Cosinus

Monitoring Electric Vehicles

Son Thanh Vo

Thesis submitted for the degree of
Master in Programming and Network
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Cosinus

Monitoring Electric Vehicles

Son Thanh Vo

© 2019 Son Thanh Vo

Cosinus

http://www.duo.uio.no/

Printed: Simula Research Laboratory, Fornebu

Acknowledgements

We want to thank you everyone who has assisted us with the work regarding this master thesis — especially those who have engaged in discussions without any consideration to time and place. Through our discussion, we have received and deepened our knowledge regarding a wide variety of different domains and polished our general programming skills. We would also like to give a huge thanks to everyone who has kept out mental state healthy during this time.

Moreover, we appreciate the human body and the things it is capable of achieving — specifically our legs, which has carried us through adventures and battles, which has assisted in keeping the mind healthy.

Another notable contributor to this thesis is our dear laptop, which has survived almost five years of rough use. It became a part of our life when it replaced its predecessor, right before an assignment was due, and has been running flawlessly ever since.

Thank you to everyone who has assisted us in some way through the master degree!

Abstract

Advances in the development of electric vehicles along with policy incentives have resulted in a tremendous growth of private electric transportation. However, the fast-paced adoption of electric cars could lead to unfavorable effects on residential power distribution networks. The effects include overloading of power components and instability in voltage throughput and will occur mainly during simultaneous charging of large numbers of electric vehicles.

Data on charging trends and driving trends is required to support the growth of renewable transportation. Battery technology and capacity continue to grow, and the key to a sustainable green future is how well we understand the power demands of an electric vehicle.

Our work aims to address the lack of driving data and charging data to improve the understanding of electric vehicles power demand. We develop the Cosinus system to monitor electric vehicles and interact with electric vehicle owners and researchers who are interested in the data. Understanding power demands are the first step to creating charging schedules that provide sufficient power to electric vehicle owners while minimizing the risk of component overload and voltage instability.

We conclude that a system that collects driving data and charging data must prioritize security, to protect its users and flexibility, to allow the system to grow and change rapidly with the introduction of new electric vehicle brands and APIs. Electric vehicle batteries are increasing in capacity and understanding driving patterns will help tremendously in the development of more advanced and sophisticated charging methods that will result in more economical charging for the car owner and power distributors.

Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Background and Motivation . . . 1
1.2 Problem Definition . . . 3
1.3 Limitations . . . 4
1.4 Research Method . . . 4
1.5 Main Contributions . . . 5
1.6 Thesis Outline . . . 5

I Background 9

2 Background 11

2.1 Security . . . 11
2.1.1 OWASP Top 10 . . . 11
2.1.2 General Data Protection Regulation . . . 13
2.1.3 Authorization . . . 14
2.2 Django . . . 17
2.2.1 Overview of Django . . . 17
2.2.2 Django Models . . . 18
2.2.3 Django rest-framework . . . 20
2.2.4 Serializers . . . 21
2.2.5 Views . . . 22
2.3 Hosting . . . 23
2.3.1 Cloud Computing . . . 23
2.3.2 Services . . . 26
2.3.3 Compute Engine . . . 26
2.3.4 App Engine . . . 26

2.3.5 Cloud Functions 26
2.4 Data 27
2.4.1 Power Market and Power Data 27
2.4.2 Vehicle Usage Data 28
2.5 Vehicle Communication 29
2.5.1 Tesla 30
2.5.2 BMW 32
2.6 Terminology 33
2.6.1 CPU Scheduler 33
2.6.2 Process 34
2.6.3 Thread 34
2.6.4 Coroutine 35
2.6.5 User Experience 35
2.6.6 Native Applications 35
2.6.7 Cross-platform Applications 36
2.6.8 Technical Debt 36
2.7 Summary 36

3 Related work **39**
3.1 Summary 41

II Design & Development **43**

4 Cosinus **45**
4.1 Overview 45
4.1.1 Infrastructure 45
4.1.2 Architecture 47
4.1.3 Database 49
4.2 Authentication and Authorization 50
4.2.1 Authentication 51
4.2.2 Authorization 54
4.3 Brand Manager 54
4.3.1 Service Interface 56
4.3.2 Brand Manager Core 58
4.3.3 Service Components 60
4.4 Interacting with the system 61
4.4.1 Resource addresses 61
4.4.2 Query The Collected Data 61
4.4.3 Connect vehicles to the system 63
4.4.4 Interaction with the auth endpoints 65

4.5 Summary 67

5 Monitor Optimization **69**
5.1 Cron Jobs 69
5.2 Current Data Collection Design 70
5.3 Benchmarking 71
5.4 Approaches 73
5.4.1 Multiprocessing 73
5.4.2 Multithreading 74
5.4.3 Coroutines 76
5.4.4 Grid Computing 77
5.5 Conclusion 77
5.6 Summary 79

6 Tailored Charging Schedules **81**
6.1 Charging Plan 81
6.2 Automated Charging 83
6.3 Estimating Driving Range 85
6.4 Discussion 89
6.5 Summary 90

7 Client Application **91**
7.1 Architecture 91
7.2 Cross-platform frameworks 92
7.3 Navigation 93
7.4 Building Interfaces 94
7.5 Device and Network Communication 95
7.6 Programming Language 96
7.7 Conclusion 96
7.8 Summary 97

III Conclusion **99**

8 Conclusion **101**
8.1 Summary 101
8.2 Main Contributions 102
8.3 Future Work 103
8.4 Final Remarks 104

IV Appendices **105**

A Monitor Optimization **107**

B Tailored Charging Schedules **111**

C Client Application **121**

List of Tables

2.1 Session token overhead with 20 runs 15
2.2 JSON Web token overhead with 20 runs 16
2.3 Cloud Categories 25

4.1 Essential system resource addresses 61
4.2 Available arguments for data retrieval 64

5.1 Multiprocessing I/O test results 74
5.2 Multithreading I/O test results 75
5.3 Coroutines I/O test results 77
5.4 Accumulated I/O test results 78

List of Figures

2.1 Illustration of session token flow 15
2.2 Illustration of JSON web token flow 16
2.3 Caption place holder 18
2.4 Django models lazy evaluation example 19
2.5 Django model 19
2.6 Custom Django model field example 20
2.7 Django models inheritance 21
2.8 Serializer example 22
2.9 Login serializer 22
2.10 Login view 23
2.11 Correlation between price and consumption 27
2.12 Weekly vehicle trends 28
2.13 Simple CPU scheduler 34

4.1 Illustrations of the system architecture 48
4.2 Illustrations of the database schema for vehicle data 50
4.3 Creating a new user 52
4.4 Generate access and refresh token 53
4.5 Authorization filter 55
4.6 Service interface 57
4.7 Brand Manager Core 58
4.8 Data Django models 60
4.9 Service components pseudo code 62
4.10 Connect curl command 64
4.11 Registration curl command 66
4.12 Login curl command 66
4.13 Refresh access curl command 66

5.1 Screen grab of monitor execution times 70
5.2 Performance of various test cases 72
5.3 Multiprocessing performance 73
5.4 Threading performance 75

5.5 Coroutines performance 76
5.6 Parallelization test results 78

6.1 Accumulation of trend data 82
6.2 Charging recommendation 83
6.3 Correlation between battery efficiency and temperature 86
6.4 Estimated monthly driving range 87
6.5 Estimated monthly driving range 88

7.1 Client Application Architecture 92
7.2 Navigation Examples 94
7.3 HTTP request example 95

A.1 System specification 108
A.2 Source code: Busy waiting 108
A.3 Source code: Find all prime numbers below n 109

B.1 Heatmap of vehicle one charging trends 112
B.2 Heatmap of vehicle two charging trends 112
B.3 Power prices and Consumption data for the first two weeks of February 113
B.4 Hourly with distance filters 113
B.5 Daily with distance filters 114
B.6 Daily Simple with distance filters 114
B.7 Greedy with distance filters 115
B.8 Hourly with efficiency filters 115
B.9 Daily with efficiency filters 116
B.10 Daily Simple with efficiency filters 116
B.11 Greedy with efficiency filters 117
B.12 Hourly with distance and efficiency filters 117
B.13 Daily with distance and efficiency filters 118
B.14 Daily Simple with distance and efficiency filters 118
B.15 Greedy with distance and efficiency filters 119

C.1 Flutter: rendering list of vehicles 122
C.2 react native: rendering list of vehicles 123

Chapter 1

Introduction

1.1 Background and Motivation

Electric Vehicles (EVs) is an environmentally friendly product that will reduce local and global pollution generated from both private and public transportation. EVs is rising in popularity as the preferred choice of transportation, especially here in Norway, which is a world leader in the adoption of EVs. The tremendous growth in EV sales is due to government incentives to make EVs an attractive option for car buyers, to reach their goal of having all vehicles sold after 2025 to be emission-free [23].

The aggressive push for a greener future with less local pollution is, however, not without its complications. The big batteries installed in the EVs require adequate power to operate the vehicle and travel between destinations. The rising power demand present problems for power distributors. Many residential areas power grid cannot stably support the high throughput required to simultaneous charging a high number of EVs [10, 32].

Charging a single EV can be the equivalent of adding three houses to the power grid. The added pressure does, however, vary depending on the charging method, with charging through a standard outlet resulting in the least amount extra stress on the power grid and dedicated EV chargers drawing a significant amount of power in a short amount of time [6]. Dedicated EV chargers can charge EVs swiftly to allow for quick pit-stops whenever needed.

The high voltage throughput is, however, not healthy for the batteries [30]. The added stress will result in quicker deterioration of the batteries. The power grid of residential areas can exceed their capacity, resulting in power components overloading and voltage instability [10, 32, 36]. The increased power demand does, however, present the potential to fill the valleys in the load curve, which will result in more electricity sales without the need for power grid

upgrades.

Additionally, modern vehicles make use of a tremendous amount of sensors to ensure stable operation. All the captured information from the sensors give investigators and car manufacturers and accurate representation of the operation of the vehicle. All the sensor information results in a tremendous amount of logged data which is an untapped source of information which can be analyzed by third-parties. Especially for EVs, where the data can provide an accurate representation of the EVs power demands and their EV owners trends. The data present significant potential for an analyst to improve current charging solutions.

The challenge, however, is that the car manufacturers carefully protects the data, which makes it almost impossible for others to retrieve and access the information stored in the driving logs. However, car manufacturers have begun to allow access for a subset of the information through dedicated mobile applications. Additionally, the introduction of the General Data Protection Regulation (GDPR) has further added pressure on full transparency for remotely stored personal data. However, the data is still challenging to retrieve and access, which discourages researchers who have an interest in the data from attempting to access the data.

In the last few years, a rising trend of using machine learning has emerged to understand large amounts of data. Machine learning is a field of computer science where the goal is to give algorithms the ability to learn and improve their performance on a specific task. The advances in analyzing methods such as machine learning have turned data into a valuable resource.

In the last few years, a rising trend of using machine learning has emerged to understand large amounts of data. Machine learning is a field of computer science where the goal is to give algorithms the ability to learn and improve their performance on a specific task. We have observed success with machine learning methods in various fields, including power and health-care. However, machine learning requires a large amount of data to understand and categorize problems.

With the rise of cloud services, we see opportunities for solutions to retrieve and store vehicle data across car manufacturers. Cloud services have made it easier than ever before to deploy systems capable of interacting with thousands of users and services. Especially services like Google App Engine, where infrastructure is configured and operated by the cloud provider has resulted in a decrease in time between project initialization and production ready deployment compared to conventional methods.

We see that cloud services can be leveraged to monitor and collect information about individual EVs. The monitorization process will, in turn,

generate enough data for machine learning models to understand and optimize the charging procedure in respect to the vehicle owners usage. This process will result in a more advanced system which solves the problem better than the available state of the art solutions, as it can predict and account for future usage.

The technological advances in transportation and cloud technology, combined with data regulations has resulted in an excellent opportunity to make data more accessible for both EV owners and researchers. The vehicle data can result in better individual and collaborative charging optimization, in addition to gaining a better understanding of driving trends.

1.2 Problem Definition

As indicated in the previous section, EVs are a technological feat which is crucial to decrease pollution from private and public transportation. The problem, however, is that EV sales are moving at a tremendous rate, which brings new user patterns and problems for power distributors. Charging an EV adds a considerable amount of load on the power grid, which can cause voltage drops and power components such as transformers and feeders to overload, which decreases the components life-span.

To prevent power grids from experiencing voltage drops and power components from overloading, we will design and develop a system capable of interacting with EVs to retrieve and store vehicle states to analyze and understand EV needs. Car manufacturers hide driving data from the users and retrieving a copy for researchers can be quite the challenge. We are therefore in need of a solution that is capable of collecting driving and vehicle state data for a considerable amount of vehicles.

In this thesis, we will investigate the following:

1. *How should a system be built that can fulfill the task of automatically retrieving and storing driving data?*
   Driving data has been, and still is, a challenge for vehicle owners and researchers to access. The rising trend of wireless interaction with mundane items has encouraged car manufacturer to create opportunities for car owners to retrieve the current state of their car through a mobile app. We examine different cloud services and architectural choices to determine how to create a driving data collection system.

2. *How can the driving data be used for smarter and more economical charging?*
   Understanding the vehicle and the owner's patterns will result in flexibility when it comes to charging times. We will, therefore, use the acquired data to improve our understanding of user trends and vehicle

capabilities, to provide a more flexible charging schedule compared to a general off-peak charging routine.

The ultimate goal of this thesis is to make driving data more accessible for the vehicle owner and researchers to improve current collaborative charging approaches. Hopefully, the data acquired through the developed system will aid future research in their work on component overload prevention and improve power components utilization. Additionally, the data will potentially result in the development of more sophisticated collaborative charging schedules that provide an economic advantage for power distributors and vehicle owners.

1.3 Limitations

Based on our questions defined in the previous section, the scope of this thesis is to design and develop a system capable of automatically retrieve and store driving data and exploring potential charging scheduling approaches made possible by the collected data.

We limited the scope of potential programming languages and cloud providers based on criteria presented by Flexibility AS, which is one of the collaborators for the work done in this thesis. Flexibility had a desire of having the system developed using the Python programming language. Further, the system must use a cloud service provided by Google Cloud Platform. We decided to limit the number of cloud services evaluated to the three we deemed to be most flexible in use, which are Google Compute Engine, Google App Engine, and Google Cloud Functions.

Additionally, instructions for remote vehicle interaction is limited, where most of the information is reverse engineered by dedicated software engineers around the globe. We, therefore, limited the number of supported vehicle brands to two, with Tesla, because of the detailed and accurate instructions found on their forums and BMW, which agreed to assist the work.

1.4 Research Method

We chose the Constructive Research methodology, better known as Design Science Research (DSR), to conduct our research in this thesis. DSR is a form of scientific knowledge production that involves the development of constructions, intended to solve problems we face in the real world, and simultaneously make a prescriptive scientific contribution [9].

DSR occupies a middle ground between traditional scientific approaches, mostly descriptive and context-related problem-solving knowledge produced in practical situations. The research methods aim is to the solving of specific problems and obtains a satisfactory solution for the situation, even if the solution is not optimal [9].

