines should be designed in a way that makes replacing them

less painful, and enables changes to be done without noticeable service breaks. This way one can benefit the most from AWS.

4.2 Conclusions

Industrial data can be utilized in multiple different ways varying from more conventional reporting use cases, to situational awareness applications all the way to predictive maintenance and operational optimization. Putting use cases together, usually requires data from multiple sources. Bringing data to a single place, liberates it from underlaying siloed and critical processes. When data is in a single place in standardized format, it can be effectively used for different kind of applications. Each use case will have its specific needs for end-to-end latency. One use case can work perfectly fine using multiple hours old data, when other would benefit from sub-second latency. This is good to note when creating connections from silos to a single place. If data is brought in by using event based principles, there might be more use cases for it, if not at the time, maybe in the near future.

Main characteristic for industrial data is the abundance of it. Moreover, data volumes will continue increasing in future as transition towards more frequent sampling rates is ongoing. The vast majority of industrial data is timeseries data produced by sensors, actuators, and measurement devices located in different parts of the process. From cost perspective, it can be crucial to have change-triggered data sending in place, or selecting meaningful data sampling rate. Avoiding noise in the data is equally important. Water temperature measurement probably does not need to be with seven decimals. When talking about storing large data volumes, it is also a good idea to start using performant columnar data formats from the start. As mentioned earlier, with big data quantities, it is justified to consider even naming conventions from price perspective. Descriptive naming might come with a hefty cost over time.

AWS provides us with great managed services which can be used to build data ingestion pipelines in a serverless manner. It is extremely fast and easy way to set up scalable and reliable data ingestion pipeline with no upfront costs. Scalable and reliable in a way that even a small team can develop and operate it. New valuable features are pouring into AWS,

which should also be kept in mind. It is advisable to design components and pipelines in a way they can be easily replaced with better ones.

4.3 Future work

One interesting development path would be improving capabilities at the edge. At the moment there is constant pressure to take more and more IoT data to the cloud. Costs are driven mostly by the amount of data that is send to cloud. How could we send less data without having to compromise on the value it brings and how it would be done in a clever way?

First of all, data could be standardized at the edge. This would definitely streamline pipelines in cloud. Also, size of messages would decrease at the same time, if standardized naming is designed to be compact. One could even evaluate use of compression at the edge. Managing and configuring these edge devices could be also done directly from cloud.

It would be interesting to see how ML models can be utilized. At some point, could we use ML models at the edge and detect actual events on the spot. Instead of sending large constant data stream, only detected events or measurements during the event would be sent to cloud. There would be a lot to accomplish at the edge.

REFERENCES

1. Gupta, G. 2018. Practical Enterprise Data Lake Insights: Handle Data-Driven Challenges in an Enterprise Big Data Lake. 1st ed. Berkeley, CA: Apress L. P. p. 21-31, 52-63.

2. Boyes, H. et al. 2018. "The industrial internet of things (IIoT): An analysis framework".

3. Ullah, M. et al. 2020. ”Twenty-one key factors to choose an IoT platform: Theoretical framework and its applications,” IEEE Internet of Things Journal.

4. Juhanko, J. et al. 2015. ”Suomalainen teollinen internet – haasteesta mahdollisuudeksi: taustoittava kooste”. ETLA Raportit No 42.

5. Treder, M. 2019. Becoming a Data-Driven Organisation Unlock the Value of Data. 1st ed. 2019. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019. Web.

