LUT University
School of Energy Systems
Electrical Engineering

Jesse Juuti

DEFINITION AND IMPLEMENTATION OF GENERAL-PURPOSE IOT CLOUD BACKEND

Examiners: Professor Pertti Silventoinen
M.Sc. Mikko Nykyri

Supervisor: M.Sc. Tero Nordström

Abstract

Author: Jesse Juuti

Title: Definition and implementation of general-purpose IoT cloud backend

Year: 2020 **Place: Salo**

Master's Thesis. LUT University, Electrical Engineering.
48 pages
Examiners: Professor Pertti Silventoinen, M.Sc. Mikko Nykyri
Supervisor: M.Sc. Tero Nordström

Keywords: IoT, cloud services, cloud backend, serverless
Hakusanat: IoT, pilvipalvelut, pilvitaustapalvelu, serverless

As a goal of this thesis is to define and implement general-purpose IoT cloud backend. Thesis work utilized cloud services provided by cloud service provider called Amazon Web Services (AWS). The commissioner of this thesis, SADE Innovations Oy, defined a set of requirements which should be met in implemented cloud backend. The core idea was that the implemented cloud backend should be as general-purpose as possible so that the commissioner could use it as basis of IoT related client projects. The implementation is meant to be expanded with client specific requirements.

First thesis focuses on set requirements by the commissioner and reasoning behind them. After requirements, thesis goes through technical backgrounds and technologies which are related to defining and implementing the general-purpose IoT cloud backend. The chapter focuses also to briefly present different kind of cloud services provided by AWS. Next, thesis focuses on actual definition and implementation work. In the definition and implementation chapter, thesis opens key points in the general-purpose cloud backend. At the end, thesis goes through analysis from perspectives of meeting the set requirements, future development possibilities and the benefits of the implementation.

As a main result of the thesis, a general-purpose IoT cloud backend was defined and implemented. The implementation met the set requirements well. The resulted implementation has been used as basis of several projects of the commissioner's clients. Based on already realized projects, it can be stated that the implementation works well and provides benefits to clients as savings in time and money which would have otherwise gone into implementing similar project basis. The commissioner gets benefit from license revenues of realized client projects which use the implementation as basis of the project.

Tiivistelmä

Tekijä: Jesse Juuti

Työn nimi: Yleiskäyttöisen IoT pilvitaustapalvelun määritys ja toteutus

Vuosi: 2020 **Paikka: Salo**

Diplomityö. Lappeenrannan-Lahden teknillinen yliopisto LUT, Sähkötekniikka.
48 sivua
Tarkastajat: Professori Pertti Silventoinen, M.Sc. Mikko Nykyri
Ohjaaja: M.Sc. Tero Nordström

Hakusanat: IoT, pilvipalvelut, pilvitaustapalvelu, serverless
Keywords: IoT, cloud services, cloud backend, serverless

Tässä diplomityössä tavoitteena oli määritellä ja toteuttaa yleiskäyttöinen IoT pilvitaustapalvelu. Työssä hyödynnettiin Amazon Web Services (AWS) nimisen pilvipalvelutarjoajan tarjoamia pilvipalveluita. Työn toimeksiantaja, SADE Innovations Oy, määritteli vaatimukset, jotka pilvitaustapalvelun tulisi mahdollisimman hyvin toteuttaa. Keskeisenä ideana oli, että toteutus olisi mahdollisimman yleiskäyttöinen, jotta toimeksiantaja voisi käyttää toteutusta yleisenä pohjana IoT maailmaan liittyvissä asiakasprojekteissaan. Toteutusta on tarkoitus laajentaa edelleen asiakkaiden omilla yksilöllisillä vaatimuksillaan.

Ensiksi työssä perehdytään toimeksiantajan vaatimuksiin ja niiden perusteluihin. Sen jälkeen työssä käydään läpi teknisiä taustoja ja teknologioita, joita tavoitteena olevan yleiskäyttöisen IoT pilvitaustapalvelun toteuttamiseen liittyy. Teknisissä taustoissa tutustutaan myös lyhyesti erilaisiin AWS:n tarjoamiin pilvipalveluihin. Teknisten taustojen jälkeen perehdytään itse määrittelyyn sekä toteutukseen. Määrittely ja toteutus luvussa avataan keskeisimpiä asioita yleiskäyttöisessä IoT pilvitaustapalvelussa. Lopuksi toteutusta analysoidaan asetettujen vaatimusten täyttymisen, kehityskohtien sekä hyötyjen näkökulmista.

Työn keskeisenä tuloksena syntyi tavoitteena ollut yleiskäyttöinen IoT pilvitaustapalvelu, joka toteutti asetetut vaatimukset hyvin. Tuloksena syntynyt toteutus on ollut jo pohjana useissa eri toimeksiantajan asiakasprojekteissa. Toteutuneissa asiakasprojekteissa on voitu todeta, että toteutus toimii hyvin ja se tuottaa asiakkaalle hyötyjä muun muassa säästyneenä työaikana sekä rahana, jota vastaavan järjestelmän kehittäminen alusta asti vaatisi. Toimeksiantaja hyötyy toteutuksesta toteutuneiden asiakasprojektien tuomina lisenssituloina.

Preface

This thesis was written while being employed by the commissioner of the thesis, SADE Innovations Oy. Writing my thesis would have not been possible in this time frame without having possibility to focus on the thesis almost fully. Therefore, a word of gratitude to my employer and manager is in place for providing the subject of the thesis and enabling this opportunity to focus on it. I would also like to thank all of my co-workers for numerous professional discussions which have contributed to building my overall understanding of the subject in question.

I would also like to thank professor Pertti Silventoinen and M.Sc. Mikko Nykyri for examining the thesis and guiding me throughout the writing process. Thanks also to M.Sc. Tero Nordström for acting as representative of the commissioner, supervisor of the thesis and for guiding me during the entire process.

Naturally, the greatest thanks to my spouse and family for continuous support during many years. First the workload of the thesis seemed to be huge but eventually remaining workload begun to reduce and now I am glad that this day arrived when I can proudly say that the thesis is finally ready.

Jesse Juuti
June 6, 2020
Salo, Finland

Table of contents

1 Introduction 3

1.1 Research problem 4

1.2 Research scope and goal 4

1.3 Research methods 4

2 Requirement specification and evaluation criteria 5

2.1 Functional requirements 5

2.2 Non-functional requirements 5

2.3 Reasoning behind the requirements 6

3 Technical backgrounds 8

3.1 Application programming interfaces 8

3.1.1 GraphQL 8

3.1.2 REST 9

3.1.3 WebSocket 10

3.2 MQTT 11

3.3 Serverless 11

3.4 Serverless Framework 11

3.5 TypeScript 12

3.6 Amazon Web Services 12

3.6.1 Amazon API Gateway 12

3.6.2 Amazon Cognito 13

3.6.3 Amazon DynamoDB 13

3.6.4 Amazon S3 13

3.6.5 AWS AppSync 14

3.6.6 AWS Identity and Access Management 14

3.6.7 AWS IoT Core 14

3.6.8 AWS Lambda 15
4 Definition and implementation 16
4.1 Architecture 16
4.2 IoT device management with IoT Core 17
4.3 GraphQL API with AppSync 18
4.3.1 Sensor data storage mutation 18
4.3.2 Sensor data query 20
4.3.3 Sensor data subscription 21
4.3.4 Device state query 21
4.3.5 Device state mutation 23
4.3.6 Device state subscription 24
4.4 User and access management with Cognito 26
4.5 Data handling with Lambda functions 27
4.5.1 Sensor data handling for storage to DynamoDB 27
4.5.2 Sensor data handling for storage to S3 28
4.5.3 Device state query handling 29
4.5.4 Device state update handling 29
4.5.5 Device state modification handling 29
4.5.6 Device provisioning handling 30
4.6 Data storage with DynamoDB and S3 30
4.6.1 DynamoDB 31
4.6.2 S3 31
4.7 Serverless Framework template 31
4.8 IAM roles and permissions 32
5 Analysis of the implementation 33
5.1 Meeting the requirements 33

5.1.1 Functional requirements 33
5.1.2 Non-functional requirements 34
5.2 Benefits of the implementation 36
5.3 Future improvement areas 37
6 Conclusions and discussion 39
References 40

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IT	Information Technology
JSON	JavaScript Object Notation
NoSQL	Not SQL
MQTT	Message Queuing Telemetry Transport
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Querying Language
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier

1 Introduction

Era of cloud computing has been going on for around a decade now. Growing numbers of companies are either adapting or have already adapted into working with cloud services (Brinda and Heric, 2017). The cloud operating model has been characterized as certain kind of Pandora's box and as such it is feared to reveal all the issues in the information technology industry (Heino 2010). On the other side of the coin there are promises of great potential in the form of seemingly unlimited computing resources, worldwide and around the clock accessibility, pay-as-you-go pricing and no infrastructure setup or maintenance costs to name a few. By managing own IT resources, a company can bind vast amount of capital and generate maintenance costs (Salo, 2010). Besides, utilization rates of IT resources tend to be low because resources are often over dimensioned by rare peak load times which lead to idling servers and which eventually realizes as pure waste of money and natural resources (2010). According to Srinivasan, pros of the cloud computing outweigh cons and therefore there is no reason to wait with the adoption (2014). Naturally there are cases where cloud computing is not an option so it should not be seen as a "silver bullet" either. In general, cloud computing offers heavy lifting on areas of business which are not creating additional value to customers. For an example, if you provide digital services to your customers, do your customers care which operating system your service backend resides on or do they care more about that the service is not crashing while your customers are using it? Hence you should focus on developing your service and leave IT resource infrastructure maintaining to your cloud service provider.

Internet of things (or IoT) as a term has also been a buzzword in the IT industry for around a decade. According to Uckelmann, Harrison and Michahelles the IoT term has been used and misused in many contexts due to indefinite definition (2011). In the cloud computing context, has been talked about "cloud envy" as a tendency to link "cloud" into things that might not have anything to do with the cloud computing (Salo, 2010). IoT can be equally seen as a "victim" of similar behavior and we could probably talk about "IoT envy" also. By the simplest definition, internet of things means a group of objects connected to the internet. Therefore, an IoT device can be almost anything that is connected to the internet. Technology companies are interested in getting their ordinary devices connected to the internet for accessing new IoT related revenue streams. However, faith of the IoT is bound to having internet access which still cannot be taken for granted everywhere (Mukhopadhyay and Suryadevara, 2014). Fortunately, recent years have shown that internet access level has been increasing (2014). As stated above, the definition leaves a lot room for interpretation and in the context of this thesis by IoT device is meant a device which has one to several sensors which measure environmental factors and then the device saves the measurement data and transfers it to cloud for utilization and more permanent storage.