During our research, we will follow the design cycle method, which is a subgenre of DSR. Research following the design cycle method begins with the identification of the problem and understanding how they should define the performance required to benchmark the solutions. Afterward, the researcher compares a variety of solutions to find a suitable solution for the problem [9].

1.5 Main Contributions

We have provided a system capable of retrieving and storing driving data and charging data for vehicles which support remote communication. We have used data collection and storage of vehicles as a scenario to explore how different architectural choices affect the maintainability of the systems and its capabilities to operate as an aggregation platform. We have also analyzed the historical driving data and charging data to improve our understanding of EVs power demands and mileage trends.

To achieve this, we identified problems and explored a variety of techniques to create loosely coupled components and abstraction throughout the system. The resulting system can rapidly expand in features and components can be changed without directly affecting the operational ability of the system. The most significant challenge of the system was to provide an acceptable security level. We protected the data through the use of customized Data Access Objects (DAO) to provide an intuitive interface for developers without the requirement of understanding how the database operates.

1.6 Thesis Outline

We divided the thesis into three parts, with the first containing crucial information to understand architectural choices and present related work associated with EV charging. The second part contains the work done throughout the thesis, and the final part is the thesis conclusion. The thesis is structured as follows:

Part 1: **Background**

Chapter 2: Background: We give essential insight into topics which has shaped the design and development of the Cosinus system. We introduce privacy regulations and security attack vectors, which we mitigate through the use of customized Data Access Objects, input validation, and hierarchical access control with the help of complementary libraries and frameworks. Further, we present a description regarding the Application Programming Interfaces used to collect driving data.

Chapter 3: Related Work: We present related work focused on raising concerns related to the dangers EVs present on the power grid and how near real-time power profiling combined with voltage manipulation can mitigate the problem. Finally, we discuss why our solution is a necessity to complement existing voltage manipulation methods and support a green future.

Part 2: **Design & Development**

Chapter 4: Cosinus We introduce the Cosinus system and the thought process that went into designing and developing the system with security and privacy in mind. Most importantly, we describe how we modularized the system to keep the maintainability of the system at an acceptable level. Through the use of design principles like "Do not repeat yourself" and object-oriented design, we made it possible to support more brands with no prerequisite knowledge of the system.

Chapter 5: Monitor Optimization: We initially designed the data collection process to run chronologically. However, we had to reduce the time needed for the data collection process, to operate inside the boundaries given by the cloud provider. We present and discuss a small experiment to highlight how common parallelization strategies compare with our use-case in mind.

Chapter 6: Tailored Charging Schedules: Encouraging EV owners to share their driving data requires us to reward the participant in some fashion. The chapter takes a closer look at the driving data and discusses how the data can be used to provide each EV owner with their custom charging plan in addition to a historical view of their EVs range throughout the time of monitorization. Finally, we discuss how the minimum charging necessary approach presented can impact collaborative charging methods, especially in conjunction with existing voltage manipulation methods.

Chapter 7: Client Application: Developing and maintaining a mobile application is crucial to reach and interact with users. This chapter presents the comparison of a new framework which is currently rising in popularity, Flutter, with a framework which is a near identical copy of the popular web development framework React, react native. Both frameworks provide a hierarchical structure over the user interface elements. However, their internal structure and accompanying libraries result in one being better for collaborative and long-time development, while the other being better for prototyping, where quick changes and easy manipulation of position and size is favored.

Part 3: **Conclusion & Future Work**

Chapter 8: Conclusion: Finally, we summarize and conclude this thesis and present ideas and suggestions for further studies surrounding the work done.

Part I

Background

Chapter 2

Background

2.1 Security

Information security refers to the process and tools designed to protect sensitive information from modification, disruption, destruction, and inspection. It is essential to be aware of vulnerabilities to develop a secure system.

This section introduces vital topics which shape the system, to protect the information stored inside our software. We begin by introducing a few of the most common exploits used by malicious actors to access and tamper with the system.

After the introduction of common threats, we provide a brief introduction to the General Data Protection Regulation, which affects all services used by a European Union citizen. Finally, we introduce two common authorization methods, session token, and JSON Web Token.

2.1.1 OWASP Top 10

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop and maintain applications and APIs that can be trusted. OWASP provides information and tools to educate organizations and raises awareness for the most common security vulnerabilities in existence [31].

This section introduces the most common risks associated with service development and maintenance. Some of the exploit mentioned here can put an organization out of business and cannot be avoided under any circumstances.

Injection

An injection attack is an attack where a malicious actor tricks the system into executing foreign code. The purpose of an injection attack is to read, modify or change data stored on the system. Injection attacks, such as SQL injection are among the oldest and most dangerous attack vectors, which can result in the attacker gaining control over the system [21, 31].

The biggest problem with injection attacks is the broad attack surface. All input that goes into the system or is used by the system is a possible entry point for an attack. Environment variables, parameters, libraries, internal services, and external services can all be used to carry out an attack [21].

Injection prevention requires a separation between data, commands, and queries. Escaping special characters or using Object Relational Mapping tools is the most straightforward method to prevent SQL injection. Another conventional method is to validate inputs before use [31].

Broken Authentication

A broken authentication attack attempts to find vulnerabilities in the systems authentication process. The purpose is to gain system access with the goal of accessing the system as an administrator [3, 31].

Broken authentication attacks attempt to brute force their way through, by attempting multiple different usernames and password combinations, either manually or automatically [3].

The most straightforward method to prevent an attack is to implement password strength validation. Other excellent measures are to blacklist common passwords, implement multi-factor authentication and limit number of login attempts allowed [31].

Sensitive Data Exposure

Sensitive data exposure, as the name implies, concern unprotected or poorly protected data. The data in question can be everything from passwords and tokens to sensitive data such as financial or health information [31, 38].

An attacker either acts as a man-in-the-middle or acquires the data through other means, like an injection attack. From there, they try to break through the encryption if present. The retrieved data can then be used to cause financial loss, discrimination or, worst-case, start a witch hunt [31].

Adequately protecting all information which enters or leave the system is the best and only method to prevent this from happening. Avoid storing unnecessary data and encrypt sensitive information. Moreover, encryption

algorithms should be up to date and follow reliable standards. System administrators should also ensure that encryption keys are secure, both in their content and the location they are stored [31].

Broken Access Control

Access control enforces policies such that users cannot act outside their intended permissions. Broken access control can result in sensitive data being exposed, with the possibility of losing control over the system [31].

Access control is legitimate only when enforced, which is why the easiest prevention method is to enforce access policies. The access control must be enforced in a trusted environment where attackers are unable to modify metadata and disable validation processes. Other prevention methods are to ensure that resources can only be viewed and modified by the resource owner [31].

Rouge Extensions

Relying on third-party software, such as libraries and frameworks, is a common strategy to speed up development time and increase quality. It provides abstractions and can result in more maintainable code. There is, however, risks associated with importing third-party software.

The impact of using rouge extensions varies depending on the protected assets. However, data is not the only aim of the attack. Rouge extensions might even result in lost control over the system, and further use it to attack others.

The most straightforward procedure to avoid rouge extensions is to avoid unmaintained libraries and frameworks, in addition to monitoring activity from the maintainers. Other methods are to remove unnecessary and unused extensions and ensure that all files come from a trusted source [31].

2.1.2 General Data Protection Regulation

The General Data Protection Regulation (GDPR) is a regulation in Europe on data protection and privacy. The regulation aims to protect all individuals within the European Union (EU) and the European Economic Area (EEA). It addresses all data, either inside or outside of EU and EEA that affect an EU citizen [11].

GDPR aims to protect EU and EEA citizens from privacy and data breaches in a digital age. It focuses primarily on reducing the severity of data breaches and informs citizens of organizations who store personal data. In the following sections, we will cover some of the essential features of GDPR.

Consent

Organizations are no longer able to use terms of service and conditions which is difficult to read and understand. A request for consent must be simple to read and understand. It must also be easily accessible, and the purpose of storing the information must be attached to the consent. In other words, a consent must explain how the company or organization use the information. Further, it must be as easy to withdraw consent as it is to give it [11, 17, 35].

Right to Access

All EU and EEA citizens have the right to receive a confirmation whether an organization or company has information associated with them stored in their systems. If that is the case, then the organization is obligated to explain the purpose of storing the information. Further, organizations and companies are obligated to provide a digital copy of the data, free of charge, on request [11, 17].

2.1.3 Authorization

This section introduces authorization followed by an introduction to two conventional methods for authorization toward a system. More precisely, the section will introduce session token and JSON Web Token.

Authorization is the process of specifying and granting a users rights to system resources. More formally, authorization is to define an access policy. E.g., a bachelor student cannot enter the school buildings on Sunday after 6 pm. A master student, on the other hand, can enter the school buildings whenever they want.

Tokens are digital artifacts used to identify the token holder, similar to a building access card. The system provides the tokens after the authentication process, where a user is required to present their credentials, usually a username and password, which is validated by a trusted source.

Session Token

A session token is a unique data-element generated by the system. The token is preferably difficult to forge, such as artifacts that are digitally signed or encrypted. The token must be present whenever the client request to communicate with the system and acts as a key to session information which is stored in the system's database.

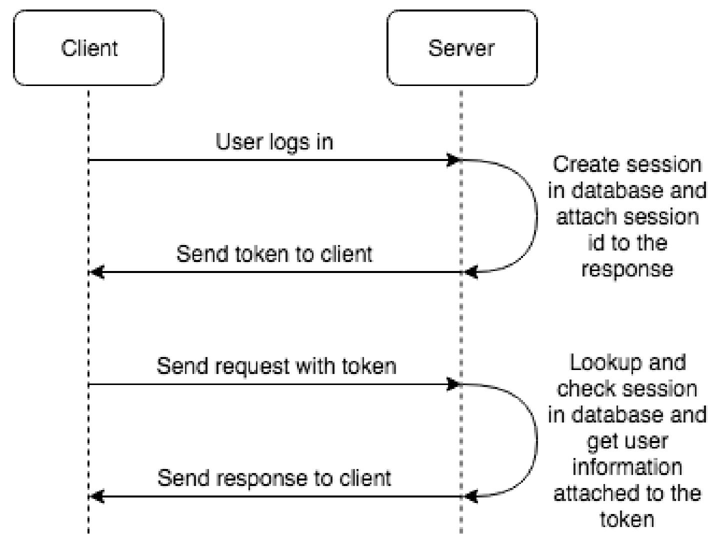


Figure 2.1: Illustration of session token flow

threads	requests\thread	avg. microseconds
20	1000	16.211914999999998
20	20000	15.521109
20	40000	15.372620374999997

Table 2.1: Session token overhead with 20 runs

Storing session information inside the system has the advantage of reducing data exposure and a reduced amount of transmitted data, compared to JSON Web Tokens. Besides, storing meta information grants the opportunity of storing a large amount of detailed information, with the possibility of locking tokens to the users IP [26].

The consequence of this process is that it relies heavily on the database. Every incoming request must be validated with a database read operation. There are however methods to reduce the database dependency by storing information inside the token, which might weaken the protection.

Figure 2.1 illustrates the usual request flow while table 2.1 shows the average time used for a database read to illustrate the overhead associated by using session token. The test was run locally with both the server and database present to reduce network latency as much as possible.

The most significant advantage of using session tokens regarding security is the ability to revoke access privileges whenever needed and its ability to prevent tokens from being used by other IPs.

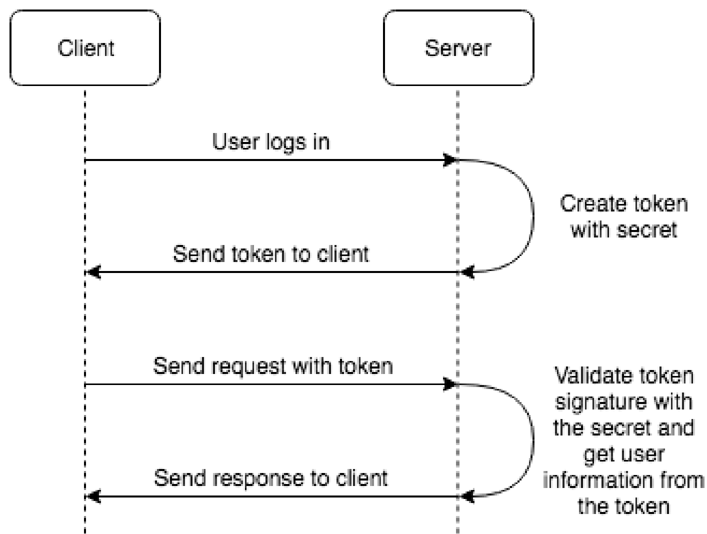


Figure 2.2: Illustration of JSON web token flow

threads	requests\thread	avg. microseconds
20	1000	13.02477
20	20000	12.35035575
20	40000	12.372653875

Table 2.2: JSON Web token overhead with 20 runs

JSON Web Token

A JSON Web Token (JWT) is a unique data-element produced by the system. JWTs consists of two JSON objects (head and payload) and a digital signature (tail) to verify the content and authenticity of the token.

JWTs represent a set of claims as a JSON object that is encoded. This JSON object is the JWT claim set. The JSON object consists of name-value pairs, where the names are strings, and the values are arbitrary JSON values. A claim is the names present in the object [24].

The contents of the head describe the cryptographic operations applied to the token, while the body contains the shared information between user and system. The token is represented as a sequence of URL-safe parts separated by a period character. Each part contains a base64url-encoded value.

The advantage of JWTs is that it is stateless. With stateless, we refer to the process of remembering information. The token has all the necessary data to inform the system about who the user is in addition to meta information about the token itself.

Figure 2.2 illustrates the usual request flow while table 2.2 shows the average

overhead associated with authorizing the request. The test was run locally to reduce overhead related to network latency as much as possible.

Another advantage of JWTs is its ability to be used by multiple services and systems. It can, for example, be created by Google and used by Gmail and YouTube. Additionally, it has excellent performance, because of the computational validation compared to a database lookup.

The disadvantage, however, is that the system can not reject non-expired token. Compromised tokens have access right to the system until the expiration date. The set life-span of a token is a significant contributor to the level of security the token is capable of providing.

2.2 Django

Using libraries and framework reduces development time with the possibility of assisting the developers in producing maintainable software. Using third-party libraries is, however not without any risks, as we mentioned in chapter 2.1.

Choosing the correct combinations of frameworks and libraries to complement a system is difficult. Choosing many small libraries might result in much technical debt, as there would be many dependencies to monitor and maintain. On the other hand, using a broad framework or library will also result in some technical dept, as the system will become tightly coupled with the extensions.