6. Mobley, K. 2002. An Introduction to Predictive Maintenance. Elsevier Science (USA). p.43-49 https://books.google.fi/books?hl=en&lr=&id=SjqXzxpAzSQC&oi=fnd&pg=PP1&dq=predictive+maintenance&ots=iGoRSACqgh&sig=HwyOFVnBKi8QySn3Qk3U2JdpXwA&redir_esc=y#v=onepage&q=predictive%20maintenance&f=false

7. Collin, J. & Saarelainen, A. 2016. Teollinen internet. Helsinki: Talentum. p. 57-60

8. Kuronen, J. 2020. E2E H&C Digital Twin Project kick-off. Presentation of E2E H&C Digital twin to ADP project team. Unpublished

9. Coughlin, T. 2020. HDD Market History And Projections. [referred: 22.11.2020]. Available: https://www.forbes.com/sites/tomcoughlin/2020/05/29/hdd-market-history-and-projections/?sh=2962f9d06682

10. OPC Foundation. History. [referred: 22.11.2020]. Available: https://opcfoundation.org/about/opc-foundation/history/

11. Nicola, M. et al. 2018. SCADA Systems Architecture Based on OPC and Web Servers and Integration of Applications for Industrial Process Control. International Journal of Control Science and Engineering 2018, 8(1): 13-21. DOI: 10.5923/j.control.20180801.02.

12. OPC Foundation. 2017. OPC 10000-6: OPC Unified Architecture, Part 6: Mappings. Release 1.04. [referred: 8.12.2020]. Available: https://reference.opcfoundation.org/v104/Core/docs/Part6/5.4.1/

13. Huhta, J. 2010. Tärkeimpien ICS-tietoturvastandardien soveltaminen Fortumissa. Tietoturvaa teollisuusautomaatioon (TITAN) – Seminaari.

14. IoT Gateway, Kepserverex advanced plug-ins. [referred: 3.12.2020]. Available: https://www.kepware.com/en-us/products/kepserverex/advanced-plug-ins/iot-gateway/

15. Naik, N. 2017. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, 2017, pp. 1-7, doi: 10.1109/SysEng.2017.8088251.

16. Wolf, M. & Wicksteed, C. 1997. Date and Time Format. W3C. [referred: 22.11.2020]. Available: https://www.w3.org/TR/NOTE-datetime

17. VGB-Standard. 2018. KKS Identification System for Power Stations - 8th revised edition 2018. VGB PowerTech e.V. ISBN 978-3-96284-043-3

18. Infrastructure as a Service (IaaS) Market. Markets and Markets. [referred: 22.12.2020]. Available: https://www.marketsandmarkets.com/Market-Reports/infrastructure-as-service-market-262058075.html

19. Gartner Says Worldwide IaaS Public Cloud Services Market Grew 37.3% in 2019. 10.8.2020. Gartner. [referred: 22.12.2020]. Available:

https://www.gartner.com/en/newsroom/press-releases/2020-08-10-gartner-says-worldwide-iaas-public-cloud-services-market-grew-37-point-3-percent-in-2019

20. Shared Responsibility Model. Amazon Web Services. [referred: 2.1.2021]. Available: https://aws.amazon.com/compliance/shared-responsibility-model/

21. Ruback, H. & Richards, T. 2021. Applying the AWS Shared Responsibility Model to you GxP Solution. AWS for Industries. [referred: 8.12.2021]. Available: https://aws.amazon.com/blogs/industries/applying-the-aws-shared-responsibility-model-to-your-gxp-solution/

22. Turunen, M. 2020. Asset Data Platform. Fortum. Unpublished

23. Language Runtime Performance. Amazon Web Services. [referred: 28.9.2021]. Available: https://docs.aws.amazon.com/whitepapers/latest/serverless-architectures-lambda/language-runtime-performance.html

24. AWS IoT Core Service Level Agreement. Amazon Web Services. [referred: 29.9.2021]. Available: https://aws.amazon.com/iot-core/sla/

25. Amazon Kinesis Service Level Agreement. Amazon Web Services. [referred: 29.9.2021]. Available: https://aws.amazon.com/kinesis/sla/

26. Amazon S3 Service Level Agreement. Amazon Web Services. [referred: 29.9.2021]. Available: https://aws.amazon.com/s3/sla/

27. AWS Lambda Service Level Agreement. Amazon Web Services. [referred: 29.9.2021]. Available: https://aws.amazon.com/lambda/sla/