The core idea of this thesis is to study how public cloud platforms and their existing services can be utilized to create backend services for IoT devices and how general-purpose cloud backend asset is defined and implemented for maximizing the usability in several IoT projects. For clarification, this thesis focuses on direct connection between IoT devices and the cloud backend. With cloud backend you can provide several additional features to your IoT system such as over-the-air updates, data processing and data storage. Cloud backend can also provide an endpoint and computing logic for IoT device fleet management needs. Naturally with worldwide and around the clock accessibility your IoT device can be almost anywhere on the planet Earth. Finally, you can save money because there are no maintenance costs and secondly

nowadays cloud services provide more than just an infrastructure. They enable you to focus on developing your application. So, to summarize, cloud backend can be used to lift your IoT system on a next level and in the forthcoming chapters you will see how.

1.1 Research problem

Commissioner of this thesis, SADE Innovations Oy, has recognized the common need related to IoT systems among its clients. Usually a new client has some kind of a device or an idea of device which measures something with its sensors. The client wishes to get the device connected to the internet for controllability and or to get device data saved in cloud for further analysis or use. Often a client wants to visualize the data gathered by the device or control the device with a simple user interface or using automated control logic in the cloud. Having recognized this common need among its clients SADE Innovations Oy wanted to develop a general software and hardware asset which can be used as basis for a new client specific IoT system. The software part of the asset includes a simple web application frontend, IoT device firmware and the cloud backend. The hardware part of asset includes basic expandable device with general sensors and its layout and schematic files.

The asset is supposed to speed up development work and be an advantage to commissioner in competitive biddings. It is also supposed to generate new revenue streams in form of sold software licenses. Also, it would reduce the total cost of the project to a client and the client gets faster to the proof-of-concept phase to further evaluate the business case or viability of the end product. The asset is not supposed to be ready made product or a platform for a client, but it should provide basic building blocks for development of a client specific IoT system with its custom-made features.

1.2 Research scope and goal

The goal of this thesis is to define and implement general-purpose cloud backend software asset which can be used as a basis for IoT system projects. The software asset should meet the requirements specified by the thesis commissioner. The scope of this thesis is limited to subjects related to cloud backend implementation including interfaces between the cloud backend and IoT devices and the cloud backend and the web application frontend. This thesis consists of requirement specification, technology background, definition and implementation of cloud backend, analysis of the implementation and conclusions.

1.3 Research methods

Methodology basis of this thesis is based on literal research and analysis of the cloud backend implemented as a result. Cloud backend definition is based on the set requirements by the commissioner and online documentations of cloud service provider.

2 Requirement specification and evaluation criteria

This chapter focuses on defining the requirement specification and also defining the evaluation criteria for to be implemented general-purpose cloud backend. The requirements are divided into two categories, functional and non-functional requirements. Functional requirements handle requirements which are about what system should be able to perform. Whereas non-functional requirements handle requirements which are about how system should perform. The requirement specification was provided by the commissioner and it has been guided by commissioners' clients and their requirements for IoT projects. The commissioner has filtered the requirements and created a set of requirements which contain very fundamental and general level requirements for the IoT cloud backend. This set of requirements guided definition and implementation work.

The general-purpose cloud backend implementation will be first evaluated against the functional and non-functional requirements listed below. After that the implementation will be critically evaluated for possible advantages and disadvantages it might raise. Finally, possible further development areas will be addressed.

2.1 Functional requirements

Functional requirements for the general-purpose cloud backend are listed below. Listed requirements are not in any specific order.

- Cloud backend must be able to control and configure IoT devices.
- Cloud backend must be able to receive sensor data from IoT devices.
- Cloud backend must be able to store received sensor data for later utilization.
- Cloud backend must provide interface for fetching sensor data.
- Cloud backend must provide interface for fetching IoT device state.
- Cloud backend must provide interface for controlling IoT devices.

Above listed requirements provide initial guidelines for defining and implementing the functional features of the system.

2.2 Non-functional requirements

Non-functional requirements for the general-purpose cloud backend are listed below. Like with functional requirements list, these are not either in any specific order.

- Cloud backend and provided interfaces must be secure.
- Cloud backend resources should scale according to usage.
- Cloud backend must be expandable, modular and customizable.

- Cloud backend pricing should scale according to usage.
- Cloud backend should provide data from devices to clients in real time.
- Cloud backend must be easy and fast to deploy.
- Cloud backend should be robust.
- Cloud backend should be always available.

Above listed requirements provide initial guidelines for defining and implementing non-functional features of the system.

2.3 Reasoning behind the requirements

Previously discussed requirements have been filtered and further generalized from the usual needs of the commissioner's clients. Usually a client has a device which is capable of measuring environment. The problem lies on how to access the data and utilize it. Devices could be located in many places with various connectivity options. Accessing the data from the device by going to particular place might work when number of devices is limited, and total distance is short. So, the solution is to send data to cloud for storage and utilization. Hence the requirement to store data to cloud. This requirement ensures that the data is accessible from anywhere via internet access.

However, the same problem of accessibility affects controllability of the device which is often needed by a client also. Therefore, the requirement to support controlling of devices from the cloud was defined. For better control, it is also necessary to know the current state of the device for proportionating the control measures. The requirement enables a client to control all its devices remotely.

To further extend the usability of the backend, interfaces for the possible client application had to be supported because otherwise the access to data would be only from the cloud. This thought was backed up by the commissioner's clients with their needs because usually they needed ability to control device or visualize the data from user friendly user interface such as mobile or web application. Therefore, the requirements for an interface to provide device data, state and controllability to user interfaces were defined. Also, separate interface improves decoupling between the user interface and the cloud services.

Reasoning behind the non-functional requirements might not be as evident as it is with the functional requirements but in general, they all aim for universal applicability. This can be seen most easily with the requirement stating that cloud backend must be expandable, modular and customizable. It is evident that universal applicability can be done to only some extent and therefore the asset should be modular, easily expandable and customizable to meet the client specific needs. Modularity and expandability also contribute to easiness to further develop the asset if and when it is deemed necessary.

Security is a corner stone in many systems. Security aims to limit access to the data or device controls for only those parties that are eligible to access them. In general, the data collected by devices might possess high value and therefore the data is wanted to keep as secret. Also, if

device controls are in wrong hands, the results might be costly or even cause accidents. Hence the security requirement of the cloud backend asset.

To operate on wider customer sector, the cloud backend asset should be viable solution for different sizes of clients and projects. Therefore, the asset's cloud resources and costs should scale according to usage. Especially, smaller clients might be reluctant to commit to relatively high costs when the overall usage level is quite low. On the other hand, bigger clients might have higher usage rates and therefore require more resources. In that kind of scenario costs might not be the most limiting factor. To take into account both scenarios, resource and cost scaling requirements are justified.

Robustness and availability of the cloud backend asset are key factors to the commissioner. If the system behaves unpredictably, the client becomes unsatisfied and asks for a refund. It is self-evident that refunds hurt the business case. Furthermore, from experience, bad reviews tend to spread further and faster than good reviews. The asset needs to be rock solid because possible clients depend on it and they trust the judgement of the commissioner on IoT related matters.

3 Technical backgrounds

Technical backgrounds chapter focuses on handling possible core technologies behind the general-purpose cloud backend. Technologies will be presented in a nutshell by discussing key points and features. Presented technologies were chosen by first evaluating briefly feasibility of the technology against the set requirements for general-purpose cloud backend. Some of the presented technologies have been already proven to work in other similar projects and therefore chosen to this project too. Final set of chosen technologies along with reasoning will be presented in the next chapter.

3.1 Application programming interfaces

According to Vijayakumar, application programming interfaces can be seen in two different types (2018). The first type is programming language construct which provides an exposed set of properties or operations inside the code for use in other layers of the code. The second type of API is a gateway between different systems which enables data and operations to flow through it. The one manifestation of the first type is software development kits (or SDK) and other software packages. (2018) GraphQL, WebSocket and REST are examples of the second type. This chapter focuses on APIs between the systems, more precisely on GraphQL, REST and WebSocket APIs.

3.1.1 GraphQL

GraphQL is a data query language originally developed internally by Facebook in 2012. It was later on publicly open sourced in 2015. (Foundation.graphql.org, 2020) GraphQL was first used by Facebook but nowadays it has many users including for example GitHub and Pinterest (Graphql.org, 2020a). GraphQL is an alternative to for example REST architectural style solution (Foundation.graphql.org, 2020).

GraphQL works on server-side runtime environment which executes queries on predefined and typed data (Graphql.org, 2020b). As stated in the GraphQL specification from June 2018, it is worth noticing that GraphQL is not a programming language. GraphQL will not either enforce to use any particular programming language. It bases on a few design principles. GraphQL offers hierarchical query structure which is optimal for clients to describe data needs from a GraphQL service. It is product centric which means that it has been developed frontend developers in mind for easing their work. GraphQL offers strong typing which means that query and data validity can be ensured. For example, if data item is defined to be type of string then client can expect to receive string value, otherwise query will fail. Or if the provided query does not match the defined GraphQL schema, query will also fail. GraphQL enables client specified queries which means that a client can specify which data items it wishes to receive and then GraphQL returns only those items in the set boundaries of GraphQL schema. The last principle is that GraphQL is introspective. Introspective means that the type of field (or data item) must be able to be queried also. (Spec.graphql.org, 2018)

GraphQL schema defines how and in what form the data can be fetched from the API. It also defines how the data must be supplied to the API. GraphQL supports three types of operations which will be presented next. Query operation is for fetching data. Mutation is for writing data. Subscription is the last operation and it works in response to source events and provides a way to implement real time communication. Operations return a selection set which is composed of fields. Fields represent information which can be requested as part of selection set. A field could be for example "name" and thus its type is most likely string. A field can also be more complex and contain its own selection set of other fields and this way one request can initiate series of nested requests. (Spec.graphql.org, 2018) In the figure 3.1 is presented example of simple GraphQL schema.

```
type Query {
    getPerson(id: ID): Person
}

type Person {
    id: ID
    firstName: String
    lastName: String
    age: Int
    occupation: String
    city: String
    country: String
}
```

Figure 3.1: Example of GraphQL schema defining simple query operation for fetching person's information.

3.1.2 REST

REST is architectural style originally introduced by Fielding (2000) in his doctoral dissertation "Architectural Styles and the Design of Network-based Software Architectures". The term REST comes from words Representational State Transfer and it is intended to be used with distributed hypermedia systems (2000). Systems implementing REST architectural style are called RESTful systems. Today, REST has become so popular that the acronym is used sometimes to describe systems which do not implement the REST style properly. Misuse is due to many people not understanding that REST is more than leveraging HTTP protocol and sending data blobs between the systems. REST is not actually bound to any protocol, but the used protocol must support uniform resource identifier scheme. (Doglio, 2015) However, in this chapter REST is assumed to utilize HTTP protocol.