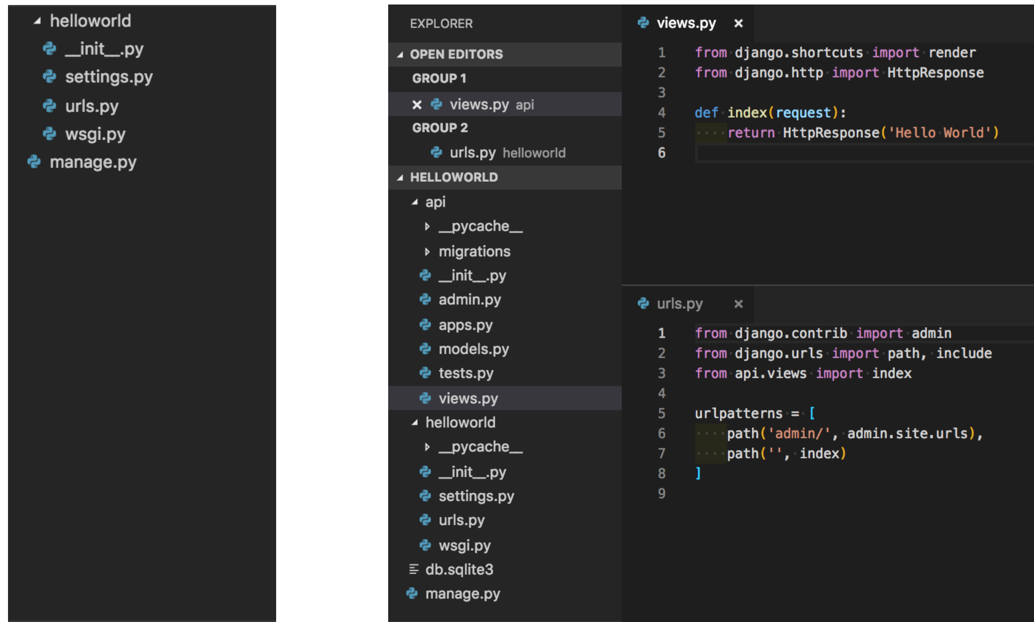
After considering the options available, we chose to use extensions with broad feature sets instead of small libraries, to reduce the risk of encountering a rouge extension, while keeping the number of dependencies to a minimum.

This section introduces the framework and libraries which the system relies on the most, Django and Django rest-framework. The reason why we chose Django was that it has an extensive feature set in addition to its built-in security features to assist us in avoiding vulnerabilities mentioned in section 2.1.1.

Django is a full-stack web framework, with an extensive feature list. The framework has everything a system need to administrate their users and database. During this chapter, we introduce the database related features of Django in addition to Django rest-frameworks APIView and serializer.

2.2.1 Overview of Django

Django is an open-source web framework based on the programming language Python. It follows a model-view-template (MVT) based architectural pattern, which facilitates server rendered views.



(a) Django initial tree view

(b) Django hello world

Figure 2.3: Caption place holder

Django makes it easy to create database-driven websites with its Data Access Objects (DAO), which is an Object Relational Mapping (ORM) tool. In addition to a straightforward API for database interaction, Django also protects against vulnerabilities such as SQL injection, Cross-site scripting (XSS), and Cross-Site Request Forgery (CSRF).

A Django project is simple to set up with the Command Line Tool (CLI) which generates everything that is needed to begin a project. Figure 2.3 displays all the generated from the project initiation command and module creation command. The only thing needed for a simple hello world response from the system is to insert code into views and register an endpoint in URLs as illustrated in the figure.

There is also a file named manage, which is generated by the CLI tool. Manage is a utility that assists in project management. It is capable of creating users, creating modules, such as 'api' in figure 2.3b and ensure a one to one relation between defined schemas and the database tables.

2.2.2 Django Models

Django models which are a DAO implementation creates an abstraction layer around the database. The models provide an extensive interface for interacting

```
q = Entry.objects.filter(headline__startswith="What")
q = q.filter(pub_date__lte=datetime.date.today())
q = q.exclude(body_text__icontains="food")
print(q)
```

Figure 2.4: Django models lazy evaluation example

```
class Person(models.Model):
    name = models.CharField()
    age = models.IntegerField()
```

Figure 2.5: Django model

with the database in addition to preventing SQL injection.

Another significant benefit is SQL query optimization. Django models are lazy, which means that filter operations can be applied multiple times as illustrated in figure 2.4. The database hit occurs at the moment information is retrieved from the object, which means that an object can be modified multiple times before the hit occurs.

A model is defined similarly as a Java class, where global attributes are defined as illustrated in figure 2.5. The model inherits from the superclass 'Model,' which provides the necessary interface to interact with the database. Also, each defined model is a representation of a database table with similar column names, which the 'manage' tool uses to check whether or not the database is up to date with the application.

The model interface provides a broad amount of different fields to support most use cases. However, it does not come with any method to automatically encrypt data. Creating custom fields is however a simple process. Figure 2.6 illustrates a simple method to create a custom field, where the class inherits the binary field and modifies how data is read and written.

The functions from_db_value and get_db_prep_value is invoked automatically whenever a read or write operation occurs respectively. Inside the functions, we can manipulate and change the values as demonstrated.

Another great feature with Django models is its integration with the programming paradigm object-oriented programming. The models have inheritance, as demonstrated earlier. However, there is more to the inheritance capabilities than illustrated in the earlier figures. There are three styles of inheritance which is supported, with multi-table inheritance being the most interesting for our system. The three styles provide a good set of options too

```python
class EncryptionField(models.BinaryField):
    def get_db_prep_value(self, value, connection, prepared=False):
        f = Fernet(settings.CRYPTOGRAPHY_KEY.encode())
        value = f.encrypt(str(value).encode())
        return super().get_db_prep_value(value, connection, prepared)

    def from_db_value(self, value, expression, connection):
        if value is None:
            return value
        elif isinstance(value, memoryview):
            f = Fernet(settings.CRYPTOGRAPHY_KEY.encode())
            value = f.decrypt(bytes(value))
            return value if value != b'None' else None
        return value
```

Figure 2.6: Custom Django model field example

represent different database needs.

Abstract base class: The parent is provides a template and does not have its own database table.

Multi-table inheritance: Every model defined has its database table where children has a direct pointer to a row in its parent table.

Proxy models: Modifies a models behavior without changing its fields.

Multi-table inheritance, as illustrated in figure 2.7, automatically maintains a link between the two models, the superclass and the subclass. The subclass has a pointer (foreign key) automatically inserted whenever the system creates a new row. Information inserted into the subclass is divided between the parent and child wherever applicable.

The advantage is the option to retrieve name and address whenever a park or restaurant is retrieved. Also, 'place' operates as a gathering spot for all the data, which means that the 'place' table has information related to all restaurants and parks.

2.2.3 Django rest-framework

Django's request and response features are mundane with the minimum needed to for API development. We, therefore, decided to include Django rest-framework to improve the system's request handling capabilities.

```python
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)

class Park(Place):
    has_fountain = models.BooleanField(default=False)
```

Figure 2.7: Django models inheritance

Django rest-framework is an extension library for Django. The library provides a set of pre-defined request handling cases in addition to serializers, which is a Django model like feature for request and responses. The library does not differ substantially from Django's procedure of creating class-based views, which provide seamless integration between the framework and library.

This section provides a fundamental introduction to the library with the focus of providing information regarding the essential features of the library. We begin by introducing serializers and finish with a brief introduction to views.

2.2.4 Serializers

Serializers allow for a complex data type, such as JSON, XML and URL query parameters to be translated to a native python datatype and vice versa. Serializers also provide automatic and manual validation of the incoming data, which provide security assistant whenever applicable.

Declaring a serializer is similar to declaring a Django model as illustrated in figure 2.8. Serializers can replicate a model, which provides a correct representation between the endpoint interface and database schema. We can also specify the desired fields, by specifying column names in a tuple instead of the illustrated '__all__' keyword.

Serializers provide object level and field level validation. Object level validation is primarily used to validate the relationship between two fields, while field level validation provides isolated data validation. The superclass provides automatic validation based on the type of field specified. However, manual validation is added with a validate function.

Figure 2.9 illustrates how we use field level validation to reduce the

```python
class PlaceSerializer(serializer.Serializer):
    name = serializer.CharField(max_length=50)
    address = serializer.CharField(max_length=80)

class ParkSerializer(serializer.Serializer):
    class Meta:
        model = Park
        fields = '__all__'
```

Figure 2.8: Serializer example

```python
class LoginSerializer(serializers.Serializer):
    email = serializers.EmailField(required=True)
    password = serializers.CharField(required=True)

    def validate_password(self, value: str) -> str:
        pattern = r"^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.{8,})"
        if re.match(pattern, value) is None:
            raise serializers.ValidationError(
                'Password does not meet strength constraints'
            )

        return value
```

Figure 2.9: Login serializer

likelihood of a broken authentication vulnerability from manifesting. Field level validation functions follows the format validate_<field_name>. Object level validation functions, on the other hand, are named validate, which takes a single dictionary as the argument.

2.2.5 Views

Django rest-framework provides a basic class, APIView, which subclasses Django's view class in addition to providing a set of views which provide commonly used patterns, such as retrieving a list of database entries.

Figure 2.10 illustrate how we created an endpoint which only accepts post request. The class forwards requests to a function of a corresponding method name, which provides a clear distinction between logic handling separate cases.

```
class LoginView(APIView):
    def post(self, request: Request) -> Response:
        payload = LoginSerializer(data=request.data)
        if not payload.is_valid():
            return Response(payload.errors, status=status.
                HTTP_400_BAD_REQUEST)

        return Response(<auth_pair>, status=status.HTTP_200_OK)
```

Figure 2.10: Login view

Additionally, Django rest-framework provides mixins, which is pre-built views to take common idioms and patterns found in API development and abstracts them to reduce development time. The mixins allow us to build quickly build abstractions between the client interface and the systems database schema.

2.3 Hosting

As mentioned in section 1.2, our system must be running on Google Cloud Platform, which is a cloud computing provider. Cloud computing is a relatively new paradigm which presents many architectural opportunities in addition to simplifying the process of making software publicly available.

This section introduces the cloud computing paradigm and the three services we considered for our system. Compute Engine, App Engine, and Cloud Functions provide different levels of infrastructure abstractions. We decided on these services because of their simplicity. Compute Engine and App Engine are both virtual machines with the difference in infrastructure configuration options, while Cloud Functions provides a simple environment for event-driven development.

2.3.1 Cloud Computing

Cloud Computing is a concept of outsourcing infrastructure management and maintenance. Traditional hosting solutions, like Digital Ocean, provides a fixed amount of computational resources. The other option is for companies and organizations to manage and maintain the infrastructure themselves, which again result in a fixed amount of computational resources [2, 15, 18, 47].

Cloud computing offers a theoretically limitless amount of computational resources, which is managed and maintained by the cloud provider. In other words, the cloud provider provides computational resources as a general utility, which can be rented by companies and organizations in an on-demand fashion. Cloud services will, as a result, remove the need for companies and organizations to manage and maintain their server hardware.

The result of using a cloud service is that more time can be allocated to developing software instead, which can provide a positive effect on the financial budget. The separation between infrastructure and software development has brought forward several compelling advantages and features, as listed below, in addition to increased specialization in their respective domains.

No upfront investment is needed as cost related to infrastructure is transferred from buying and configuration to renting resources from an infrastructure provider.

Lowering operational cost as the operational cost is dynamically allocated depending on service demand. Service providers do not need to allocate resources according to peak load which provides savings whenever service demand is low.

Highly scalable services because infrastructure providers pool high amount of resources and make them easily accessible. A service provider can easily expand geographically and handle surges of traffic thanks to infrastructure providers many data-centers which are located around the world.

Easy access to deployed service because services hosted the cloud are generally web-based, which is why they are easily accessible through a wide variety of artifacts (e.g., laptop and car).

Reducing business risk: Risks associated with infrastructure management such as hardware failure and staff training as well as knowledge. Infrastructure providers have staff who are highly trained in network and server maintenance, which result in better infrastructure then many businesses can afford to maintain.

However, cloud services are not a perfect service which is capable of replacing every company or organizational need. There are security risks of relying on a third-party, where sensitive information might leak. For example, how hard-drives are re-used or disposed of effects the likelihood of leaked data.

Cloud providers offer a wide variety of services so everyone can find something that fits their needs. The services vary in their environment configuration

Category	Description
Infrastructure-as-a-Service	Provides a highly configurable virtual environment and infrastructure
Platform-as-a-Service	Provides pre-configured environment and infrastructure with the option to configure them to a slight degree
Software-as-a-Service	Provides a locked environment and infrastructure which is fully managed by a third-party.

Table 2.3: Cloud Categories

and the amount of control which is available over the infrastructure. Users can choose to use a simple service such as Compute Engine, where they have complete control over the environment and infrastructure. On the other side, users who do not want to manage their infrastructure can use a service such as App Engine, which is a pre-configured environment, and focus solely on the development of their software.

Table 2.3 lists the three categories which cloud services fall under, and indicate the user's ability to affect the infrastructure. Infrastructure-as-a-Service (IaaS) provides the most flexibility, where users have the opportunity to configure most of their infrastructure. Software-as-a-service (SaaS), on the other hand, is the most rigid, where users are unable to influence the infrastructure.

IaaS services are highly configurable, where the user is essential renting an empty virtual machine which gives full control over the operating system, a significant degree of hardware configuration options in addition to a low degree or non-existent vendor lock-in. An advantage of IaaS compared to the other two services is the low amount of dependency on the cloud provider and their service enhancing features [44].

PaaS, on the other hand, is a pre-configured environment which is managed by the cloud provider. PaaS builds further on IaaS and is in many cases a pre-configured IaaS instance. PaaS restrict its users to a set of options provided by the cloud provider, which makes it easier for them to scale the instance. Users are, however, able to slightly change the configuration [45].

SaaS is a locked pre-configured environment which cannot be affected by the user. SaaS is, as the name implies, a fully functional service which the user leverage to achieve their goal. An example of a SaaS product is Gmail, which is fully managed and maintained by Google. The consequence of using SaaS service is the instant vendor lock-in, which provides complication whenever a user wants to move over to a new provider.

2.3.2 Services

The three services which we considered is IaaS (Compute Engine) and PaaS (App Engine and Cloud Functions). The three options are, theoretically, built on top of each other, which is why we will begin by introducing Compute Engine, then move on to App Engine and finally Cloud Functions.

2.3.3 Compute Engine

Compute Engine is highly configurable and scalable virtual machines. Compute Engine provides all necessities for a system, such as persistent storage and low network latency. The service provides an empty virtual machine with a highly configurable infrastructure to perform acceptably under any given circumstance. The instances can be created and managed through multiple methods, such as the online console, the gcloud CLI or a REST API.

2.3.4 App Engine

App Engine is one or more pre-configured Compute Engine instances which collaborate to form an application. Google Cloud Platform partially manages the instances and provides a list of run-time environments. The environments provide more predictable behavior, which makes it easier for Google to scale the application.

Google provides two different environments, a standard environment, and a flexible environment. The standard environment is a sandboxed and restricted environment to make it as easy as possible for Google to respond to sudden or extreme spikes of traffic. The flexible environment, on the other hand, allows for more configuration options and supports a broader range of runtimes compared to the standard environment.