28. Amazon DynamoDB Service Level Agreement. Amazon Web Services. [referred: 29.9.2021]. Available: https://aws.amazon.com/dynamodb/sla/

29. Amazon Compute Service Level Agreement. Amazon Web Services. [referred: 29.9.2021]. Available: https://aws.amazon.com/compute/sla/

30. Best Practices for Using MQTT Topics in the AWS IoT Rules Engine. Amazon Web Services. [referred: 29.9.2021]. Available:

https://docs.aws.amazon.com/whitepapers/latest/designing-mqtt-topics-aws-iot-core/best-practices-for-using-mqtt-topics-in-the-aws-iot-rules-engine.html

31. Asynchronous invocation. AWS Lambda, Developer Guide. Amazon Web Services. [referred: 30.9.2021]. Available: https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#invocation-async-errors

32. Using AWS Lambda with Amazon Kinesis. AWS Lambda, Developer Guide. Amazon Web Services. [referred: 30.9.2021]. Available: https://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html

33. Chakrabarti, R. 2019. Amazon Kinesis Data Firehose custom prefixes for Amazon S3 objects. AWS Big Data Blog. [referred: 3.10.2021]. Available: https://aws.amazon.com/blogs/big-data/amazon-kinesis-data-firehose-custom-prefixes-for-amazon-s3-objects/

34. Amazon Kinesis Data Firehose Data Transformation. Amazon Kinesis Data Firehose, Developer Guide. Amazon Web Services. [referred: 3.10.2021]. Available: https://docs.aws.amazon.com/firehose/latest/dev/data-transformation.html

35. Data Delivery and Destinations. Amazon Kinesis Data Firehose FAQS. Amazon Web Services. [referred: 3.10.2021]. Available: https://aws.amazon.com/kinesis/data-firehose/faqs/

36. Converting Your Input Record Format in Kinesis Data Firehose. Amazon Kinesis Data Firehose, Developer Guide. Amazon Web Services. [referred: 3.10.2021]. Available: https://docs.aws.amazon.com/firehose/latest/dev/record-format-conversion.html

37. AWS Key Management Service pricing. AWS Key Management Service. Amazon Web Services. [referred: 5.10.2021]. Available: https://aws.amazon.com/kms/pricing/

38. Amazon S3 Bucket Keys reduce the costs of Server-Side Encryption with AWS Key Management Service (SSE-KMS). 2020. About AWS. Amazon Web Services. [referred: 5.10.2021]. Available: https://aws.amazon.com/about-aws/whats-new/2020/12/amazon-s3-bucket-keys-reduce-the-costs-of-server-side-encryption-with-aws-key-management-service-sse-kms/

39. AWS IoT Core pricing. Internet of Things Products. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/iot-core/pricing/

40. AWS Lambda pricing. Compute Products. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/lambda/pricing/

41. Amazon EC2 On-Demand Pricing. Compute Products. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

42. Amazon CloudWatch pricing. Management Tools. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/cloudwatch/pricing/

43. Pricing for Provisioned Capacity. Database Products, Amazon DynamoDB. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/dynamodb/pricing/provisioned/

44. Amazon Kinesis Data Streams pricing. Analytics Products, Amazon Kinesis. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/kinesis/data-streams/pricing/

45. Amazon Kinesis Data Firehose pricing. Analytics Products, Amazon Kinesis. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/kinesis/data-firehose/pricing/

46. Amazon S3 pricing. Storage Products, Amazon S3. Amazon Web Services. [referred: 10.10.2021]. Available: https://aws.amazon.com/s3/pricing/

47. Achieve up to 34% better price/performance with AWS Lambda Functions powered by AWS Graviton2 processor. 2021. About AWS. Amazon Web Services. [referred: 12.10.2021]. Available: https://aws.amazon.com/about-

aws/whats-new/2021/09/better-price-performance-aws-lambda-functions-aws-graviton2-processor/