REST architectural style bases on several constraints which enable style implementation. Implementation of REST begins from the so-called null style which means that there are no

constraints. The first constraint is about client and server and there should be clear separation of concerns between the client and the server. The second constraint states that communication between the client and the server should be stateless which means that request from the client should contain all the needed information to execute business logic on the server. The third constraint states that data of response for a request should be labeled as cacheable or non-cacheable to improve network efficiency with cached data responses. Next constraint requires that there is a uniform interface between system components which increases decoupling and independent development. Next constraint brings the layered system which ensures that components only know about the immediate layer which increases independency and provides a way to encapsulate entities. The last constraint is optional, and it enables code-on-demand which means that the client functionality can be extended with external downloadable code. The above presented constraints are simplified and for further information it is encouraged to read Fielding's doctoral dissertation. (Fielding, 2000)

Core concept of REST is virtual resources. A resource can be anything from a web page to an image or a text. Common to all resources is that every resource can be identified with a URI. (Fielding, 2000) Typically, resources can be created, read, updated and deleted. HTTP protocol provides verbs which can be used to reference the right type of action to be done to a resource. Generally used HTTP verbs are POST, GET, PUT and DELETE. POST is usually used to create a new resource out of the request provided representation. GET is used to read the resource. PUT is used to update resources. DELETE, as its name suggests, is used to delete the resource. (Doglio, 2015) Put simply, resources or precisely states are transferred with a request provided representations, hence the name Representational State Transfer (REST) sounds quite logical.

Table 3.1: Simplified examples of REST API requests over HTTP (Doglio, 2015)

HTTP verb	URI	Action
GET	/api/books	Gets list of available books
GET	/api/books/example-book	Gets the defined book
POST	/api/books	Creates a new book with request provided representation
DELETE	/api/books/example-book	Deletes the defined book
PUT	/api/books/example-book	Updates the defined book with request provided representation

3.1.3 WebSocket

WebSocket API works via WebSocket protocol to provide real time communication between the client and the server. WebSocket protocol complements the lacking features of real time communication in HTTP. Before WebSocket emerged, real time communication needs were tackled with HTTP polling, long polling or streaming which all have their own issues. Common issues among all of the HTTP based real time communication methods were inefficiency and latency. WebSocket is event driven and after a client has been connected to a server, the server can send messages to clients and clients can also send messages to the server. (Wang, Salim and Moskovits, 2013)

3.2 MQTT

MQTT is messaging protocol which is rather simple and lightweight. It is intended for devices which might have limitations in bandwidth, latency or connection reliability. Its design principles make it ideal for use in for example IoT or mobile applications. Dr Andy Stanford-Clark of IBM and Arlen Nipper of Arcom created MQTT in 1999. MQTT provides publish and subscribe communication methods where publish is for sending messages and subscribe for listening to messages in specified topics. (Mqtt.org, 2020) MQTT supports hierarchical topic structure which enables a client application to subscribe to only some of the messages or to publish message to only some of the subscribers.

3.3 Serverless

The term serverless refers to an architectural style to implement cloud-based backend applications using serverless computing. Serverless computing comprises cloud services which are entirely managed by the cloud service provider. Such cloud services let users to focus on using the service without worrying about scaling or adequacy of computing resources. It is worth noting that serverless does not mean that there are not any servers present in the underlying infrastructure of the service. It means merely that servers are managed by the cloud service provider and user does not have to take care of them. One embodiment of serverless computing is function as a service (FaaS) type cloud service which offers code execution environment in the cloud. For example, AWS Lambda offered by Amazon Web Services is such service. (Stigler, 2018)

Serverless services are typically paid by actual usage. This means that you do not have to commit on fixed periodic costs even though the service was not actually used which is the case with for example rented servers. Serverless services have many benefits including scaling and fast development but they also have limitations like becoming too coupled to specific cloud service provider or losing the control over infrastructure. However, benefits and limitations are not so black and white and must be weighed case by case. For further information it is encouraged to read Stigler's book from 2018 which presents serverless computing and related matters. (Stigler, 2018)

3.4 Serverless Framework

Serverless Framework is a software package and tool which enables to define cloud service backend infrastructure as code. It is a convenient tool for developing and deploying serverless applications (Serverless.com, 2020b). Serverless Framework enables storing cloud backend configurations in version control and then cloud backend can be developed by multiple developers at the same time. Anyone who needs a copy of cloud backend configuration can deploy own copy of it with single command. It supports several cloud service providers but for simplicity and to maintain scope, this thesis focuses only on Amazon Web Services.

Serverless Framework cloud backend configuration consists of several key sections. Those sections are service, functions, events and resources. A service section contains functions and resources sections. Functions section contains definitions of your service specific Lambda

functions and triggering event configurations. Resources section is for defining infrastructure like S3 buckets or DynamoDB tables from which functions are depended on. Framework functionality can be also extended by introducing optional plugins section which contain references to respective plugins. (Serverless.com, 2020a)

3.5 TypeScript

JavaScript has gained solid position among web application development. However, it lacks some features which would make it more ideal for large scale applications. TypeScript is developed to address some of these issues and ease development work. It provides capability to for example define interfaces and type annotations. Type annotations make it easier to know what kind of data a function returns or a variable holds even though the project might not be familiar at all. Thinking the other way around they help to limit what kind of data is accepted in function parameters or variables. That is a clear benefit when a project has several developers. (Microsoft, 2016)

Interfaces are used to define public properties which implementing class should have defined. An interface could have several implementations providing same kind of functionality but in different implementation specific way. For example, the same interface defining cloud operations could have been implemented in Amazon Web Services and Microsoft Azure specific ways. This way interface can be used to break coupling with actual implementations. TypeScript compiler's type checking verifies that the code is compliant with for example type and interface definitions which reduces bugs in the code. TypeScript is actually JavaScript in different syntactic form, and it can sometimes cause cumbersome situations if you are familiar with other strongly typed programming languages. After TypeScript is compiled it has turned into plain JavaScript and type annotations and other TypeScript specific syntax is lost. Nevertheless, before compilation TypeScript has ensured that the source code follows defined rules and principles set by the developer in the form of code. (Microsoft, 2016)

3.6 Amazon Web Services

SADE Innovations Oy has chosen Amazon Web Services (AWS) for its cloud service provider and therefore this thesis focuses only on AWS. According to estimates provided by Synergy Research Group, AWS had 33 % in worldwide market share of cloud infrastructure service providers in final quarter of 2019 which means it is the biggest player in the market (Richter, 2020). This chapter focuses on presenting short summary of essential features in each possible key cloud service and for further information it is encouraged to explore Amazon Web Services online documentation.

3.6.1 Amazon API Gateway

Amazon API Gateway provides HTTP and WebSocket based APIs for your cloud-based applications. API Gateway API acts as a gateway to your cloud services without client having to know anything about technologies or services in the background. For example, API can be coupled with Lambda functions to provide server-side logic for executing database queries. API

Gateway supports several authentication methods so that unauthenticated requests can do no harm. It also supports implementing REST APIs. (Amazon Web Services, 2020a)

3.6.2 Amazon Cognito

Amazon Cognito can be used to implement authentication, authorization and user management in cloud-based applications. It can be used to grant permissions to authenticated application users to call REST or GraphQL API for getting data from database for example. It is also capable of being hooked with third party authentication providers for example Google or Facebook. Amazon Cognito consists of User Pools and Identity Pools. User pool provides sign-in and related capabilities and it contains user data for your application. Identity pool is for providing access to other AWS cloud services. (Amazon Web Services, 2020b)

3.6.3 Amazon DynamoDB

Amazon DynamoDB provides NoSQL database tables. Its key features are rapid performance, high availability and durability and good scaling capabilities. DynamoDB tables can store large amount of data while still maintaining the capability to offer fast queries. It automatically replicates all data across several availability zones inside AWS region and distributes the data and traffic across several servers to achieve high performance and durability. A table contain zero to many items and a one item contain one to many attributes. DynamoDB tables are schemaless so new attributes can be added on-the-go. Each item represents one row in a table and each attribute represents one column in a table. (Amazon Web Services, 2020c)

Each item must be uniquely identified in a table. In DynamoDB context unique identification is handled with primary key. Primary key consists of only partition key (also known as hash key) or partition key and sort key (also known as range key) combination. Queries and writes are done by partition key or partition key and sort key combination. This means that to write or query items, you need to provide needed keys along your request. If primary key alone does not provide enough query options for your table, query options can be expanded by creating secondary indexes. Attribute values can be scalar or nested so you could save for example strings, numbers or JSON objects into attribute values. (Amazon Web Services, 2020c)

3.6.4 Amazon S3

Amazon S3 also known as Amazon Simple Storage Service is a service for file storage. S3 offers buckets which are simply containers for files. Files can be in any format and up to 5 TB in size. Buckets can also contain folders which can be used to organize files like you normally would with your local hard drive. Geographical location of bucket can be controlled by setting desired AWS region in favor of meeting regulatory needs or minimizing the latency. Stored data in bucket is replicated to several locations inside of chosen AWS region for ensuring high availability. (Amazon Web Services, 2020d)

S3 documentation refers to files as objects. An object contains naturally the file data but also metadata as name-value pairs. Metadata can be used to control for example caching. Each object

in a bucket is identified with a unique key. The key consists of names of the parent folders and object name for example “folder_name/object_name.jpg”. An object can be requested when the object key is known along with the bucket endpoint. Permissions to access data in S3 bucket can be granted even on object level. S3 buckets can be also used to host static web applications. S3 offers storage of objects in different storage classes which can be used in favor of saving costs if objects are not accessed very often. (Amazon Web Services, 2020d)

3.6.5 AWS AppSync

AWS AppSync provides GraphQL APIs for your cloud-based applications. Just like API Gateway API, AppSync GraphQL API also acts as a gateway to cloud services without client having to know about the background technologies. AWS AppSync GraphQL API consists of GraphQL proxy, data sources and resolvers. GraphQL proxy is responsible for running the GraphQL engine which handles requests and forwards those to logically right places. Resolvers connect GraphQL proxy to respective data sources. Resolvers are basically functions which convert GraphQL requests to respective service protocol for accessing the service on the background. These functions are called as a mapping templates in AppSync documentation. Resolvers are mapped to fields in GraphQL schema. Resolvers consist of request and response functions which can be used to process requests or responses further before accessing the data source or respectively before returning the response from the data source. Only some of AWS services connect directly with AppSync resolvers for example Lambda and DynamoDB. However, rest of services can be still connected via Lambda functions relatively easy. GraphQL API can have multiple data sources defined and one request could access several data sources on the background based on the request. (Amazon Web Services, 2020e)

3.6.6 AWS Identity and Access Management

AWS Identity and Access Management also known as IAM is service for defining access rights for example to roles, users and groups. It can be used to grant access for example to Lambda function which executes DynamoDB database writes. Otherwise database write would fail. IAM has essential role when using other services because by default services cannot communicate with each other in any way. AWS recommends granting least privileges possible in IAM best practices documentation. (Amazon Web Services, 2020f)