2.3.5 Cloud Functions

Cloud Functions is a further advancement of the PaaS paradigm, with a much more restricted environment. Cloud Functions is a serverless execution environment, which means that the work of managing servers, configure software, updating extensions and patching the operating system. Further, the service itself encourages fast development and deployment of small features inside isolated environments.

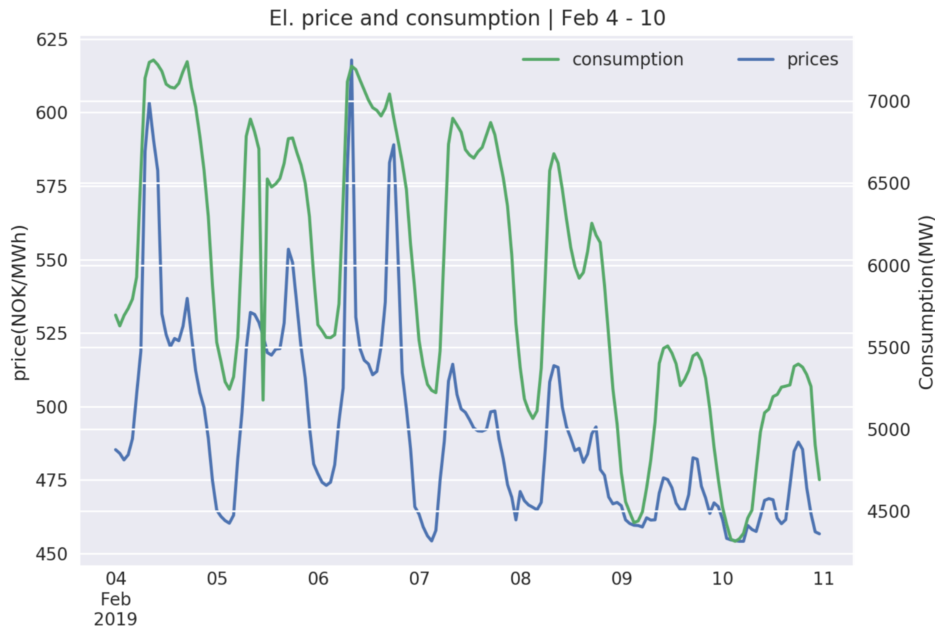


Figure 2.11: Correlation between price and consumption

2.4 Data

2.4.1 Power Market and Power Data

The power industry here in western Europe is a competitive industry in which market forces dictate the price of electricity and reduce the net cost through increased competition [39]. In other words, supply and demand dictate the value of available products on the power market.

Supply and demand play a central role in theoretical economics. In its purest form, the concept can be described roughly in the following terms: In a free market, the price of each commodity depends on the extent to which consumers demand it. If at a given set of cost, the demand for a good exceeds the available supply, then its price will rise, thus causing the demand to decrease. However, if the supply exceeds the demand, then the price will decrease, and demand will thereupon increase [16].

As visualized in figure 2.11, the electricity price varies throughout the day in correlation with electricity consumption. Most of the expenditure on electricity goes toward mornings, where everyone wakes up and eat breakfast.

We observe another peak during evening hours, which is after work when families gather for dinner after a long day. The valleys, on the other hand,

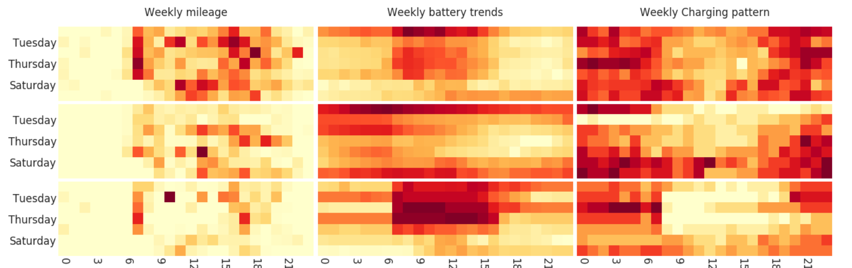


Figure 2.12: Weekly vehicle trends

occurs during night time, where most people sleep and is therefore incapable of actively consuming electricity through procedures such as cooking and entertainment.

The presented information describes the trends here in the Oslo area, which means that variations in patterns might vary depending on the location. However, we are confident that the trends will be nearly identical as presented.

Another significant trend of note is that night from Sunday to Monday is generally the most reasonable time for vehicle charging. Weekends have lower power rates during a week, which carries over to the next week to some degree. Appendix B.3 provide an extended window, which illustrate the phenomenon.

2.4.2 Vehicle Usage Data

Weekly driving trends provide valuable information for third-party systems, such as Cosinus, when assisting vehicle owners in their charging decisions and moving charging times to hours with less power consumption.

Most people have a predictable trend, which follows the power usage to some degree. Morning usage consists of traveling to work and driving the children to school or kindergarten, while evening usage consists of picking up the kids on the way home from work, with the occasional detour to the grocery. Additionally, late driving includes hobbies, sports, and fitness.

Figure 2.12 visualize the weekly trends recorded by the Cosinus system, with deep colors representing high frequency, while light colors represent low frequency. Each column in the figure represents one form of information. From left to right, we have mileage information, battery percentage throughout the week and charging trends. We further divide the figure into rows, with the top row representing all vehicles connected to the Cosinus system. The following two rows represent a vehicle, to provide a more intimate understanding of how usage varies between cars.

In the top left corner, we see that vehicles are usually standing still during nighttime, with most of the driving is occurring during the morning and evening hours as discussed earlier. Additionally, the usage stretches further into the night, which is caused by interests such as hobbies, sports, and fitness.

We observe the same patterns for the two vehicles presented in the figure. Car one (middle row), however, seems to have a consistent usage throughout the day, while car two (bottom row) has a more general usage, where the vehicle is most likely to stand still during working hours.

Current battery charge trends and charging trends provide additional meta information to understand the conditions which the vehicle operates under regularly. We see vehicle one have a larger interval between each charging cycle, compared to vehicle two, which is most likely due to one of two factors, or both. It can be due to battery capacity differences or traveling distance differences between the two. The most significant feature of note, however, is that vehicle two is most likely to have a high charge during the working hours. This state is most likely due to free charging during at the workplace or the absence of a charging station at home.

Charging patterns indicate that most EV owners plug-in their charging cable whenever the vehicle is either not in use, right away when they come home. We observe a high frequency of charging during night-time, which begins already at around 6 pm - 8 pm. We see the same trends for our two vehicles. Vehicle two, however, seem to have a mismatch between the battery charge trend and charging pattern trend. We believe the most likely cause is the instability we experienced between the Cosinus server and the external API [1] in addition to the pre-processing method used on the extracted dataset.

The trends presented highlights two crucial pieces of information. EV owners are prioritizing convenience rather than power cost when they decide on whether or not they should charge their vehicle. The other portion of data is that charging can be postponed to nighttime. We see in our dataset that the chance of nighttime use is on the lower end.

2.5 Vehicle Communication

This section introduces information regarding methods to communicate with Tesla and BMW. Tesla does not provide any official documentation to communicate with their service. Therefore, the presented information regarding Tesla where scraped from their official forum [28] and the unofficial documentation

[1] Application Programming Interface: A predefined communication protocol to interact with a service.

provided by Tim Dorr [42]. BMW, on the other hand, provides documentation to communicate with their service. The information regarding BMW is, therefore, a small snippet, with the entire documentation located in the appendix section.

2.5.1 Tesla

We achieved communication with Tesla vehicles by mimicking app requests. As such, the whole process begins with user authentication. The authentication process requires user credentials, in the form of an email and password combination as illustrated in the example below. Another alternative to the traditional authentication method is a token refresh procedure. Both methods result in a similar behavior from the API. The access token is then used for further communication following the bearer schema, where the token is present in the header.

```
1   # Login
    curl -X POST \
    -H "content-type: application/json" \
    -d '{"grant_type": "password", "client_id": "abc", "client_secret":
        "123", "email": "johndoe@company.com", "password": "password123
        "}' \
5   https://owner-api.teslamotors.com/oauth/token

    # Refresh token
    curl -X POST \
9   -H "content-type: application/json" \
    -d '{"grant_type": "refresh_token", "client_id": "abc", "client_secret":
        "123", "refresh_token": "<token>"}' \
    https://owner-api.teslamotors.com/oauth/token

13  {   # Response payload
        "access_token": "<token>",
        "token_type": "bearer",
        "expires_in": 3888000,
17      "refresh_token": "<token>",
        "created_at": 1538359034
    }
```

A valid access token is then used to retrieve information about a single vehicle, or list all cars connected to the user. Depending on the request, we receive either a list of JSON objects or a single object. These JSON objects contain

all information that is needed to collect data. As the example below illustrates, each car object has an id and vehicle id. In addition to identifiers, we received the VIN, name and enabled options for each car.

```
1   # GET url: /api/1/ vehicles /{ id }
    [{
            "id": 12345678901234567,
            "vin": "5YJSA11111111111",
5           "display_name": "Nikola 2.0",
            "option_codes": "MDLS,RENA,AF02...",
             ...
     }],
```

Car data is then accessible through GET requests to various endpoints when an access token and ID is retrieved. Tesla provides access to multiple types of information. However, the information we are interested in for the thesis is battery and vehicle information. Also, Tesla offers the option to retrieve the temperature around the car and its location at the time of the request, which is a complementary data source which is good to complement the battery data. The examples below illustrate responses for each type of data request.

```
    # GET url: /api/1/ vehicles /{ id }/ vehicle_data
    {
        "vin": "5YJSA11111111111",
4        ...,
        "drive_state": {
            "latitude": 33.111111,
            "longitude": −88.111111,
8            ...
        },
        "climate_state": {
            "inside_temp": null,
12          "outside_temp": null,
             ...
        },
        "charge_state": {
16          "battery_level": 64,
            "charging_state": "Disconnected",
             ...
        },
20      "vehicle_state": {
            "odometer": 33561.422505,
             ...
```

```
    },
    "gui_settings": {...},
    "vehicle_config": {...}
}
```

2.5.2 BMW

BMW provides an API intended for business use and differs significantly from Tesla. The most significant difference lies in the authorization procedure. Moreover, BMW requires a configuration step to enable interaction with its vehicles. As mentioned earlier, we provide the BMW documentation in the appendix section.

Data retrieval regarding BMW vehicles is a simple and straightforward process. The interaction begins by adding a VIN to a specific container, as illustrated below, which is BMW's method manage vehicles and data. The event triggers an access request which is sent to the vehicle owner in the form of an email. The user is then required to accept the request if they permit us to access data access regarding their vehicle.

```
curl -X POST \
-H <API_KEY> \
https://api.bmwgroup.com/otpclearance/api/thirdparty/v1/test/
    applications/containers/<CONTAINER_ID>/vehicles/<VIN>/
    clearances

# Successful post response
{
    "clearanceId": "11111111-1111-1111-1111-111111111111"
}

# Invalid VIN response
{
    "errorCode": "TP-101",
    "message": "No permission for specified VIN"
}
```

A clearance number identifies a vehicle and is used for further communication. Moreover, BMW uses the clearance number to notify the server regarding vehicle permissions. In other words, BMW informs us whenever the user approve, decline or revoke our permission to collect data. The notification arrives in the form of a GET request as

shown below, where the identifier and event description present this url: **/v1/clearances/<clearance_id>/<event_type>/<event_value>**. When permissions are approved, we can collect said information through a GET request.

```
curl -X GET \
-H <API_KEY> \
"https://api.bmwgroup.com/otpdatadelivery/api/thirdparty/v1/test/
    clearances/<CLEARANCE_ID>/telematicdata"
```

```
{
    "telematicKeyValues": [
        {
            "name": "bmwcardata_mileage",
            "timestamp": "Wed Feb 01 01:44:07 CET 2017",
            "unit": "km",
            "value": "98511"
        },
        ...
    ]
}
```

2.6 Terminology

2.6.1 CPU Scheduler

The CPU scheduler is an essential part of a modern operating system. It is central in the operating system's design and affects the overall performance of the system. The schedulers task is to maximize the systems CPU utilization by allowing many processes to run concurrently [40].

The scheduler determines which process to run whenever there are multiple runnable processes. CPU scheduling is important because it can have a significant effect on resource utilization and other performance parameters. There exist many CPU scheduling algorithms like First-Come-First-Server, Shortest-Job-Scheduling, Round Robin, etc. But due to many disadvantages, these are rarely used, except Round Robin scheduling in timesharing and real-time operating system [37].

A basic scheduling algorithm works by distributing process on two types of queues, a ready queue, and a waiting queue as illustrated by figure 2.13. The ready queue simulates a line of workloads which can be run and waits for their time to utilize system resources. The waiting queue, on the other hand,

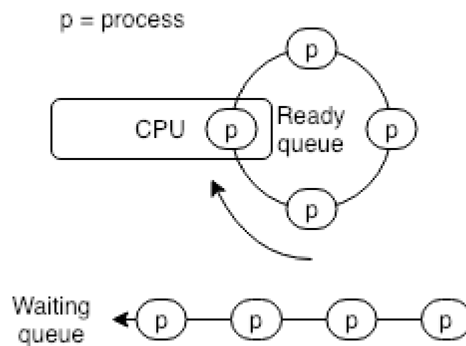


Figure 2.13: Simple CPU scheduler

is reserved for processes which are waiting for I/O input and cannot continue without it. The processes are moved from the waiting queue to the ready queue when they receive their respective I/O, while processes in the ready queue are transferred to the waiting queue if they need to wait for I/O to continue.

2.6.2 Process

All runnable software on the computer, including the operating system itself, is organized into many processes. A process is just an executing program, including the current values of the program counter, registers, and variables. In other words, a process is a set of instructions. Depending on the process, it may be made up of multiple threads which assist the process in accomplishing its goal [41].

2.6.3 Thread

A thread is a dispatchable unit of work. Threads are light-weight processes within a process and can be assumed to be a subset of a process [7]. A process operates inside an isolated memory space and can consist of multiple threads, as mentioned earlier. Threads, on the other hand, can share a memory space inside the parent process [5], which a significant advantage when it comes to co-operative problem-solving. Another significant difference between a thread and a process is the scheduler. Threads are managed by the parent process, compared to processes, which is controlled by the operating systems CPU scheduler [41].

2.6.4 Coroutine

The concept of coroutines is one of the oldest proposals of a general control abstraction [27, 29]. Researchers in fields such as artificial intelligence and concurrent programming explored coroutines widely during its first 20 years after introduction. However, most general purpose language designers disregarded this powerful control construct, partly attributed to the lack of a uniform view of the concept, which was never precisely defined [29].

Coroutines is a general control mechanism for multitasking. However, coroutines are a non-preemptive control structure, which means that coroutines provide concurrency but not parallelism [29]. The advantage, however, is that coroutines can pass control among themselves. Passing execution control without the need for system calls or blocking calls makes coroutine an excellent lightweight option to threads and processes. Coroutines are especially useful when applied to certain kinds of programming problems such as networking or distributed computation — for example, creating a processing pipeline [4].