3.6.7 AWS IoT Core

AWS IoT Core provides way to securely connect IoT devices with AWS cloud. IoT devices use certificates to authenticate communication with IoT Core. Certificate is also saved in IoT Core where it can be disabled to deny IoT device from communicating with the cloud. (Amazon Web Services, 2020g)

IoT Core enables receiving messages from IoT devices via MQTT but also sending messages to them. Messages can contain for example measurement data from sensors like temperature and humidity measurements. It also enables controlling IoT devices remotely. IoT devices can be configured to send messages to specific MQTT topics which can be coupled for example

with Lambda functions. This coupling mechanism is provided by Rules Engine and it supports SQL like language for selecting relevant pieces of data from messages. Lambda functions can be used to for example process data further or saving data as it is to database. (Amazon Web Services, 2020g)

IoT Core also holds representation of device state of each configured IoT device. This device state is called Device shadow which is simply a JSON blob containing for example device settings. Device shadow service keeps device state in sync with actual IoT device state whenever IoT device is connected to cloud. However, if IoT device is disconnected at the time when device shadow is modified, device shadow will be synced next time when IoT device connects. So, device shadow enables controlling device state even though the device might be offline at the time. Device shadow service operates also via MQTT. (Amazon Web Services, 2020g)

3.6.8 AWS Lambda

AWS Lambda provides cloud runtime for functions. It supports multiple programming languages and runtimes including Node.js (JavaScript), Java and Python. (Amazon Web Services, 2020h)

Lambda functions are triggered to run in response to events. Such events can be for example periodic triggers, API calls or triggers from other services. Lambda functions are intended to be run rather short periods of time so they cannot be kept running forever. On need basis the same function can be run at the same time in many Lambda execution contexts which enables scaling up when there is high demand for computing logic. Lambda functions are stateless by nature which needs to be taken into account when implementing Lambda functions. Lambdas can be used as a “glue” between the other services which means that hypothetical service A can trigger a Lambda function to filter sensor data and then to save data into database provided by service B. Lambda functions can also trigger another Lambda function if needed. (Amazon Web Services, 2020h)

4 Definition and implementation

This chapter focuses on describing how the cloud backend definition was made and how the implementation work was done while taking into account set requirements by the commissioner.

4.1 Architecture

First, it is important to go through main principles which have been guiding the definition and implementation of cloud backend. The core principle was to use as many AWS solutions as possible to fulfill the needs specified in the requirements. This way there is less need to implement equivalent solutions by yourself. Implementing equivalent solutions would have required a lot more work and therefore it would have delayed this project greatly. Yet alone fulfilling for example security, resource scaling and availability requirements would have been quite challenging. Also, one requirement stated that cloud backend price should scale according to usage which makes usage of virtual and dedicated servers being out of the question because they would have introduced fixed costs.

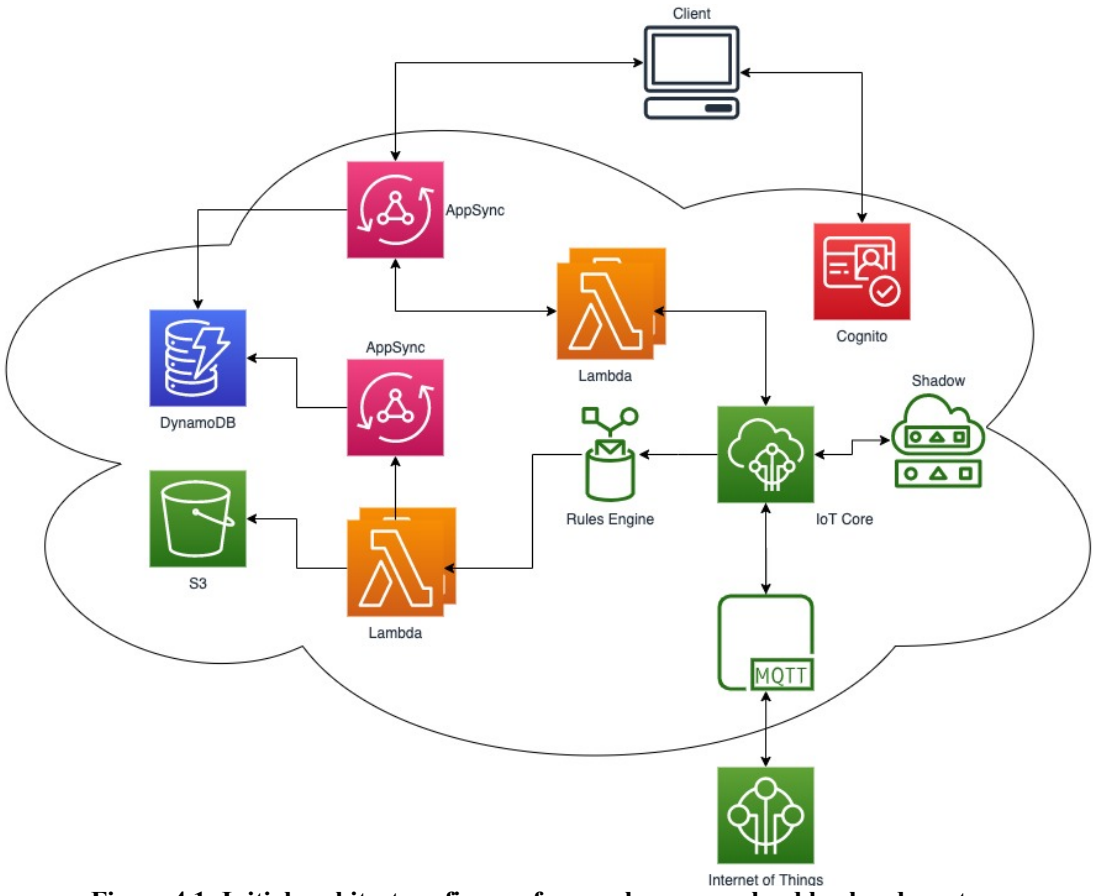

Figure 4.1: Initial architecture figure of general-purpose cloud backend asset.

In the figure 4.1 is presented initial architecture of general-purpose cloud backend. From the figure it can be seen that AWS AppSync GraphQL API is a gateway for a client application to which Cognito provides authentication mechanism. DynamoDB database stores the data for the

client to access via AppSync GraphQL API. S3 bucket acts as data archive and data source for future needs of data analytics. IoT Core provides gateway for IoT devices to connect to cloud. It also manages device configurations and certificates. Lambda functions are linkage and data processing units between the IoT Core, AppSync and S3. Forthcoming chapters will discuss more in detail about the architecture.

4.2 IoT device management with IoT Core

IoT Core provides device management, device connection endpoint and device status and message delivery. IoT devices communicate with the cloud via IoT Core provided MQTT endpoint. MQTT messages are used to initialize IoT device, deliver sensor measurement data and device shadow state. Next will be discussed IoT Core related configurations on general-purpose cloud backend.

The first-time a device connects to IoT Core with a valid device certificate, the device publishes a message to predefined handshake topic. The handshake topic is for triggering the provisioning of the device in the IoT Core and the topic is configurable on Serverless Framework template file. Provisioning creates a new IoT Thing which represents the actual IoT device in the IoT Core. After provisioning, the device can be seen in the IoT Core management console and its shadow state can be seen and modified. Device provisioning will be discussed further in chapter 4.5.

After the device is provisioned in the IoT Core, its state can be controlled via device shadow. Device shadow does not require any configuration to be working. It has predefined topic structure to which IoT Core user has no control over. Device shadow is a mere JSON document representation of device state. It consists of two states, desired and reported. Reported state is the last known state reported by the device whereas desired state is the next state which will be eventually the reported state. Desired state can be used to trigger changes to reported state. For example, a device can be controlled to turn on LED by setting respective Boolean value in desired state to true. The device receives this change in desired state and reports back with reported state having the respective value set true. In general-purpose cloud backend, device shadow is the main way to control device remotely. Device shadow ensures that the device state is synchronized with the device sooner or later even though the device would have been offline when the state was modified. However, devices could be controlled also via regular MQTT messages and it might be in some cases a valid option. Further discussion about device shadow is in chapter 4.5.

IoT device can report sensor measurement data periodically or intermittently. Whenever a device has new data to send to cloud, it publishes a new MQTT message to predefined sensor data topic. Sensor data messages are in JSON format and message payload contains the id of device, payload timestamp and one to several measurements with measurement name, value and measurement specific timestamp.

4.3 GraphQL API with AppSync

AWS AppSync provided GraphQL API is the gateway for the client application. API for a client application could have been implemented with Amazon API Gateway which provides REST and WebSocket APIs. However, the usage of API Gateway would have required more work especially to provide similar real time communication features which are already built-in in AppSync. For example, managing connected WebSocket clients would have required another DynamoDB database to store clients and several AWS Lambda functions to execute client management. Because of this, AppSync GraphQL API was chosen. All client originated requests to cloud go through GraphQL API which resolves requests and forward them to right data sources. According to requirement specification, device sensor data and state must be fetchable by the client and they should be also received in real time if a client is subscribed to changes in them. Also, device state must be controllable from the client. Due to requirement specification GraphQL API has also a role to play on device data database writes and shadow updates so GraphQL API is not only for the client to use. Following subchapters will go through the main features guided by the requirement specification and what requirements they set for the GraphQL API.

4.3.1 Sensor data storage mutation

First of all, device data must be saved into database for a client to query. This can be achieved by creating GraphQL mutation which will execute database write to DynamoDB on its resolver. This mutation is initiated by Lambda function which responds to new sensor data messages. One might ask though why write to the database is not done in a Lambda function. This is due to requirement of getting device data to a client in real time and it will be discussed further below. Mutation accepts sensor payload object as an argument. The object should contain device id, timestamp, timestamp id and the actual measurement data. The measurement data content can vary according to available measurements in the message payload. Timestamp id is a mere combination of timestamp and UUID and it will be explained later on chapter 4.6. The entire process flow of sensor data storage mutation can be seen in the figure 4.2.

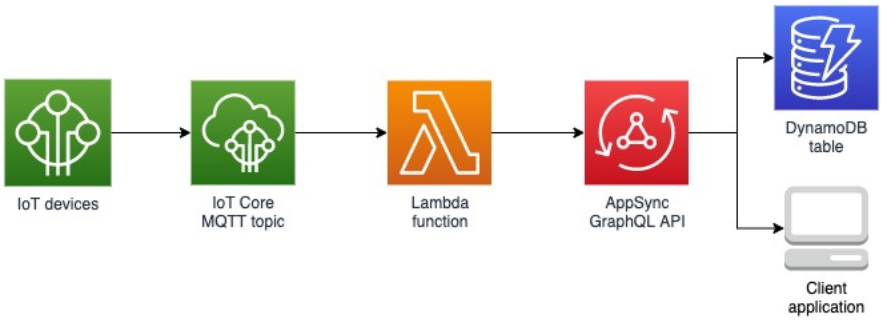

Figure 4.2: Process flow chart of sensor data mutation.

An IoT device first publishes a new sensor data message to MQTT topic which is then delivered to a Lambda function which calls the GraphQL API to write message to a database and notify subscribers. This chapter focuses on AppSync GraphQL API part and in the figure 4.3 can be

seen example of GraphQL mutation implementation to store sensor data. The implementation also includes definitions of input payload and response payload.

```
type Mutation {
    saveDeviceData(dataPayload: DataPayload): Data
}

input DataPayload {
    deviceId: String!
    timestampId: String!
    timestamp: String!
    temperature: Float
    humidity: Float
}

type Data {
    deviceId: String!
    timestampId: String!
    timestamp: String!
    temperature: Float
    humidity: Float
}
```

Figure 4.3: Example of sensor data storage GraphQL mutation implementation.

Next, for the mutation to work it is necessary to create a resolver for accessing the data source. Resolvers connect GraphQL operations to data sources with request and response mapping templates. Request mapping template configures how the operation should be executed against a data source whereas response mapping template configures what should be done with the response from the data source. Even though mapping templates can contain quite advanced logic, rather simple configuration was sufficient enough in this case.

```
{
    "version" : "2017-02-28",
    "operation": "PutItem",
    "key": {
        "deviceId": { "S" : "${context.arguments.payload.deviceId}"},
        "timestampId": {"S": "${context.arguments.payload.timestampId}"}
    },
    "attributeValues": $util.dynamodb.toMapValuesJson($context.arguments.payload)
}
```

Figure 4.4: Example of request mapping template writing item to DynamoDB database.

From the figure 4.4, it can be seen that the resolver executes a put item operation to DynamoDB with device id, timestamp id and rest of the sensor data as a payload. Payload items are provided by the mutation payload argument.

```
1    $util.toJson($context.result)
```

Figure 4.5: Example of response mapping template converting a response to JSON blob.

When the put item operation is ready, the response is processed by response mapping template. In this case, the response is just converted to a JSON blob as it can be seen on figure 4.5.

4.3.2 Sensor data query

The general-purpose cloud backend should also provide API for a client to make queries in database. Query can be implemented by defining GraphQL query and respective resolver for making DynamoDB queries. The GraphQL query is defined so that DynamoDB query can be made using device id and timestamp range. Figure 4.6 describes a process flow of sensor data query which is quite simple compared to previously presented mutation.

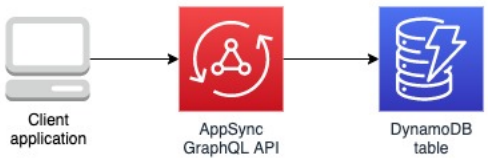

Figure 4.6: Process flow chart of sensor data query.

First client calls the GraphQL API which then executes the actual database query and returns with matching database items. Figure 4.7 describes an example of GraphQL sensor data query implementation.

```
1  type Query {
2      getDeviceData(deviceId: String!, startTimestamp: String!, endTimestamp: String!): DataResponse
3  }
4
5  type DataResponse {
6      items: [Data]
7      nextToken: String
8  }
```

Figure 4.7: Example of sensor data GraphQL query implementation.

Next, a resolver was created for the query. Resolver mapping templates are quite simple throughout general-purpose cloud backend and therefore they differ relatively little from resolver mapping templates accessing same kind of data sources. The core changes are due to change of operation specific request structure.

```
{
    "version": "2017-02-28",
    "operation": "Query",
    "query": {
        "expression": "deviceId = :deviceId AND timestampId BETWEEN :startTimestamp AND :endTimestamp",
        "expressionValues": {
            ":deviceId": { "S" : "${context.arguments.deviceId}" },
            ":startTimestamp": { "S" : "${context.arguments.startTimestamp}" },
            ":endTimestamp": { "S" : "${context.arguments.endTimestamp}" }
        }
    },
    "select": "ALL_ATTRIBUTES"
}
```

Figure 4.8: Example of request mapping template querying items from DynamoDB database.

In the figure 4.8 it can be seen that query operation is executed against database with device id and timestamp range arguments. The response mapping template is similar to previously presented save sensor data response mapping template in the figure 4.5.

4.3.3 Sensor data subscription

GraphQL subscriptions make it possible for a client to get up-to-date sensor data whenever a new sensor data payload is sent to the cloud backend by the device. Subscriptions are bound to mutations and in this particular case subscription was bound to mutation writing sensor data to DynamoDB. Subscription was defined so that device id is given as an argument which make it possible to subscribe only to selected devices instead of receiving sensor data from all of devices. Process flow of sensor data subscription follows the previously presented figure 4.2.

```
type Subscription {
    onDataUpdates: Data
    @aws_subscribe(mutations: ["saveDeviceData"])
}
```

Figure 4.9: Example of sensor data GraphQL subscription implementation.

Figure 4.9 presents rather simple implementation of GraphQL subscription which couples with the previously implemented “saveDeviceData” mutation. Every time the mutation is called all subscribers will get notified with the payload of mutation. Subscriptions differ from queries and mutations because they do not require resolvers because they are only coupled with mutations.

4.3.4 Device state query

Requirement specification also stated that device state should be fetchable via API. This can be achieved by defining yet again GraphQL query and respective resolver. However, this time resolver is bound to a Lambda function because AppSync does not support IoT Core as a data source. The process flow of device state query can be seen in the figure 4.10.

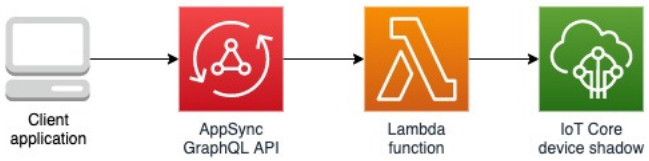

Figure 4.10: Process flow of device state query.

First a client calls the GraphQL API which then triggers the Lambda function to fetch device shadow. Next in the figure 4.11 can be seen example of GraphQL query implementation to fetch device shadow.

```
type Query {
    getDeviceShadow(deviceId: String!): ShadowResponse
}

type ShadowResponse {
    desired: Shadow
    reported: Shadow
    timestamp: Int
}

type Shadow {
    deviceId: String
    displayName: String
    ledEnabled: Boolean
}
```

Figure 4.11: Example of device state GraphQL query implementation.

The query requires device id as an argument and returns response which contains desired and reported states of the device and timestamp. Next in the figure 4.12 can be seen example of resolver's request mapping template implementation triggering a Lambda function with GraphQL provided arguments.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getDeviceShadow",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

Figure 4.12: Example of request mapping template invoking a Lambda function to fetch device state.

The resolver acts merely as middleman and delivers device id to the Lambda function. Then the Lambda function fetches the device shadow from the IoT Core according to the matching device id. The response mapping template is once again similar to previously presented one in the figure 4.5.

4.3.5 Device state mutation

For enabling device state modification via API, one more GraphQL mutation must be defined along with a respective resolver. Data source for this mutation is Lambda function due to reasons stated above. Lambda function modifies the desired state of device in device shadow service according to the provided state payload in a request. In the figure 4.13 can be seen process flow chart of device state mutation.

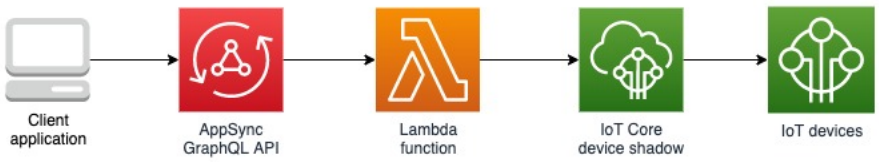

Figure 4.13: Process flow chart of device state mutation.

First a client calls the GraphQL API which triggers a Lambda function to execute device shadow change in the IoT Core. IoT Core device shadow synchronizes changed device state to an IoT device which reports back with changed state if the change is approved by the device. Next in the figure 4.14 can be seen the GraphQL device state mutation implementation.

```
type Mutation {
    clientChangeShadow(deviceId: String!, payload: ShadowPayload!): Shadow
}

input ShadowPayload {
    deviceId: String
    displayName: String
    ledEnabled: Boolean
}
```

Figure 4.14: Example of device state GraphQL mutation implementation.

The mutation requires device id and payload object as arguments. Payload object contains the key-value pairs of changed state items. Device id and payload object are needed in the data source Lambda function which executes the change of device shadow in the IoT Core. Next in the figure 4.15 can be seen request mapping template which is quite similar to previously presented request mapping template in the figure 4.12.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "clientChangeShadow",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

Figure 4.15: Example of request mapping template invoking a Lambda function to request device shadow change.

The only change in these request mapping templates is in the “field” value which is due to changed mutation. The resolver delivers arguments from the GraphQL mutation call to the Lambda function executing the change in device shadow. The response of the Lambda function is processed in already familiar response mapping template presented in the figure 4.5.

4.3.6 Device state subscription

Final feature setting requirements to GraphQL API is to have possibility to client follow device state in real time. This feature is a bit cumbersome because device state updates happen inside IoT Core without possibility to hook GraphQL API in between devices and the device shadow service. Also, GraphQL subscriptions require a mutation for to be bound with. However, subscription cannot be bound to previously presented mutation for a client originated device state change because it would require IoT Core rules engine to trigger Lambda function in response to device shadow change. It would lead to Lambda function call the mutation to execute device shadow change yet again. This kind of approach would create a loop and add

costs for nothing in return. To circumvent this dilemma, another mutation is defined for IoT Core to report when device shadow changes. The mutation requires device shadow payload as an argument which is passed from a Lambda function. However, this mutation does not require any data source for the resolver to be working with and it simply returns the request payload as response to the caller. Finally, a new subscription can be defined and bound to newly defined mutation. This way IoT Core triggers a Lambda function when a device shadow change and the Lambda function calls GraphQL mutation with device shadow as an argument. Eventually, the client receives payload via the subscription. In the figure 4.16 can be seen process flow chart of device state subscription.

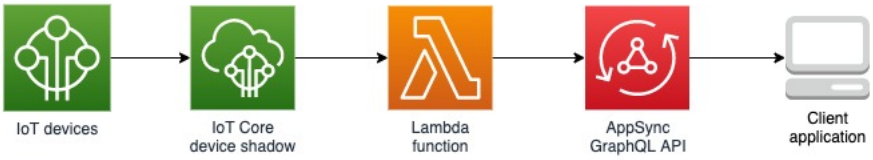

Figure 4.16: Process flow chart of device state subscription.