2.6.5 User Experience

The term user experience refers to the usability of a technological product. User experience looks beyond the functional aspects of the product. User experience is a term used to describe the emotional characteristics of a product, in other words, user experience looks into the psychological aspects related to the product, to understand the user's feelings and design more positive interfaces [19].

There are a lot of variables which affect the user experience in technological products, from the hardware device itself to the software running on the machine. Everything from button placements, responsiveness, and user flow affect the overall user experience, which can result in either frustrated users who refuse to use the product further or happy users who preach the greatness of the product.

2.6.6 Native Applications

Native applications refer to applications developed using tools and software provided by the platform owner. Native applications target one single platform and are in general more difficult to develop and require a high level of experience and expertise compared to other types of applications. However, native apps have the possibility of providing the best user experience. Dedicated compilers, programming languages, and tools, for the targeted platform result in fast performance, consistent interface and full access to the

underlying hardware and data [20, 46]. The tight coupling between application and device provides the developers with everything needed to craft rich and responsive user interfaces.

2.6.7 Cross-platform Applications

Cross-platform applications refer to applications developed using tools or software provided by a third-party instead of the platform owner. Cross-platform applications can run on multiple platforms with the same codebase, which makes it an attractive option in comparison to native apps, which is developed for a single platform. Cross-platform applications strive to achieve native performance while running on as many platforms as possible, without limiting the capabilities of the application. [20, 46].

There are multiple variations of cross-platform applications, where development speed and applications speed varies significantly between the options. The purest form of a cross-platform app is web and hybrid apps, which is use web technology. The most significant disadvantage of web and hybrid apps, however, is the limited access and communication with the underlying device. Hybrid apps run inside an embedded native container, compared to web apps which are browser based. Finally, we have generated apps, which compiles the codebase into individual versions for each respective platform. The advantage of generated apps compared to web and hybrid apps are near-native performance [20, 46].

2.6.8 Technical Debt

Technical debt is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. The original description of the metaphor reads: "not quite right code which we postpone making it right." The metaphor is often used to describe things that stand in the way of deploying, selling or evolving a software system [25].

2.7 Summary

In this chapter, we presented an extensive amount of topics, from infrastructure related topics to more abstract topics such as security. Building a system that handles personal data requires much consideration to security, which we mitigate to some degree through the use of database access objects, input validation and a feature-rich back-end framework such as Django. Further,

we presented power data and aggregated driving data collected by the system, to improve our understanding of EV usage trends. Finally, we describe basic terminologies which we use in our main work.

Chapter 3

Related work

The widespread adoption of EVs has been a research topic and concert for quite some time. The vehicles introduce new user patterns and demands that could result in disadvantageous effects on the network.

Rahman and Shrestha [32] investigates the challenges associated with EV charging in residential areas as early as in 1993. Earlier research was optimistic and concluded that charging the batteries during off-peak hours would be sufficient to prevent instability and overload. However, as discussed by Rahman and Shrestha, off-peak charging will not be sufficient to prevent overloads from occurring. Early off-peak hours will spike up, with the possibility of exceeding the regular peaks experienced, which is why it would be necessary to stagger the charging process to mid or late off-peak hours. Further, they discuss how technological advances in battery and charging technology will be able to ensure faster charging, which bestows the ability to start charging at late off-peak hours. However, as they mention, technological advances in battery technology will increase battery capacity and increase the power demand even further.

Richardson et al. [36] investigate the problem through linear programming, to reduce and remove the need for historical data. To ensure that the power grid never exceeds its capabilities, he explores the possibility of active charging management. By continually lowering or increasing vehicles charging rate, he can ensure an acceptable result, while preventing network components from overloading. Through a standard approach of lowering and increasing the output for each car, he saw that those close to the transformers were favored, which resulted in several EVs not reaching a desirable charging state of 100%. To combat the issue, he introduced a weighted formula where vehicles with low battery charge state were favored and allocated a higher charging rate compared to cars with a high charge state, which resulted in a much more desirable outcome, with fully charged cars.

Similarly, Clement-Nyns et al. [8] investigate the problem through voltage manipulation in addition to bi-directional power flow through the use of plug-in hybrid electric vehicles. Clement-Nyns et al. apply dynamic programming and quadratic programming to maintain a fully utilized and stable power grid with the added possibility of having plug-in hybrid vehicles as a complementary power source. Further, using a bi-directional power flow to support the power grid, will reduce the risk of large voltage drops. The benefits are, however, minimal unless done during the moment of power peaks.

Rahman and Shrestha provided an accurate prediction for the future. Advances in battery and charging technology have increased battery capacity in addition to speeding up the charging time, which has resulted in more flexibility when it comes to charging time. Some vehicles can even drive multiple days between each charging cycle, which present the opportunity for more dynamic charging schedules and better prevention of sharp power peaks.

Further, Richardson et al. and Clement-Nyns et al. present collaborative charging in an area through continues power profiling and voltage manipulation. The most significant weakness of their approach, however, is the goal of reaching full charge for all vehicles connected to the grid. User patterns and demands vary considerably, which present more flexibility in the charging of vehicles.

EVs who require long charging cycles must begin their charging cycle during the early off-peak hours. Others, however, have the opportunity to charge during mid or late off-peak hours, which leave more capacity for the vehicles in need of the extra time and power to pull more voltage in the first hours of the charging cycle.

Profiling of EVs is required to enable charging with a staggered start. EV profiling does, however, require driving data for every vehicle in the area for full optimization, which is a challenge to retrieve. However, new opportunities have presented itself with the introduction of GDPR. Another driver for more accessible data is mobile applications. Mobile apps which give EV owners the ability to manipulate their car and see instant data related to the vehicle provides value to the car manufacturer and encourages the development of proper infrastructure to display instant vehicle data for their customers.

For better EV charging methods to be developed and better utilization of the power grid, we need driving data to profile as many vehicles as possible. A car data aggregation platform which can improve our understanding of EVs power demands and trends. The need for smarter charging grows with EVs popularity. Based on the need for more advanced and sophisticated charging methods, we developed an automatic vehicle data collection system, called Cosinus, which we present in the next chapter.

3.1 Summary

In this chapter, we introduce and discuss old and modern research related to the threats introduced with EVs. EVs add considerable load on the local power grid, which can result in power components overloading and voltage to drop. Rahman and Shrestha discussed that the problem might resolve itself, with better batteries and charging technology. However, better batteries would also result in increased power demand. Richardson et al. and Clement-Nyns et al. further discuss how we can mitigate the tremendous power demand. Through local power profiling, we monitor the power grid in close to real-time and adjust the voltage distribution to each household to prevent overload and instability.

Part II

Design & Development

Chapter 4

Cosinus

With our goal of making driving data more accessible and support future research in individual and collaborative charging, we designed and developed the Cosinus system. From related work, we found that current methods rely on power profiling and voltage manipulation to optimize power grid utilization. However, the current methods do not consider mileage trends, which is caused by the challenges associated with retrieving driving data. In this chapter, we will provide detailed information on how we designed and implemented a system for autonomous collection of driving and charging data. We discuss design decisions and provide a detailed explanation of the data collection process.

4.1 Overview

In chapter 2, we introduced our choice of libraries along with a variety of technologies, concepts, and challenges that have shaped the system to allow for a flexible, yet secure system. During this section, we provide discussions on the infrastructure, architecture, and database of the system.

4.1.1 Infrastructure

The infrastructure is one of the most crucial parts of a system. The server environment has a significant effect on the overall security and performance of the system. Most importantly, the infrastructure will have a considerable effect on the systems ability to move between cloud vendors and server environments. To ensure an acceptable level of technical debt, we evaluated three cloud services, Compute Engine, App Engine, and Cloud Functions to determine the best service for growth.

All three services offer a python supported server environment in addition to automatic resource allocation to ensure that the service is operational and responsive with a large volume of traffic. The difference between them lies mainly in the amount of control we have over the environment configuration and system architecture.

Cloud Functions offered the most exciting service, promoting full partitioning of the code-base into small independent features. The service takes full control over the server environment and ensures a highly scalable system that can quickly grow in features without affecting existing parts of the system. The most significant advantage, however, is the ability to change out existing parts of the system without worrying adjacent features. After experimenting with the service, we found it to be promising, yet constraint and unorganized. Each feature receives a separate deployment configuration, and the isolated aspect quickly results in repeated code, to reduce the overhead from communication across features.

Compute Engine offers the most flexibility when it comes to environment configuration. We had full control over the server environment and architecture, which allows for specific configuration to match our demands. We could, for example, configure the environment increase the memory of our instance, to allow for a massive amount of processor or thread spawning, or increase the number of virtual CPUs to accommodate a significant computational load. However, the service requires experience and knowledge for secure and full utilization. With full control of the environment entails vigilance over new and old attack vectors. The server environment is configured and maintained by the developer, and must, therefore, be manually secured and updated. An unattended server instance can result in malicious actors gaining control over the system, which can result in a tremendous amount of damage.

App Engine provides the middle ground between the two extremes. The service offers complete control to choose whichever system architecture we would like while managing the server environment. The service offers a variety of combinations for computing resource classes to match our needs as close as possible. On the other hand, we lose precise control over the hardware configuration, which can discourage specific tasks, such as GPU heavy computation. However, giving the cloud provider full control over the server environment means that parts of the security are maintained by a third-party, which is a benefit in this case. Google experience a tremendous amount of network traffic every day and has experience with malicious activity.

App Engine did not have official support for Python 3 when we began our work on the system, which is why we initially chose Compute Engine as our service of choice. Having complete control over the environment makes it easier

to develop a cloud provider independent system. The threshold to configure and maintain the environment is higher than we initially imagined. Server configuration consumed much of our time, which is why we, later on, moved over to App Engine when official support for Python 3 came to their standard environment. App Engine allowed for a quick and easy move without affecting the design of the system, which made it easy to decide to move.

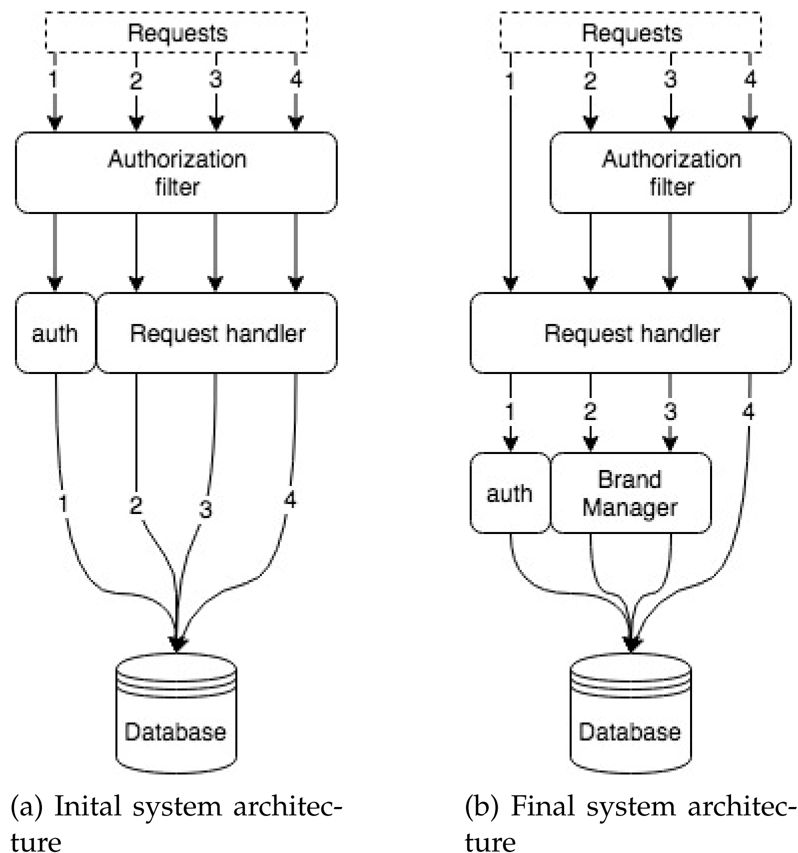
4.1.2 Architecture

The systems architecture is the most crucial part of the system that either allows or restricts the system from growing and adapting to changes in system requirements. We developed the system with Django and Django rest framework as the core dependencies of the system. We chose a feature-rich framework which contains more feature than needed to minimize the total amount of third-party library dependencies. Micro frameworks such as Flask are lightweight and a pleasure to use. However, the minimalistic nature of the framework results in a high number of extra dependencies to prevent the development of customized features for many aspects of the system. Many custom features can result in a high degree of technical debt and security flaws, which is why we wanted to rely on trusted open source options instead of creating a tailored solution.

Many things affect the systems ability to adapt to changes, which is why we designed and developed the system with a variety of combinations to gain insight into how the system is affected by individual architectural choices. We created a variety of architectural layouts, ranging from compact designs to more abstract and expanded designs. We began with a super compact version, with heavy request handlers that contained almost all the logic to full abstraction using an object-oriented approach.

A version of the compact design which relied on helper functions was quite favorable for quick changes in the data handling flow. However, adding new features required a full understanding of the flow, which was especially apparent when one of our fellow students attempted to add a new feature to the system. The full abstraction version was much easier to add features. However, changes to existing features required considerable more time to grasp how the data flowed through the system.

Figure 4.1 illustrate the initial design which we were satisfied with and the final design of the system. The initial design was compact and incorporated a few object-oriented components to assist with the introduction of new features. However, after re-evaluating the design, we felt that incorporating the brand manager into the request handlers added more logic into the request handlers



(a) Inital system architecture

(b) Final system architecture

Figure 4.1: Illustrations of the system architecture

than desirable. Similarly, we isolated the authentication endpoints in a separate part of the system were everything was contained. However, the request handler was too heavy in our opinion.

The final design has two fundamental changes to the design: a flexible authorization system and a complete abstraction of the brand manager. Creating an automatic hierarchical authorization system is advantageous. We define the lowest possible security clearance at a given endpoint, and every above automatically gain access to the resource. The automatic clearance system can confuse and result in restricted resources opening up, which is why we moved over to a verbose approach instead. The final authorization filter requires a clear definition of each security category to ensure a full understanding of who has access and who do not have access to a specific resource.

We also separated the core logic of the authorization provisioning logic from the authentication endpoints, to reduce code duplication caused by

the authorization filters verification process. The separation of authorization provisioning and verification logic will also allow for a contained environment which can be changed later on without digging into the request handlers.

For the brand managing logic, which is responsible for multiplexing between car Application Programming Interfaces (API), we wanted to make it easier for future features to interact with a vehicle. The current feature set if the system is quite small, which made is blind to specific future opportunities. To allow for more the incorporation of more advanced features without changing the request handlers, we decided to contain vehicle communication logic inside a specialized component. Providing a generic interface that multiple parts of the system can use is the best action to ensure future growth and would allow for more refined changes in the component.

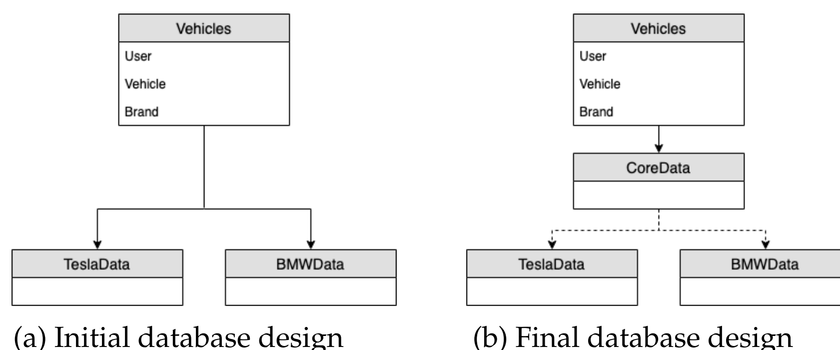
The most notable difference in the brand manager is the change from the declarative object initialization and function parameters to the use of static functions with flexible parameters. We decided to leverage pythons ability to group unknown parameters into a dictionary, to allow for more flexibility in the sent information between stages in the data flow. The destination is then responsible for confirming the presence of required information.

4.1.3 Database

Choosing the database was arguably the most difficult choice. The system generates a massive amount of I/O operations. There is network traffic to collect vehicle data and database interaction to store the data and retrieve vehicle authorization information every hour. We had to decide between a solution with low overhead or a high degree of data transaction validity.

A time-series database would be the best option. The collected vehicle data is time sensitive and informs the system of changes from one data point to another. Similarly, a document based database provides a flexible data structure allowing for the storage of new attributes with minimal changes to the code-base. We decided to use an SQL database, precisely because of the high degree of flexibility introduced by a document based database. The most critical data to store at this point is driving and charging data. The introduction of specific validation tests before storage to ensure the presence of our desirable attributes will create confusion for other developers. We decided that a meaningful design of a SQL database is the better option at this point. However, we did not rule out the possibility for future migration to a time-series database and designed the database for a smooth migration. Additionally, a SQL database allows for optimal utilization of Django and Django rest-framework.

Figure 4.2 illustrate the initial database schema, where we prioritized data



(a) Initial database design (b) Final database design

Figure 4.2: Illustrations of the database schema for vehicle data

storage based on origin, and the final database schema which allow for better integration with Django models. We initialized stored everything we could collect inside separate database tables. For Tesla, this meant four large tables, and for BMW, this meant one. We could have combined the Tesla tables into one, but there was just too many attributes and would result in over a hundred columns inside one table. After a few days, we took a closer look at the data to investigate what each piece informed on. Some of the data was not obvious, combined with two letter attribute names and we were in for a treat to understand what each piece of information contained.

The initial plan was to remove attributes that correlate to user settings and continue to use the schema. However, extracting the data resulted in a massive SQL statement, which is not maintainable in any sense. We incorporated object-oriented design on the database schema to allow for more optimal use of Data Access Objects (DAO) with the added benefit of automatic encryption and decryption of data. The final schema discarded many attributes and combined and linked the tables to create an object-oriented database schema. CoreData contains the most critical information, while more detailed information have a foreign key to its respective pair inside the CoreData table.

4.2 Authentication and Authorization

Authentication and authorization play an essential role in securing the system from malicious actors. Authentication confirms user identity and authorization grant access to system resources. Deciding between a variety of authentication and authorization options was the first task. Django has built-in authentication and authorization capabilities that can operate independently or collaborate. Django rest-framework provides an extension of Django's authentication and provides a variety of authentication opportunities. Finally, designing and

developing a custom authentication and authorization system would also be a viable option.

Django has separate tables for storing user passwords and user information. Another benefit of the authentication process is automatic password hashing, to protect user passwords. The accompanying interface is feature rich and intuitive in use, resulting in an easy decision to use the built-in authentication feature. The authorization feature, on the other hand, is session-based which is not desirable in the system. A session-based authorization has more overhead than JWTs and relies on information stored in the database. The data collection's performance relies on fast database write, which is why we want to avoid session-based authorization if possible.

Django rest-framework combines a plug-and-play mentality to the authorization process while leveraging Django's authentication feature. The available authorization options offered both session-based and JWT options. However, the authorization options relied on third-party libraries which had low activity. Additionally, the use of a third-party authentication feature abstracts the code from the process, which is desirable in many scenarios. However, the system needs a flexible hierarchy of security classifications to allow for growth in different directions, from regular users to businesses and researchers. Full control over the authorization process will give the system the ability to grow with minimal effort.

To provide a flexible level of protection and maintain an acceptable level of security, we decided to create a custom solution that follows security standards to ensure the protection of driving data from malicious actors. Creating the custom authorization combined with JWT would allow for quick request authorization with minimal overhead. We incorporated the PyJWT library into the system to ensure correct implementation of the standard. PyJWT is an auth0 sponsored library which is a company that focuses on system security and is, therefore, relatively a safe addition to our list of dependencies.

4.2.1 Authentication

Authentication is the first line of security where users have to prove their identity by supplying the system with a username and password combination. The system uses Django's authentication module which provides developers with an object-oriented interface to create new users and validate user credentials. The provided user objects allow for user categorization through the use of groups. Groups are metadata structures that describe additional user permission in addition to general permissions.

Figure 4.3 demonstrate the user creation procedure and the authentication

```python
from django.contrib.auth.models import User
from django.contrib.auth import authenticate

# Register user
User.objects.create_user(
    username=email,
    email=email,
    password=password,
    first_name=first_name,
    last_name=last_name
).save()

# Authenticate user
authenticate(username=username, password=password)
```

Figure 4.3: Creating a new user

procedure. The only required arguments for user creation is username and password. However, supplying the system with a name will allow for more personal interaction between the system and its users. We decided to use email as the username, resulting in two arguments needed for a successful registration. We left the name as an optional argument to leave the final choice to the client application.

A successful login or registration attempt triggers the production of two tokens for further interaction with the system. Figure 4.4 illustrate the procedure to create an access and refresh token. JWTs are vulnerable to man-in-the-middle attacks, which is why we decided to keep the access tokens life-span at a reasonably low time-span without affecting the user experience. The refresh token received a considerable higher life-span to allow for a single sign-on system where we expect the user to interact with the system at least once every six months. Additionally, the inclusion of the user's security level allows for quick provisioning of resources without confirming with the database.

New token pairs are retrieved by either resupplying user credentials or through an access refresh process. The access refresh process is a two-step verification process. The refresh tokens authenticity is validated through a JWT validation process. We create a new JWT with an identical payload to the supplied token and compare the signature. For valid refresh tokens, we added a second step with a database confirmation, to reject users were suspicious activity. Another benefit of the database validation is prevention against fake tokens that surpassed the first check.

```
def get_auth_pair(cls, user: User) -> dict:
    try:
        user.groups.get(name='researcher')
        security_level = 'researcher'
    except Group.DoesNotExist:
        security_level = 'superuser' if user.is_superuser or user.
            is_staff else 'user'

    general_token = {
        'usr': user.pk,
        'lvl': security_level,
        'iss': settings.TOKEN_ISSUER,
        'iat': datetime.now()
    }

    access_token = encode(
        payload={
            'exp': datetime.now() + timedelta(minutes=15),
            **general_token
        },
        key=settings.ACCESS_SECRET_KEY,
        algorithm=settings.JWT_HASHING_ALGORITHM
    )
    refresh_token = encode(
        payload={
            'exp': datetime.now() + timedelta(days=182),
            **general_token
        },
        key=settings.REFRESH_SECRET_KEY,
        algorithm=settings.JWT_HASHING_ALGORITHM
    )
```

Figure 4.4: Generate access and refresh token

4.2.2 Authorization

The authorization mechanism is the systems main security feature for regular use. The session-based option included with Django did not fit the system, while the plug-and-play options provided by Django rest-framework did not give a satisfying amount of control over the authentication process. Developing a new middleware would create a rigid solution with little to no hierarchy, which is why we developed two decorators to protect the systems resources. A regular decorator to protect typical endpoints and a decorator to protect system specific endpoints.

Decorators are a flexible mechanism that allows fine-tuned adjustment to provide a considerable amount of control over the resource provisioning process. The security groups combined with decorators allow for separate behavior depending on the permission level which results in better use of existing endpoints and request handlers. As such, by using decorators, we can provide different capabilities depending on the user's clearance level in addition to restricting a specific user from using a given resource.

Figure 4.5 show the implemented decorator used to protect regular endpoints. All authorized users have permission by default. For more vulnerable endpoints, a list of strings, specifying the permitted security levels must be provided to ensure adequate protection of the respective endpoint. Additionally, the authorization filter is automatically disabled when the system runs in debug mode, to allow developers to test features manually without the authorization process.

The authorized user's security level and unique identification in the system is appended to the request header, as seen on line 16 and 17. Injecting user meta information allows for variations in behavior, as mentioned earlier in addition to the identification of the requester for accurate data retrieval and storage operations.

4.3 Brand Manager

The brand manager is the heart of the system. The module manages communication with external resources and ensures data collection and vehicle connection. The module consists of four components; the core, Tesla component, BMW component, and a programming interface to ensure consistency in the brand's interface. The following sections provide a detailed description of each element, beginning with the programming interface, then the module core followed with the Tesla and BMW module.

```
def protected(arg):
    security_level = ['user', 'researcher', 'superuser']
    def __validate_request(request: Request):
        if settings.DEBUG:
            ...
            return
        if 'HTTP_AUTHORIZATION' not in request.META\
                or not request.META['HTTP_AUTHORIZATION'].
                    startswith('Bearer '):
            raise PermissionDenied
        token = request.META['HTTP_AUTHORIZATION'].replace('
            Bearer ', '')
        if not JWTManager.is_valid_access_token(token.encode()):
            raise PermissionDenied
        token = JWTManager.get_claims(token)
        if token['lvl'] not in security_level:
            raise PermissionDenied
        request.META['APPLICATION_USER'] = token['usr']
        request.META['APPLICATION_RESTRICTION'] = token['lvl']
    def __initiated_without_arg(*args, **kwargs):
        __validate_request(args[1])
        return arg(*args, **kwargs)
    def __initiated_with_arg(api_view):
        def __inner_wrapper(*args, **kwargs):
            __validate_request(args[1])
            return api_view(*args, **kwargs)
        return __inner_wrapper
    if callable(arg):
        return __initiated_without_arg
    elif isinstance(arg, list):
        security_level = arg
        return __initiated_with_arg
    else:
        raise TypeError(
            'Unexpected argument: expected List[str] received {}' % type
                (arg)
        )
```

Figure 4.5: Authorization filter

4.3.1 Service Interface

The service interface is an abstract base class (ABC) which dictates a minimum required implementation for classes which inherits it. There were problems which surfaced quite early on the implementation of the system. We decided to support Tesla and Nissan[1] at the beginning, and the difference between the protocol was huge.

The big difference in the confirmation process of user credentials resulted in a significant difference between the components interfaces. The differences affected the request handler, which handled the work in the beginning, resulting in individual tests to verify and multiplex correctly between brands.

The result was that the complexity of the system grew quicker than lines of code. To prevent the complexity from increasing further, we decided to refactor to achieve lightweight request handlers and more maintainable code.

ABC is a blueprint which enforces a minimum set of functions which must be implemented. Similarly to Java's interfaces, an error, or exception on this case, is raised whenever a class which inherits the ABC does not fulfill the requirements.

To achieve maintainable code with low complexity, we had to define the ABC with all the nuances in mind. After looking into Nissan and Tesla, and how evaluating the database models which would accompany the system, we decided on a simple and explicit interface.

As illustrated by figure 4.6, we kept the ABC simple. There are four fundamental tasks which are necessary for our system to function. Those are registering a vehicle to a user, retrieving information about a car, store the extracted data and renew access privileges whenever necessary to collect data.

The most significant obstacle and the part which introduces most of the complexity we encountered were the procedure to get authorization rights for vehicle data extraction. Which is why the function, connect, is crucial for the system to function correctly.

The most significant obstacle and the part which introduces most of the complexity we encountered were the procedure to get authorization rights for vehicle data extraction. Which is why the function, connect, is crucial for the system to function correctly. The connect signature allow rigidity in the sense that all services would have the same entry point.

We decided to separate the logic of retrieving and storing information related to a vehicle. The separation allows for easier debugging which is why we decided to separate the two, even though persistent storage is a crucial part of the data collection procedure. For example, when debugging, we sometimes

[1]We attempted to integrate Nissan in the beginning, however, as it were unstable and unpredictable, we decided to discard it as an potential supported brand at its current state.

```python
from abc import ABC, abstractmethod

class ServiceInterface(ABC):
    @abstractmethod
    def connect(self, **kwargs: dict) -> None:
        pass

    @abstractmethod
    def get_vehicle_state(self, **kwargs: dict) -> dict:
        pass

    @abstractmethod
    def save_data(self, **kwargs: dict) -> None:
        pass

    @abstractmethod
    def refresh_access(self, **kwargs: dict) -> dict:
        pass
```

Figure 4.6: Service interface

```
class BrandManager:
    SUPPORTED_BRANDS = ['TESLA', 'BMW']

    def connect(class, user_id: int, **kwargs: dict) -> None:
        service.connect(user_id = user_id, **kwargs)

    def monitor(class) -> None:
        for vehicle in vehicles:
            tokens = vehicle.tokens
            class.__monitor_helper(vehicle, tokens)

    def __monitor_helper(class, vehicle, tokens):
        data = service.get_vehicle_state(vehicle, tokens)
        service.save_data(**data)
```

Figure 4.7: Brand Manager Core

do not want to store the information. To disable storage the necessary step is to comment out the one line which invokes the function. However, if persistent storage had been a side-effect from retrieving data, we would be forced to comment out the necessary code for each service component which exists in the system (The components which handle communication with the external car APIs).

A vital signature which is essential to note is the **kwargs** argument which is present in each function. Kwargs stands for keyworded arguments. In other words, kwargs is a dictionary and is used to provide flexibility in which arguments that is necessary. By sending and extending the dictionary, each function and service is guaranteed to have the information it needs without implementing edge cases to ensure compatibility between services.

4.3.2 Brand Manager Core

The brand managers core acts as a glue and creates an abstraction to hide the service modules. The core's foremost responsibility is to direct request towards the correct service module. Second, it ensures a stable data collection flow.

Figure 4.7 provides pseudo-code which illustrate the content of the core in addition to describing its functionality. The class consists of three features, a list of supported brands, and two functions, connect and monitor.

The list of supported brands is meta information, to determine if a request should be rejected or accepted before entering the module. The brand

validation takes place in a serializer, which handles request payloads.

The serializer is a class which provides a model to our expected payload in addition to providing validation of the arguments. In this case, we use the serializer to reject invalid requests when it enters the request handler before moving on to other parts of the system. We will provide more information regarding the serializer later in the section regarding the format of requests intended for vehicle registration.

The connect feature is straightforward in logic as one might expect. The function retrieves the correct service module based on the brand keyword, located in kwargs, and invokes the service modules connect function. From there, the service module handles all the necessary work related to registering a vehicle to our system.