First a device reports a new state to the IoT Core. Then IoT Core triggers a Lambda function to call the GraphQL API mutation. The mutation call triggers a notification to all subscribed clients. Next in the figure 4.17 can be seen the GraphQL implementation containing the needed additions.

```
type Mutation {
    deviceShadowUpdated(payload: ShadowUpdatedPayload): ShadowResponse
}

type Subscription {
    onShadowUpdated: ShadowResponse
    @aws_subscribe(mutations: ["deviceShadowUpdated"])
}

input ShadowUpdatedPayload {
    desired: Shadow
    reported: Shadow
    timestamp: Int
}
```

Figure 4.17: Example of device state GraphQL subscription implementation.

The mutation in the figure 4.17 is exceptional because it does not do anything like rest of previously presented mutations. The mutation is solely for triggering the “onShadowUpdated” subscription. In the figure 4.18 is shown the respective request mapping template.

```
{
  "version": "2017-02-28",
  "payload": $utils.toJson($context.arguments.payload)
}
```

Figure 4.18: Example of request mapping template of device shadow updated mutation.

The request mapping template just delivers payload arguments from the GraphQL mutation to the response mapping template. This approach is necessary to get the subscription working.

4.4 User and access management with Cognito

Amazon Cognito has an essential role securing the GraphQL API. AWS AppSync requires that GraphQL APIs are secured by at least one of four supported authentication methods. For this project AWS IAM was chosen as authentication method. This method requires that identity which is used to call GraphQL API has an IAM policy which explicitly allows request to be made. One way to provide such identity is to configure Cognito user pool and identity pool. User pool enables managing and controlling access to your application and backend resources. It provides mechanism to authenticate users against existing users in the user pool.

User pool was configured to ask first name, last name and email address at user creation. User pool is tied to identity pool which assigns identities and identity bound access rights to user pool users. Identity pool enables to define and assign roles for authenticated and unauthenticated users of the application. These two IAM roles can be tied to different IAM policies which enables defining different kind of access rights for authenticated and unauthenticated users. However, in this project unauthenticated users were left with no access rights at all. The authentication flow is presented in the figure 4.19.

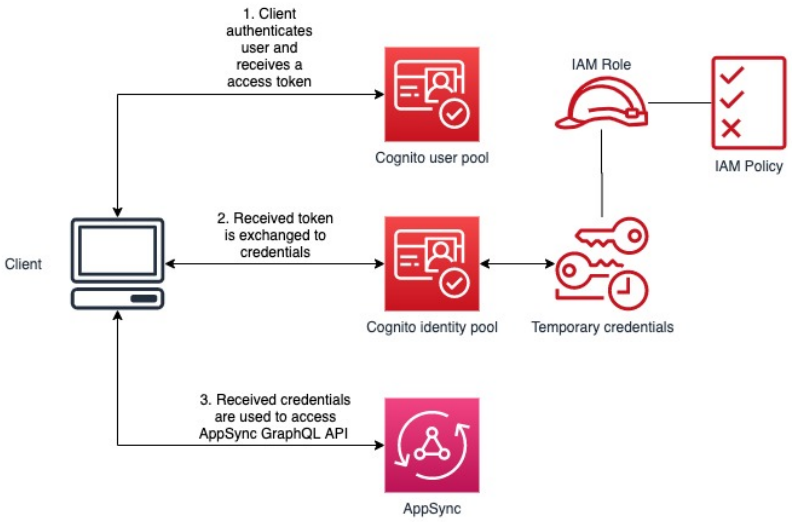

Figure 4.19: Authentication flow to access GraphQL API. The figure is based on Amazon Cognito developer guide. (Amazon Web Services 2020b)

User pool authenticated users are eligible to use authenticated IAM role according to trust policy defined in the IAM role. Trust policy can be used to set different conditions for role users. If the conditions are met, user is eligible to use IAM role and assigned policies. In this case trust policy was configured to check that identity is from the right identity pool and user is authenticated. IAM role for authenticated users was configured to allow only to call GraphQL API and its fields. It could be further restricted by limiting access right to some operations or fields only, but it was not necessary at the time.

4.5 Data handling with Lambda functions

AWS Lambda functions have an essential role to play in this cloud backend asset. They act as a "glue" between the services and convey data between them. They also provide additional computing logic and data processing. There are six use cases in the system which are implemented with Lambda functions: saving sensor data to database, archiving sensor data, getting device state, modifying device state, device state update notification and device provisioning. Lambda functions were configured to have NodeJS runtimes, 512 MB memory and timeout in six seconds. All Lambda functions of this project were implemented in TypeScript programming language.

TypeScript is compiled to JavaScript automatically before deployment because AWS Lambda does not support TypeScript natively. However, TypeScript was chosen mostly for its type system and it can be also used to develop the frontend side. Common programming language minimizes amount of used programming languages in the whole project and this way makes the development work easier. When Lambda functions access other services programmatically, it is necessary to remember grant all the needed permissions for the Lambda function to work properly. From experience, when starting to develop applications using Lambda functions, it is common that Lambda function does not work correctly because of lacking permissions.

4.5.1 Sensor data handling for storage to DynamoDB

IoT devices report sensor data over MQTT broker provided by AWS IoT Core. Sensor data is sent to a MQTT topic which is configured to trigger a Lambda function. So, every time when a device publishes message to the topic the Lambda function will be triggered with the message payload. Example of supported message payload can be seen in the figure 4.20. Message payload object contains device id, timestamp and values array which contains actual measurements. Values array can be easily read through and picked measurement names and values by using natively supported JavaScript array and object functions. After all relevant things have been picked from payload, GraphQL request can be generated and executed. Lambda will call GraphQL mutation which saves provided sensor payload to the DynamoDB database. One of the awesome features of DynamoDB is its schemaless nature which allows to add or remove measurements on the go without changing its schema.

Figure 4.20: Example of sensor data payload sent by an IoT device.

As stated above in the chapter 4.3, Lambda function must call GraphQL API mutation instead of writing to DynamoDB directly because a GraphQL subscription is tied to a GraphQL mutation. The subscription is required due to requirement specification for real time data updates and it ensures that all clients which are subscribed to data updates will receive them. It is notable that a Lambda function must have a permission to call GraphQL API. The permission is granted in an IAM role which is assigned to the Lambda function. In the figure 4.2 was presented the process flow chart of sensor data storage to DynamoDB.

4.5.2 Sensor data handling for storage to S3

In addition to saving data to database, a more permanent storage solution was also implemented. Another Lambda function was created to write JSON files containing sensor data payload content to S3 bucket. The Lambda function was connected to the same MQTT topic as previously presented Lambda function, so it was triggered at the same time when a new sensor data payload was published. An example of sensor data payload can be seen on figure 4.20. The Lambda function picks device id, sensor payload timestamp and all measurement values from the sensor data payload and creates a JSON document from them. The JSON document will be then saved in S3 bucket. At the time of implementation there was no reasonable data analysis use case so S3 bucket's role was merely to act as a data archive. The Lambda function must have a permission to create a new object in the S3 bucket. In the figure 4.21 is presented process flow chart of archiving device data.

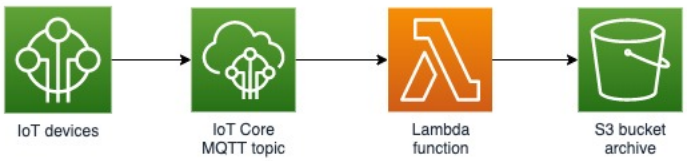

Figure 4.21: Process flow chart of archiving device sensor data to S3 bucket.

First a device publishes a new sensor data message to IoT Core MQTT topic. Then IoT Core triggers a Lambda function to handle the message and save it as JSON file to the S3 bucket.

4.5.3 Device state query handling

To support client's requirement to provide device state via GraphQL API, a Lambda function was created. This Lambda function acts as a data source for the GraphQL API. When a client makes a GraphQL request to fetch device state by provided device id a resolver triggers a Lambda function and passes device id along triggering event payload. Lambda function will then initiate request to AWS IoT Core via AWS JavaScript SDK provided functions to get device shadow by the provided device id. IoT Core then responds with the device shadow document. The shadow document contains desired and reported states of the device and metadata. From metadata the Lambda function checks that which shadow attribute was updated most recently and picks its timestamp to return to the client. Finally, a permission to get device shadow state from IoT Core had to be created.

4.5.4 Device state update handling

The requirement for getting device state updates in real time to a client require yet again a Lambda function to couple IoT Core and AppSync. Lambda function is coupled to IoT Core MQTT topic which provides device shadow updates every time when a shadow document of a device change. Coupling ensures that a Lambda function is triggered on every device shadow change. The Lambda function will first generate a GraphQL request to call a mutation based on the supplied device shadow document. The GraphQL request will be then submitted to the GraphQL API which resolves request to the right mutation and all subscribers will get notification. To allow Lambda function to call GraphQL API a permission to do so must be granted in the assigned IAM role.

4.5.5 Device state modification handling

According to requirement specification, clients should be able to control devices remotely. This can be achieved by coupling a GraphQL mutation to a Lambda function. As stated above, mutation accepts device id and shadow payload as arguments and passes those to a Lambda function. Lambda function will then initiate device shadow update by using AWS SDK provided methods. The Lambda function must have IAM role assigned which grants permission to update shadow. Shadow update is made according to the provided device id and shadow

payload. Device shadow change will be made to desired state only so a device can eventually accept the change and report back with reported state changed.

4.5.6 Device provisioning handling

Device provisioning is not specified in the requirements, but it makes adding new IoT devices to IoT Core easier. IoT devices cannot be controlled from management console until the device is provisioned properly. Provisioning means that a virtual resource called a Thing is created. Thing represents the actual IoT device in the cloud. To automate provisioning, a Lambda function was created. The Lambda function was coupled with predefined "handshake" MQTT topic. When IoT device first time connects to cloud it publishes MQTT message to handshake topic. Handshake message contains only device id which is used to create a new thing. The Lambda function uses AWS SDK provided methods to get access to IoT Core and create a Thing. Also, the Lambda function must have a permission to create a Thing specified in its assigned IAM role. In the figure 4.22 is presented process flow chart of device provisioning.

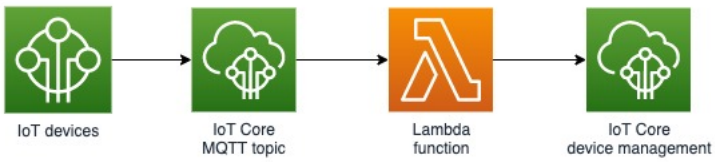

Figure 4.22: Process flow chart of device provisioning handling.

First a device publishes a message to IoT Core MQTT topic which is then handled by the Lambda function which creates a new Thing to the IoT Core.

4.6 Data storage with DynamoDB and S3

Data storage requirements were tackled by thinking use cases and what AWS solution would suit best for the use case in question. Also, the requirements had to be kept in mind. In general, there were two use cases which required data storage in the cloud. The first was storage of sensor data for queries made by the client and the second was sensor data storage for data analysis or archiving. Data analysis was not specified at the time, so the evaluation had to be made on general level. It was evident that the best option might be to have different solutions to tackle use cases which were quite different by their nature. From a client point of view data queries should be relatively fast so that excessive loading times could be avoided while using the client application. On the other hand, the client application might not need huge amount of data at once. Whereas data analysis might handle huge amount of data and the speed might not be so at the essence. The sensor data being saved to storage solutions is time series data by its nature.