Each service has a different protocol for validating and connecting a vehicle. When it comes to BMW, the user is required to provide the cars VIN, while Tesla can retrieve all vehicles related to a Tesla user. This difference in protocol made it challenging to implement a general database logic unless we added tests to detect the difference, which is why we decided to isolate the persistent storage logic inside each respective module. The consequence is that a change in the database model requires a manual update of every single module.

The last feature is the monitorization of the vehicles. The feature is responsible for collecting and storing information related to every active car in the system. With active, we refer to vehicles which the system are permitted to monitor. Users can change their mind and remove the permit whenever they want. Also, providing the option to stop information collection of a vehicle increases the chance for users to register.

The monitor and monitor helper collaborates to gather information and refresh access privileges whenever necessary. The **get_vehicle_state** function triggers an HTTP request which might fail because of an expired token. Whenever an expired token is detected, the function refreshes the token and attempts again. However, successful requests are redirected to the **save_data** function, which stores the vehicles state with an timestamp in our database.

Storing Vehicle State

The amount of information retrieved depends on the brand. BMW and Tesla provide different meta-information about their cars. Tesla, for example, provides the settings of the vehicle in addition to its state. BMW, on the other hand, has a much more restricted approach, offering only the state of the car.

The differences resulted in a database table inheritance schema, to provide the ability to store the minimum necessary information, in addition to meta-information if possible. We decided on a core set of attributes, with the option

```python
class CoreData(models.Model):
    vehicle = models.ForeignKey(
        Vehicle, on_delete=models.CASCADE
    )
    odometer = models.DecimalField(
        max_digits=8, decimal_places=3
    )
    battery_level = models.IntegerField()
    charging_state = models.CharField(
        max_length=12, null=True, blank=True
    )
    ...

class TeslaData(CoreData):
    battery_heater = models.BooleanField(null=True, blank=True)
    conn_charge_cable = models.CharField(
        max_length=25, null=True, blank=True
    )
    ...
```

Figure 4.8: Data Django models

for each brand to extend the table if the opportunity presents itself.

Figure 4.8 provides the source code for CoreData and TeslaData. As presented in the figure, CoreData inherits from the base class Model, which provides an interface for data storage and retrieval. Further, TeslaData inherits from CoreData, which results in the creation of a new table with an extra attribute. More specifically, the TeslaData table has the listed attributes in addition to a foreign key pointing to the CoreData table.

Designing the database with inheritance provides one significant advantage compared to having separate tables, which is a unified table. CoreData provides a common connection point for all the data, resulting in easy retrieval of core information related to all brands. We can, in other words, retrieve all data in one query without any join operations.

4.3.3 Service Components

This section presents the BMW and Tesla service components. The explanation and straightforward, as it is a translation from the information provided in section 2.5 to working python code.

Resource address	Description	Methods
auth/login/	Generate new access and refresh tokens with an email and password combination	POST
auth/refresh/	Generate new access and refresh tokens with a refresh token	PUT
auth/registration	Register new user and generate access and refresh tokens	POST
api/connect/	Connect a new vehicle for monitoration	POST
api/data/	Retrieve information regarding vehicle(s) present in the database	GET

Table 4.1: Essential system resource addresses

The two components outline are similar because of the service interface we introduced in section 4.3.1, which is why we present and explain both through the use of the same pseudo code which presented in figure 4.9.

The service interface defines the available functions. Both **connect** and **get_vehicle_state** has a single task, create a URL and payload which is handed off to **api_request**.

Further, **save_data** retrieves all column names from the relevant table and create a new dictionary with keys that are present in both the retrieved information and list of column names. Afterward, we store the data with a single command as illustrated.

4.4 Interacting with the system

4.4.1 Resource addresses

There are a variety of resource addresses (endpoints) to perform actions, table 4.1 describe what we believe to be the essential endpoints of the system. During this chapter, we will explain the interface for the various resource addresses. As represented by the table, there are two main endpoints; auth and api. The auth endpoints are related to authorization and authentication, while api is a general endpoint for everything else, except automated tasks such as data gathering and database cleaning.

4.4.2 Query The Collected Data

This section introduces the **/api/data/** endpoint with information regarding data extraction and the implementation of the request handler. The **/api/data/**

```
from urllib.request import Request, urlopen
from urllib.parse import urlencode
from json import loads

class Service(ServiceInterface):
    __ENDPOINTS: dict = {
        'BASE': 'Service base endpoint',
        'AUTH': 'Service authentication endpoint',
        'DATA': 'Service data retrieval endpoint'
    }

    def __init__(self, **kwargs: dict):
        self.access_token = kwargs['access_token']

    def __api_request(self, req_type, url, payload, headers) -> dict:
        req = Request(
            url, data=payload, headers=headers, method=req_type
        )

        res = urlopen(req)
        return loads(res.read())

    def connect(self, **kwargs) -> dict:
        Retrieve access privileges for vehicle
        Retrieve the vehicles id
        Store retrieved information in database

    def get_vehicle_state(self, vehicle_id, **kwargs):
        Generate url with the vehicle id
        Retrieve vehicle state
        Pre-process data
        return data

    def save_data(self, **kwargs):
        Get column names from database model
        Get intersection between kwargs and column names
        Data(**intersection).save() # Store data
```

Figure 4.9: Service components pseudo code

endpoint serves two purposes depending on the user privilege. For general users, the endpoint helps to retrieve information to comply with GDPR. On the other hand, researchers can retrieve the data to support their research. Briefly explained, researchers have the capability of extracting all the data present in the database while regular users are restricted to information related to them.

The current implementation has a few restrictions to reduce the amount of data and information. Additionally, we decided to restrict the amount of information regular users are capable of extracting until we confirm the systems security capabilities. Information such as GPS location and the temperature, which are sensitive information informing the reader of the vehicle location is only available for researchers. Additionally, each desirable attribute has to be requested explicitly, to reduce the amount of data traveling from our server as much as possible.

Table 4.2 describe a list of optional arguments which are used to interact with the endpoint. The available parameters provide flexibility in the type of data retrieved, to potentially reduce the amount of necessary pre-processing required of the data. Parameters such as created__<lt|gt> and battery_level__<lt|gt> provides the ability to reduce the scope of the queried data, while fields are used to declare which fields the user desires. For users who want a dump of all existing attributes, we provided the '*' symbol as a shortcut, removing the need to mention each end every field explicitly.

Further, all request requires an authorization header following the bearer schema, which is illustrated in figure 4.10.

4.4.3 Connect vehicles to the system

The method to connect a vehicle to the system depends on the brand in question. BMW connection relies on the VIN while Tesla requires access to the users Tesla account credentials. To accommodate both through the same endpoint, we decided to separate the two depending on a single argument, brand, which validate the remaining request parameters.

BMW, which is VIN based, makes a simple validation of the VIN argument and forwards the identification number to BMW afterward. Tesla, on the other hand, requires a client-side authentication toward the Tesla endpoint, to retrieve a refresh token. The refresh token can then be sent with the refresh_token argument, which is forwarded and validated by Tesla to retrieve a fresh authorization pair. Additionally, the serializer with validates request parameters become more manageable, because of the separation in logic.

Figure 4.10 provides an example of an attempt to register a Tesla vehicle to the system. All communication with the system requires an authorization

Argument name	Argument description
sort_by	Sort the result on the timestamp either in ascending or descending order. The default order is ascending.
created__gt	Retrieve data which were collected after the provided date. The date follows the standard ISO format (YYYY.MM.DD).
created__lt	Retrieve data which were collected before the provided date. The date follows the standard ISO format (YYYY.MM.DD).
vehicle_state	Retrieve data which are connected to a vehicle of a certain state. Vehicles of all states are returned by default. However, active (A), pending (P) and deactivated (D) vehicles can be retrieved separately.
vehicle_brand	Retrieve data related to a specific brand.
battery_level__gt	Retrieve only data instances which are greater then the provided value.
battery_level__lt	Retrieve only data instances which are lower then the provided value
charging_state	Retrieve only data instances where the charging state is either Disconnected, Charging, Stopped, Complete or Error.
fields	Append the provided fields to the retrieved dataset. Available fields are est_battery_range and charging_state. In addition, the fields longitude, latitude, inside_temp and outside_temp is available for researchers and admin users.

Table 4.2: Available arguments for data retrieval

```
curl -X POST \
-H "Content-Type: application/json" \
-H "Authorization: Bearer xxxx.yyyy.zzzz" \
-d '{"brand": "Tesla", "refresh_token": "token"}' \
http://127.0.0.1:8000/api/connect/
```

Figure 4.10: Connect curl command

header, following the bearer schema, as illustrated. Registering a BMW is similar to the illustration, where the difference lies with a VIN argument instead of the refresh_token which is present in the figure.

4.4.4 Interaction with the auth endpoints

The sections above introduced data extraction and connecting vehicles to the system. Both endpoints require the user to be authenticated. This section describes how users are authenticated and authorized to interact with the system. Each of the following sections provides examples and describe restrictions related to the required and optional arguments.

Each endpoint responds similarly, with an access_token and refresh_token when successful and a JSON which describes why an argument is invalidated. The access token has a life-span of 15 minutes and is required to further interact with the system. The refresh token, on the other hand, has a life-span of six months and is used to retrieve a new authorization pair, through the refresh endpoint described later in this section.

Registration

The registration endpoint has two required arguments (email and password) and an optional argument (name). All three parameters are validated according to simple rules, to enforce a minimum level of security for the user.

Email validation confirms the correctness of the value while the password validation ensures a minimum strength level for all passwords. All passwords must be at least eight characters long and contain at least one lower case character, one upper case character, and a numeric character. A valid email, on the other hand, must consist of two parts, a local part and a domain part, separated by the symbol '@.' Finally, the optional argument, name, must consist of at least two parts separated by a space ('␣'). The name parts are reasonably flexible, where everything is allowed as long as there are some alphanumeric characters separated by a space.

Figure 4.11 demonstrate a successful registration attempt, where the input conforms to the restrictions mentioned above. A name has the capability of consisting of multiple parts, as mentioned earlier. When that occurs, the system extracts the last part as the last name while everything else handled as the first name.

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"email":"example@mail.com", "password":"greatPassw0rd", "
    name": "John Doe"}' \
http://127.0.0.1:8000/auth/registration/
```

Figure 4.11: Registration curl command

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"email":"example@mail.com", "password":"greatPassw0rd"}' \
http://127.0.0.1:8000/auth/login/
```

Figure 4.12: Login curl command

Login

The login procedure is similar to registration and adopts the same restrictions. Authentication requires an email and password combination, as illustrated by figure 4.12, where passwords must be at least eight characters long and contain at least one lower case character, one lower case character, and a numeric character. Additionally, emails must consist of a local part and a domain part, separated by the '@' symbol.

Refresh Access

Retrieving new authorization pairs are achieved by sending a valid and unused refresh token as demonstrated in figure 4.13. As described in section 4.2.1, refresh tokens authenticity is validated in two steps, which is why there are three requirements for generating new tokens. (1) The tokens must contain an authentic signature. (2) The token cannot be expired and (3) The token must be present in the database of the system.

```
curl -X PUT \
-H "Content-Type: application/json" \
-d '{"refresh_token":"xxxx.yyyy.zzzz"}' \
http://127.0.0.1:8000/auth/refresh/
```

Figure 4.13: Refresh access curl command

4.5 Summary

In summary, we present and discuss the systems architecture. There are many nuances which affect the systems ability to grow, and many choices which introduce a significant amount of technical debt. E.g., using Cloud Functions to partition and isolate the code-base based on the feature will allow the system to grow independent of existing code. However, the aggressive partitioning of the code-base results in repeated code and vendor lock-in, introducing technical debt which is troublesome to relieve at a later point in time. On the other hand, collecting the entire code-base in one place, e.g., inside request handlers, will prevent quick changes to fix or change a feature. Creating a system, therefore, requires consideration to different abstraction methods, to allow the system to grow without introducing bugs.

However, abstraction is not the only thing to consider when we consider flexible design. State management plays an essential role in how tightly coupled each piece of the system is connected. Stateful objects require a good understanding of the data flow throughout the system, while stateless objects result in more data passing between objects than necessary. A stateless design, however, introduces more flexibility and allow the system to change without directly changing the interface, which is not always possible with a stateful design.

Additionally, we developed the data gathering process as an assembly line with four stations. (1) Retrieve vehicles from the database, (2) map the correct service component to each vehicle, (3) fetch the data and (4) store the data. The assembly line design introduces a few limitations which we will discuss more in the next chapter. The advantage of the assembly line design, however, is the ability to change each stations procedure without affecting the other stations.

Finally, we introduce the interface for a subset of endpoints, giving developers a good indication over how to interact with the system. Most importantly is the authorization header. The interface requires the presence of a JWT following the bearer schema.

Chapter 5

Monitor Optimization

The data collection process is the heart of the system. The monitor is responsible for retrieving and storing driving and charging data, to make it accessible for EV owners and researchers. In chapter 4.3, we described the implementation of the arguably most important part of the system. The brand manager, which handle everything related to external communication with the vehicles and car manufacturer.

The presented information regarding the monitor illustrates an assembly like design to allow for safe modification of various parts of the process. This chapter provides more in-depth information on the monitor and the limitations associated with the implementation presented. We begin with an overview of the monitor and its limitations and present alternative methods to improve the data collection process.

5.1 Cron Jobs

The data collection process relies on the cron service as a trigger to begin the collection procedure. The cron service is a service provided by Google which lives inside the same environment as the system. The cron service allows for the executed of tasks at regular intervals.

Tasks trigger a specific feature through an HTTP request which has a maximum execution time of 60 minutes. The cron service considers a task as failed if there are no replies or if the system returns a response code outside the range of 200-299 [1]. In other words, the system has a time limit of 60 minutes to retrieve and store data for all active vehicles registered.

The system is currently averaging around 2.5 seconds to retrieve and store data for the five vehicles which are present in the system as visualized by figure 5.1. With the current performance of the system, we are capable of handling

▸	2019-01-16 11:25:00.891 CET	GET	200	189 B	12.4 s	AppEngine…	/tasks/monitor
▸	2019-01-16 10:25:00.566 CET	GET	200	189 B	14.2 s	AppEngine…	/tasks/monitor
▸	2019-01-16 09:25:00.362 CET	GET	200	189 B	16.1 s	AppEngine…	/tasks/monitor
▸	2019-01-16 08:25:00.332 CET	GET	200	189 B	12.8 s	AppEngine…	/tasks/monitor
▸	2019-01-16 07:25:00.347 CET	GET	200	189 B	12.9 s	AppEngine…	/tasks/monitor
▸	2019-01-16 06:25:00.821 CET	GET	200	189 B	15.4 s	AppEngine…	/tasks/monitor
▸	2019-01-16 05:25:00.989 CET	GET	200	189 B	13.7 s	AppEngine…	/tasks/monitor
▸	2019-01-16 04:25:00.593 CET	GET	200	189 B	11.4 s	AppEngine…	/tasks/monitor
▸	2019-01-16 03:25:00.484 CET	GET	200	189 B	15.1 s	AppEngine…	/tasks/monitor

Figure 5.1: Screen grab of monitor execution times

1440 vehicles, best case scenario. One thousand four hundred forty cars are only 0.73% of the registered EVs here in Norway as of March 2019[1].