4.6.1 DynamoDB

DynamoDB is a primary storage solution in the general-purpose cloud backend asset. However, the most suitable database solution would have been time series database because of the nature of sensor data. Typically, a time series database has features which makes storing and querying time series data easier and more efficient. DynamoDB is not a time series database nor it is especially meant to be used to store time series data. Nevertheless, DynamoDB can be used to store time series data and it has been proven to work in several similar projects earlier. Also, at the time of implementation AWS did not have a time series database service available and the requirements stating that resources and costs should scale according to usage also limited available SQL database options. Figuratively, jumping to the unknown with a new less familiar database service would require a lot more testing. Eventually the compromise had to be made and DynamoDB was chosen to be the main database solution in general-purpose cloud backend.

The first step to do when defining a DynamoDB database table, is to identify how stored data needs to be queried from the table. If there are several different queries, the most common query should dominate the table definition. In this particular case only one query was identified, and it was combination of device id and time range. Therefore, device id was defined as primary key of the table. Sort key enables to query by a time range, but one addition had to be made because of device capability to provide timestamps only in second resolution. Timestamp id was created to ensure that a data is not overwritten even if device id sent two payloads during one second which was quite unlikely in practice but possible in theory. Timestamp id is combination string of timestamp and UUID and it was defined as a sort key of the table. The table was created with automatic read and write capacity scaling enabled. The automatic scaling ensures that you only pay for storage when there are no requests made. Also, if there are unpredictable number of requests, the table can scale up automatically to serve all of them properly.

4.6.2 S3

S3 can store loads of data and it can be used as a data source for AWS data analysis tools. Costs follow storage amount and access requests and costs can be further optimized by changing storage levels for less frequently accessed files. However, in this use case default storage level was used for all stored files because there was no need to optimize costs. A S3 bucket was created to provide archive location for sensor data payloads and bucket name was provided to a Lambda function as an environment variable. S3 bucket creation required no additional configurations and the Lambda function can access it by using AWS SDK provided methods when permissions are set correctly in the Lambda function’s IAM role.

4.7 Serverless Framework template

The general-purpose cloud backend was defined and configured in a Serverless Framework project files. Serverless Framework uses YML files where all resources are defined and configured. Decision to define all resources in a single Serverless Framework project was made due to simplifying and speeding up the deployment process. The project was created by using aws-nodejs-typescript template provided by Serverless Framework. The template ensures that

all TypeScript related libraries will be installed automatically, and you do not have to spend time on finding right libraries. This project also required a third-party provided AppSync plugin library because Serverless Framework did not natively support defining AppSync GraphQL API resources at the time.

To ease development, definitions were divided into logical entities which all had their own files. Divide into several files helps to find the right places in the project and also single file does not grow unreasonably large. For example, all IAM roles assigned to Lambda functions were in one file. All separate YML files were then imported in the main Serverless YML file which is the file that the framework reads by default.

4.8 IAM roles and permissions

IAM roles and permissions are essential part of any cloud service-based backend deployment in the AWS. They enable services to communicate with each other within the bounds set by assigned permissions and act as a corner stone of the security. Therefore, IAM roles and permissions must be defined before the cloud service-based system can work properly. This thesis does not handle IAM roles and permissions very thoroughly because they do not actually contribute to general-purpose cloud backend asset in a way that would make it even more general-purpose. However, this thesis does not either mean to neglect the fact that they play essential role in the system. It is highly encouraged to explore IAM documentation on AWS website for further information.

5 Analysis of the implementation

This chapter focuses on analyzing the implementation from three different perspectives. The first analysis is made from the perspective of how the implementation met the set requirements. The second analysis focuses on benefits of the implementation and the last analysis focuses on future improvement areas.

5.1 Meeting the requirements

Analysis of meeting the requirements is divided in two categories which were already presented in chapter 2 handling requirement specification. Analysis will be conducted requirement by requirement.

5.1.1 Functional requirements

- Cloud backend must be able to control and configure IoT devices.

  IoT Core provided device shadow service fulfills all needed requirements for controlling and configuring IoT devices remotely. Furthermore, it ensures that control actions and configurations are delivered to IoT device eventually even if the device was offline at the time. This feature makes it possible to for example optimize device current consumption by keeping it offline without worrying whether the device is online or not when the device is being configured or controlled. However, the device shadow service does not make the device stupid in the sense of not being able to control its own state which enables device to have additional logic related to state.

- Cloud backend must be able to receive sensor data from IoT devices and store it for later utilization.

  The cloud backend receives sensor data via IoT Core provided MQTT broker. A device publishes a new sensor data as a MQTT message to predefined MQTT topic. In the cloud backend Lambda functions are coupled with MQTT topic and they get triggered whenever a new message is published. Lambda functions process all sensor data messages and take care of archiving the data to S3 bucket and trigger a GraphQL mutation in the GraphQL API to handle the storage to DynamoDB. Lambda functions enable the system to scale automatically by processing messages in parallel when messages are sent to cloud at higher message sending rates.

- Cloud backend must provide interface for fetching sensor data and device state and controlling devices.

  The cloud backend provides a GraphQL API which has query fields to query DynamoDB and fetch device state from the IoT Core and a mutation field to control devices in the IoT Core. The GraphQL queries make it possible to request only some of the available values if needed. Furthermore, the GraphQL API enforces loose coupling between the service and the client which enables for example changing the database solution without breaking the client implementation if needed. Due to strong typing in the GraphQL API, both the client and the backend can trust to receive the data in specified format.

To summarize it can be stated that all functional requirements were met without limitations.

5.1.2 Non-functional requirements

- Cloud backend and provided interfaces must be secure.

  The cloud backend provides secure GraphQL API via AWS AppSync which requires signed requests over HTTPS protocol. Signed requests can be made when a client application user has authenticated in the application. The authenticated user has access to call only those GraphQL API fields which are needed and nothing more. The GraphQL API can be called without a valid signature but such requests will be rejected.

  Generally, the security on AWS bases on IAM roles and permissions which support fine grained access control. All services which are communicating with each other require permissions to do so. In the cloud backend the security model has followed the least permissions needed strategy which ensures that the permissions are rather fine grained on what a particular service can do with the other services.

- Cloud backend resources and pricing should scale according to usage.

  The cloud backend has been implemented using such AWS cloud services which can scale according to usage. These services are also managed by the cloud service provider which means that cloud service provider has the responsibility to ensure scaling of resources. One good example of resource scaling can be seen with Lambda functions. There can be one to several Lambda functions processing in parallel incoming data based on load of the system. AWS Lambda adjusts automatically to increasing load by adding new parallel execution environments. Respectively when load decreases, number of parallel execution environments goes down.

  The pricing of those services is based on parameters which describe for example throughput or resource usage of the service. Therefore, overall pricing follows usage of

the whole system. For example, Lambda function pricing bases on memory usage over time period which the function was run. Whereas AppSync pricing bases on amount of made requests. Further and up-to-date information about pricing can be seen on Amazon Web Services website.

- Cloud backend must be expandable, modular and customizable.

  The cloud backend has been implemented by keeping expandability, modularity and customizability in mind. From customizability point of view, for example, all client specific definitions of the backend are centralized in one folder. Therefore, to deploy a new deployment all configurations can be made to minimal number of files. However, currently supported customizability is limited to those areas which are commonly customized in projects. Rest of customization has to be made by expanding the cloud backend asset with client specific features.

  The whole asset is quite modular due to all Lambda functions being implemented in a way that embraces clear responsibilities. This means that a Lambda function is responsible for doing only one thing instead of using only one Lambda function to do all required processing. This ensures that breaking one Lambda function does not cripple the whole backend.

  Expandability of the system is on quite good level. For example, IoT Core's MQTT broker enables to couple additional services to predefined or totally new MQTT topics whenever needed. Also new operations and fields can be added to GraphQL schema for future needs and use cases. However, with GraphQL you have to pay attention to not change anything that would cause breaking change in backwards compatibility. There is also a limitation in maximum number of defined resources in the Serverless Framework project which limits the expandability. However, the limitation can be circumvented easily by splitting the service down to several microservices which would lead to several Serverless Framework projects.

- Cloud backend should provide data from devices to clients in real time.

  The cloud backend implementation processes device state and sensor data in close to real time. Lambda functions are coupled with MQTT topics to which devices report changes in device state and sensor data messages. Lambda functions convey the messages to the GraphQL API which offers subscriptions to which clients can subscribe to and receive new data whenever it is available. Furthermore, less than one second end-to-end latencies can be achieved by selecting cloud backend region which is geographically close to an IoT device and user of web application. End-to-end latency means time which is taken from the moment when the data is sent from a device to the moment when the data can be seen in the user interface. It is notable that end-to-end latency contains also the time which is taken to process the data in the cloud.

- Cloud backend must be easy and fast to deploy.

  The current implementation of cloud backend can be deployed fast and easily thanks to Serverless Framework in use. When all initial configurations are already made, the

deployment can be made by executing single command. The deployment itself takes some time due to several cloud resources but it is a lot faster compared to manually configuring all resources. The template file is also parametrized to enable for example different kind of deployments for different development stages.

- Cloud backend should be robust.

  Cloud backend robustness is mainly due to robustness of AWS infrastructure. When using managed cloud services, it is typical that robustness of the infrastructure is taken care of by cloud service provider. However, a cloud service user can affect robustness in implementations side, especially in Lambda functions. In general-purpose cloud backend implementations were kept as rather simple with clear responsibility areas. For example, a Lambda function triggering storage of sensor data was responsible for calling the GraphQL API mutation and nothing more. This ensures that if a Lambda function fails in processing payload it only affects to the same Lambda function and not rest of the system.

- Cloud backend should be always available.

  Cloud backend availability is mostly due to global availability of AWS infrastructure. When using managed cloud services, it is typical that availability of underlying infrastructure is ensured by the cloud service provider. Generally, AWS availability bases on multiple geographical regions with multiple availability zones. Regions are separated from each other and they have availability zones which are responsible for maintaining the availability inside a region. The basic idea is that if a datacenter is down inside the availability zone it does not cripple the whole system inside the region because the other datacenters in other availability zones will act as a backup. (Amazon Web Services, Inc 2020)

To summarize, it can be stated that non-functional requirements were met in good extent with rather small limitations. However, it is evident that all non-functional requirements were not met due to a way the cloud backend implementation was made but the way cloud service provider has implemented things. Nevertheless, both parts contributed to meeting the non-functional requirements. The chosen cloud services also demonstrate how easily these non-functional requirements can be achieved just by selecting right cloud services and cloud architecture without needing to implement something by yourself. Before the arise of serverless cloud services and serverless architecture style, implementing similar backend services with similar robustness level and automatic scaling capability would have needed lot of efforts. Whereas nowadays cloud service providers offer those features out-of-the-box bundled in the service. This is yet another example of how cloud service providers have evolved and aimed to do heavy lifting for their clients so that clients could focus on using the service.

5.2 Benefits of the implementation

At the time when the definition and implementation work was in the beginning, the commissioner had initial assumptions on what kind of benefits the general-purpose cloud

backend would provide. Those initial assumptions were the reasons for beginning the project in first place. Now that the initial implementation is ready the realized benefits can be evaluated. The general-purpose cloud backend asset provides at least following benefits to the commissioner of this thesis.

1. It is easy to deploy to a new AWS account when a new project begins.
2. It is easy and fast to configure with client specific needs for a new client.
3. It saves time because general features are already implemented into the asset.
4. It helps to compete in bidding competitions because the workload is reduced.
5. It speaks for the commissioner that its employees are capable of working with IoT projects.
6. It provides license revenues.

Naturally the commissioner is not the only one getting benefits from the implementation. Thinking from a client's point of view the key benefits can be seen listed below.

1. The project becomes ready faster.
2. The project requires less time and money.
3. Efforts can be focused on client's needs.
4. The software asset is tested and proven to work in several other projects which means that the overall quality is on good level.

However, the true benefits from the client's perspective would require separate study to assess them properly.

5.3 Future improvement areas

From experience, while defining and implementing software or any other type of project there are many things that might limit the definition and implementation work. Usually the limiting factor might be the time, money or working human resources allocated to the project. Limiting factor might also be level of expertise of human resources or something else related to anything. The fact is there is always some limitations. Limiting factors will lead to compromises which might not be the best solutions, but they will do just fine until there are fewer limiting factors and the implementation can be improved. The key point is that there is always room for future improvements and so is in this project.

Following future improvement areas were identified during working with this thesis and the general-purpose cloud backend project.

- Splitting monolithic service to several microservices
- Implementing unit tests

- Improving device certificate provisioning
- Switching from current database solution to time series database solution

Splitting monolithic service to several microservices is mainly an architectural improvement but it would make it possible to deploy general-purpose cloud backend asset in smaller logical entities. This way changes to some area might not require deploying the whole service but only the service containing changes which is faster. It would also enable to leave out not needed entities of the deployment more easily. This architectural improvement would moreover improve and ensure loose coupling between the services and this way limit risk of breaking other services.

Unit tests play an important role on verifying changes in code did not break any existing features. Unit tests can be used to test for example functions to work as intended. The great benefit can be seen for example when code refactoring is in place and the newly created function should work similarly than predecessor. Good unit tests and high enough unit test coverage bring great deal of security to the developers making changes. However, they do require commitment and extra time to keep them up-to-date and that is why they were not on focus area during the project. Nevertheless, it is definitely an area that requires more attention in the future.

During the development phase certificates were assigned manually to all IoT devices. However, it would be better to automatically handle device certificate creation when a device first time connects to the IoT Core and requests for device provisioning. In that case the device would have initial certificate which would allow only to publish MQTT message to the “handshake” topic. The Lambda function handling device provisioning would handle certificate creation and respond to device with the information where to download the right certificate. The device would download the certificate and take it in use and then be capable of communicating with the cloud properly. This change would minimize the need to someone manually create certificates and therefore certificates would not be stored in several places where they are not actually needed. Also, risk of certificates getting leaked or stolen would be smaller. The change would also enforce usage of single certificate on single device without an extra effort. However, the proposed change also has its flaws. For example, what happens if the initial certificate gets leaked or stolen. Changes affecting security are not straight forward and therefore those should not be done without thinking the risks thoroughly.

Switching from current DynamoDB database solution to time series database would be beneficial because they perform better with time series data and natively support functions like averaging in queries. DynamoDB is normally fast but when the query covers very long time period the data set in question might grow vast and the query become slow. However, at the time of writing the thesis there was still not time series database publicly available in the Amazon Web Services. Nevertheless, eyes should be kept open for emerging possibly better alternatives for DynamoDB to store time series data.

6 Conclusions and discussion

As a result of this thesis, a general-purpose IoT cloud backend software asset was defined and implemented. The core idea was to implement asset which provides fundamental and general level features for cloud connected IoT devices. The commissioner of this thesis, SADE Innovations Oy, has identified several key features and requirements which are often requested by the client when discussing about their needs related on IoT cloud backend. The requirements set by the commissioner guided whole definition and implementation work. The commissioner is already using the asset as basis of new projects in favor of speeding up overall development work. The speed up is due to the time savings that would otherwise have gone into developing fundamental features. Faster development leads to cheaper project costs and eventually more time can be allocated to development of client specific features. Because of the common basis, overall quality of the asset can be paid more attention to by testing it and therefore the quality improves over time. The asset bases on cloud services provided by Amazon Web Services which is the main cloud service provider used by the commissioner and therefore was chosen for this project too.

The main principle was to take advantage of as many ready cloud service solutions as possible and minimize the need to develop equivalent solutions by yourself. The same principle applied also to third party provided open source software packages. For cloud services, the main focus was on cloud services which are automatically scaling based on usage, and costs follow the resource usage of the service in question. These principles contributed to minimal amount of source code to meet the requirements and it ensures that the system can be rather easily expanded. Furthermore, all implementations were developed universal applicability in mind which ensured the general purposiveness of the cloud backend. Usage of cloud services has enabled to minimize the need to maintain the infrastructure needed behind the backend.

At the time of writing the thesis, the general-purpose IoT cloud backend asset has been in use by several clients of the commissioner. Initial functional and non-functional requirements for the asset also proved to be correct. The asset has not yet required bigger or critical architectural changes so it can be said that the defined architecture has been success so far. However, the asset has been already developed further to meet new use cases. Nevertheless, it can be proudly stated that the asset has already proved its worth to the commissioner and its clients. To maintain its worth it is evident that the asset has to be developed further also in the future. Furthermore, it might be worth to develop support for other cloud service providers too. Although further development is critical in terms of continuity, all future improvements have to be thought critically from the perspective of universal applicability. All improvements should maintain or improve the already achieved universal applicability of the asset.

Developing and defining the general-purpose IoT cloud backend has been a huge privilege. It has forced to learn new things and to look for ways to circumvent different kind of issues met along the development process. It has also enabled growing as software developer and architect of the cloud-based backend and will also continue do so in the future. Even though the asset might not be as basis of some new project, it has provided many lessons learned and therefore its role as an informational basis cannot be denied.

References

Amazon Web Services. (2020a). *Amazon API Gateway: Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-dg.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020b). *Amazon Cognito: Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-dg.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020c). *Amazon DynamoDB: Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/dynamodb-dg.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020d). *Amazon Simple Storage Service: Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020e). *AWS AppSync: AWS AppSync Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/appsync/latest/devguide/appsync-dg%20.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020f). *AWS Identity and Access Management: User Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/IAM/latest/UserGuide/iam-ug.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020g). *AWS IoT: Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/iot/latest/developerguide/iot-dg.pdf [Accessed 2 Apr. 2020].

Amazon Web Services. (2020h). *AWS Lambda: Developer Guide*. [pdf] Amazon Web Services. Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf [Accessed 2 Apr. 2020].

Amazon Web Services, Inc. (2020). *Global Infrastructure Regions & Azs*. [online] Available at: https://aws.amazon.com/about-aws/global-infrastructure/regions_az/?p=ngi&loc=2 [Accessed 20 May 2020].

Brinda, M. and Heric, M. (2017). Adapting To The Cloud's Mainstream Future. Forbes, [online]. Available at: https://www.forbes.com/sites/baininsights/2017/03/16/adapting-to-the-clouds-mainstream-future/#45194f627f28 [Accessed 20 Mar. 2020].

Doglio F. (2015). *Pro REST API Development with Node.js*. Berkeley, CA: Apress.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [pdf] University of California, Irvine. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf [Accessed 24 Mar. 2020].

Foundation.graphql.org. (2020). *The GraphQL Foundation | An open and neutral home for the GraphQL community*. [online] Available at: https://foundation.graphql.org [Accessed 23 Feb. 2020].

Graphql.org. (2020a). *GraphQL: A query language for APIs.*. [online] Available at: https://graphql.org/ [Accessed 23 Feb. 2020].

Graphql.org. (2020b). *Introduction to GraphQL*. [online] Available at: https://graphql.org/learn/ [Accessed 26 Feb. 2020].

Heino, P. (2010). *Pilvipalvelut*. Helsinki: Talentum.

Microsoft. (2016). *TypeScript Language Specification*. [pdf] Microsoft. Available at: https://github.com/Microsoft/TypeScript/blob/master/doc/TypeScript%20Language%20Specification.pdf?raw=true [Accessed 3 Apr. 2020].

Mqtt.org. (2020). *FAQ - Frequently Asked Questions | MQTT*. [online] Available at: http://mqtt.org/faq [Accessed 23 Feb. 2020].

Mukhopadhyay, S. C. and Suryadevara, N. K. (2014). Internet of Things: Challenges and Opportunities. In: S. C. Mukhopadhyay, ed., *Internet of Things: Challenges and Opportunities*, 1st ed. Cham: Springer International Publishing, pp. 1-17.

Richter, F. (2020). Amazon leads $100 Billion Cloud Market. Statista, [online]. Available at: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ [Accessed 27 Mar. 2020].

Salo, I. (2010). *Cloud computing: Palvelut verkossa*. Jyväskylä: Docendo.

Serverless.com. (2020a). *Serverless Framework - AWS Lambda Guide - Introduction*. [online] Available at: https://serverless.com/framework/docs/providers/aws/guide/intro/ [Accessed 3 Apr. 2020].

Serverless.com. (2020b). *Serverless - The Serverless Application Framework Powered By AWS Lambda, API Gateway, And More*. [online] Available at: https://serverless.com/ [Accessed 21 Apr. 2020].

Spec.graphql.org. (2018). *GraphQL*. [online] Available at: http://spec.graphql.org/June2018/ [Accessed 26 Mar. 2020].

Srinivasan, S. (2014). *Cloud Computing Basics*. New York, NY: Springer New York.

Stigler, M. (2018). *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Berkeley, CA: Apress.

Uckelmann, D., Harrison, M. and Michahelles, F. (2011). An Architectural Approach Towards the Future Internet of Things. In: D. Uckelmann, M. Harrison, F. Michahelles, eds., *Architecting the Internet of Things*, 1st ed. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1-24.

Vijayakumar, T. (2018). *Practical API Architecture and Development with Azure and AWS: Design and Implementation of APIs for the Cloud*. Berkeley, CA: Apress.

Wang, V., Salim, F. and Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket*. Berkeley, CA: Apress.