5.2 Current Data Collection Design

Section 4.3.2 described the mechanics behind the collection process at this point. Vehicle data is collected chronologically, from the oldest to the newest vehicle in the system in a method similar to an assembly line. The process requires four steps to achieve its goal, beginning with (1) the retrieval of all vehicles from the database. Afterward, the monitor (2) locate the correct service interface to interact with the vehicle. When the correct service component is retrieved, it (3) attempts to retrieve the current state of the vehicles. Finally, the monitor (4) store the retrieved information.

There are two critical problems with the current data collection design. The first and most crucial problem is that the data is retrieved chronologically. The retrieval process is an I/O task, which means that the system is idle while waiting for the data. The result is, as presented earlier, where the system is incapable of handling any significant number of EVs.

The second problem is associated with the final step of the process, the storage process. The current design establishes a new database connection, send the data over for persistent storage and cuts the connection for every single vehicle. In other words, there is a significant amount of overhead associated with the current data storage process. It is, however, possible to mitigate the problem through by accumulating the collected data and write to the database in one massive operation. The result is less network between the server and

[1] As of 29 of March 2019, SSB reported 195 351 registered electric vehicles [34]

header in addition to a reduced amount of bytes sent[2].

The first problem is not quite as simple to solve. Retrieving data for multiple vehicles at once is outside our capabilities, and some form of concurrency or parallelization strategy is therefore required to increase the capabilities of the system.

A variety of strategies exist to provide a system with concurrency or parallelization abilities. The different strategies range from local methods like multiprocessing to distributed methods such as clusters. To maintain an acceptable level of flexibility and maintainability in the system, we will make use of local approaches such as multiprocessing, multithreading, and coroutines. The reason is that the system is still young and the challenges associated with maintaining the system is not clear. Local optimization of the data retrieval process will provide more time to understand the challenges associated with maintaining the system while the user-base grows.

5.3 Benchmarking

We will benchmark the different optimization methods inside a familiar environment with a constant amount of computing resources. The optimal environment for the benchmarking is inside a similar environment as the server. However, the system runs inside an environment which automatically increases and decreases its computational resources based on load, which can result in a misrepresentation of the capabilities of the optimization methods for our use-case. Additionally, a significant amount of memory is required to run the tests due to the nature of how we benchmark the performance, which would result in huge costs associated with the experiment. We, therefore, decided to run the experiment inside a familiar environment with a constant amount of computing resources.

The experiments will be conducted on our local laptop, which is a MacBook Pro early 2015 model, with a 2,7 GHz Intel Core i5 with four cores, 8 GB of memory and Intel Iris Graphics 6100 1536 MB graphic card. All programs will be shut down during the experiment to ensure little to no interference with the results from external applications.

We perform two types of tests to gain insight into the optimization strategies benefits for our use-case. An I/O bound test in the form of busy waiting and a computational bound test in the form of finding prime numbers. The I/O bound test will provide a good representation of the data retrieval procedure, with a significant amount of idle time. The computational bound test, on the

[2]Reduced amount of headers sent between the two nodes to establish a connection

Figure 5.2: Performance of various test cases

other hand, will provide an excellent contrast to the I/O bound test, where the CPU is continuously working.

We initially wanted to include a third test, which provides an almost identical reproduction of the data retrieval procedure. The test case required a high amount of HTTP GET requests towards a remote endpoint, which created a few complications. We, therefore, decided to use the busy waiting method instead as our primary indicator instead, because it creates a similar effect to the CPUs waiting and ready queue.

Figure 5.2 visualize an initial benchmark of our test cases and show why we use a combination of busy waiting and finding prime numbers. The busy waiting test produces approximately constant output every time while the computational test varies in the elapsed time. Busy waiting produced an average of 1.29 seconds while the computational load provided an average of 0.85 seconds.

To represent a similar scenario to the data retrieval process, we decided to run each test a hundred times. We can then observe the performance of each strategy based on the elapsed time each strategy used to finish the test. Additionally, performing the same operation many times will result in a similar scenario as our monitorization process.

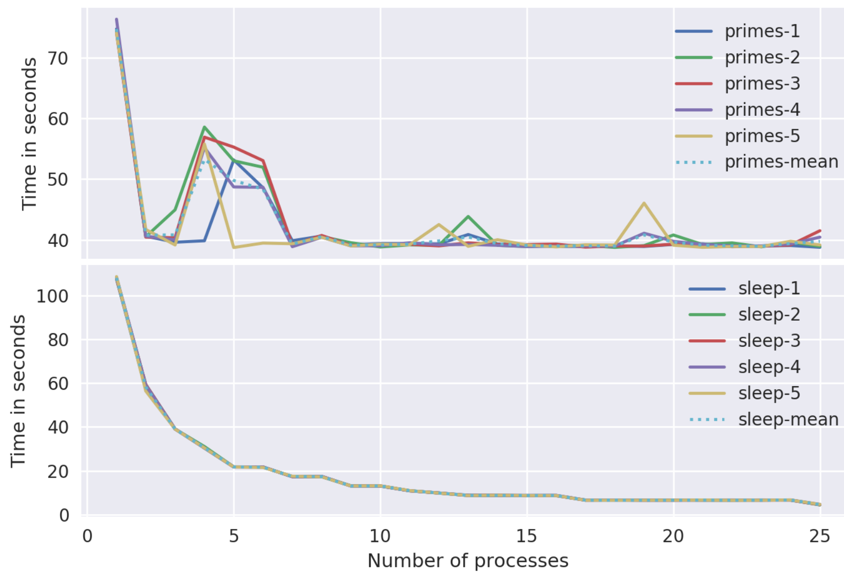


Figure 5.3: Multiprocessing performance

5.4 Approaches

This section introduces the available optimization strategies to increase the number of vehicles the system is capable of supporting. We begin by describing multiprocessing and the results achieved with the strategy. Then, introduce multithreading and coroutines in a similar style and finish with a comparison between the three approaches.

5.4.1 Multiprocessing

Multiprocessing is a parallelization strategy where a new process is spawned to parallelize work. Multiprocessing is the procedure of creating new processes and assigning them a specific workload. Processes are handled by the operating systems CPU scheduler, as mentioned earlier.

Figure 5.3 illustrate how the tests were affected by the number of processes we used, while table 5.1 show the first results of the I/O bound test. As shown, employing more than one process provides a performance benefit. Using many processes for our busy waiting test resulted in a growing waiting queue without obstructing the CPU in any significant way. We can see that the total elapsed time continues to decrease as the number of processes grows.

Nr. of Processes	run 1	run 2	run 3	run 4	run 5	mean
1	107.532	107.584	107.753	108.323	108.521	107.943
2	57.1807	59.2712	59.2903	59.0499	56.3454	58.2275
3	39.0480	39.0234	39.0441	39.0117	39.0023	39.0259
4	30.3512	31.0456	30.3994	30.3750	30.4232	30.5189
5	21.7117	21.7905	21.6860	21.7653	21.7185	21.7344
6	21.7066	21.7004	21.7006	21.7001	21.5014	21.6618

Table 5.1: Multiprocessing I/O test results

Our computational bound test, on the other hand, have an evident improvement in the beginning, which levels off quickly. More processes for a computationally heavy process result in a more massive ready queue, where the wait time between each time a process uses the CPU increase. Spawning more processes then the number of CPU cores introduce overhead which eliminate the advantage of parallelization.

For both tests, we see a performance boost of approximately 45% from one to two processes. The difference in performance then levels off for our computational test while the I/O bounded test continue to receive a performance boost that decreases slowly.

5.4.2 Multithreading

Multithreading is a parallelization strategy similar to multiprocessing. Multithreading initializes and assign specific workload to threads in an attempt to reduce the overall elapsed time and is commonly used to split a workload into smaller pieces.

Figure 5.4 illustrate how the tests were affected by the number of threads we used, while table 5.2 show the first results of the I/O bound test. As shown, employing more than one thread provides a performance benefit, especially to I/O bound operations. Using many threads for our busy waiting test resulted in a similar effect to the multiprocessing strategy.

On the other hand, the computational bound test provides little to no improvement with the increase of threads. The result is most likely due to a non-preemptive nature of the thread scheduler, where the threads are allowed to run until the finish. The best use of threads would be to split up the prime number search space into smaller pieces and assign the search space to each thread, instead of having each thread search the entire number space as we did in our test here.

As visualized in the results, the computational bound test has little to no

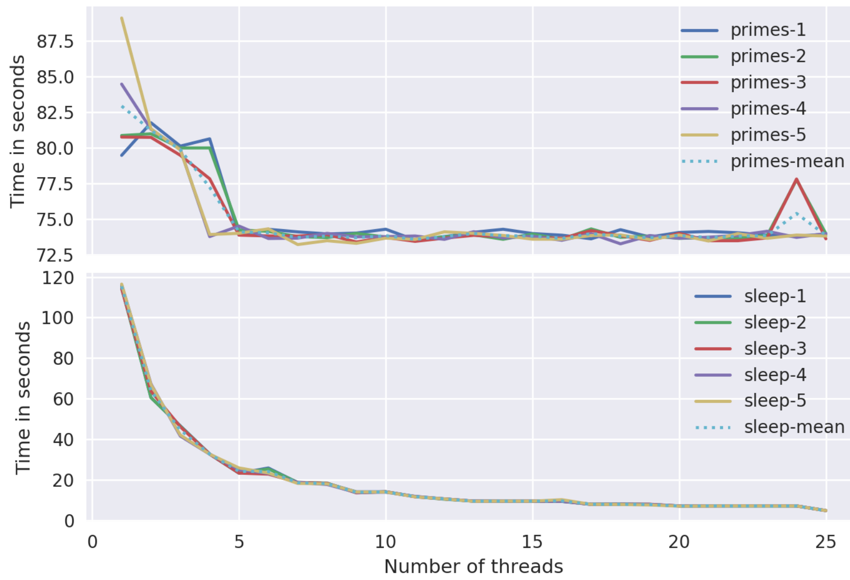


Figure 5.4: Threading performance

Nr. of Threads	run 1	run 2	run 3	run 4	run 5	mean
1	115.442	114.594	114.183	116.199	116.502	115.384
2	60.4978	60.4673	63.6874	67.2007	66.8974	63.7501
3	46.5573	46.4054	46.0974	41.5111	42.0293	44.5201
4	32.6949	32.6088	32.5944	32.5793	32.5900	32.6135
5	23.2082	23.2568	23.2310	24.9441	25.7813	24.0843
6	25.7665	25.2915	22.7485	23.0459	23.2834	24.0272

Table 5.2: Multithreading I/O test results

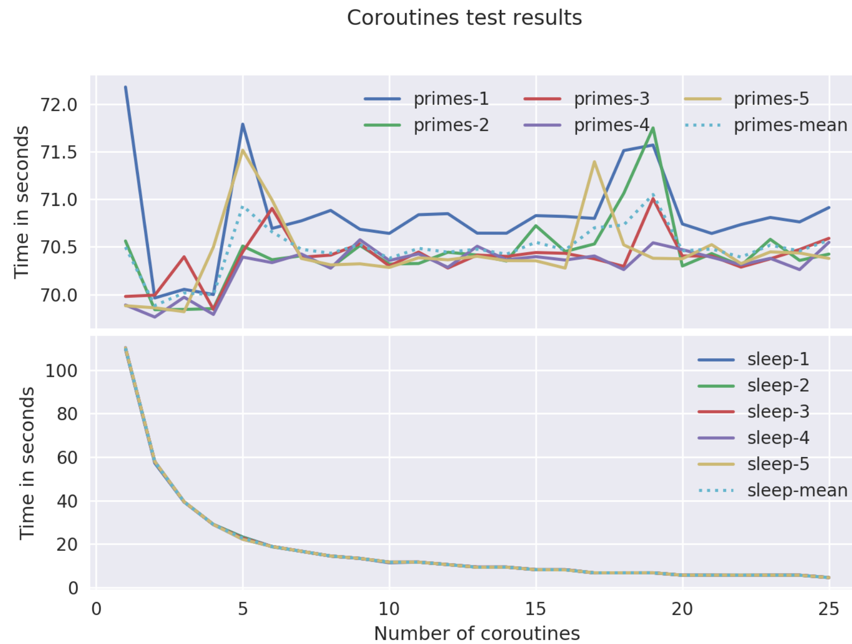


Figure 5.5: Coroutines performance

correlation with the increase of threads. The I/O bound test, on the other hand, experiences a 44%, 30% and 26% initial performance gain which continues to level off with the increase in threads.

5.4.3 Coroutines

Coroutines are generators which can be used for either asynchronous execution of tasks or cooperative multitasking. One of the advantages of coroutines is that they can as a producer and consumer where one coroutine is responsible for gathering while another is responsible for storing.

Figure 5.5 illustrate how the different tests were affected by the number of coroutines we used, while table 5.3 show the first results of the I/O bound test. As shown, employing coroutines provides a similar effect to the use of threads. Increasing the number of coroutines provides a steady and stable stream of results, especially for our I/O bound test.

On the other hand, the computational test provides a similar effect as the threading performance, where we see little to no performance with the increase of coroutines. The results offer near identical numbers throughout the test because of the non-preemptive nature of coroutines, where it is up to the programmer to yield.

Nr. of Coroutines	run 1	run 2	run 3	run 4	run 5	mean
1	109.694	109.932	110.157	110.358	110.229	110.074
2	57.1813	57.7926	57.8345	57.9279	57.9162	57.7305
3	39.2756	39.3874	39.3894	39.3697	39.4079	39.3660
4	28.9191	28.9457	28.8639	28.9100	28.9314	28.9140
5	23.1643	23.0922	22.6156	22.1225	22.0981	22.6185
6	18.8025	18.6561	18.8209	18.7957	18.8137	18.7778

Table 5.3: Coroutines I/O test results

5.4.4 Grid Computing

Grid computing combines multiple computers to attain a common goal, to solve a single goal [14], which is the collection of vehicle states in our case. Grid computing can be combined with the parallelization strategies mentioned above to increase the scalability of the system further.

Grid computing is best used with heavy computational tasks because of its distributed nature, where computers might be located at separate locations. Coordinating the different machines can also become a complex task, especially if the flow of information moves bidirectionally.

5.5 Conclusion

The monitor consists purely of I/O operations and is heavily dependent on HTTP requests and database communication to achieve its goal. We will, therefore, take a closer look at the busy waiting results, as those are the most interesting for our problem.

As visualized in figure 5.6 and table 5.4, we can see that, from an elapsed time standpoint, there is little difference between the three strategies. However, we can see that there is one feature from one of our approach, which is not present in the others. We can see that coroutines are more stable in their performance compared to the other two.

Both coroutines and multiprocessing produce near identical results for each test run. However, the performance advantages seen is most likely due to the ability to pass control without system calls or blocking calls, resulting in less overhead, especially in an I/O task such as our data collector, where switching occurs frequently. Another peculiarity observed is the minimum to non-existing advantage from using four threads or processes to five. We observe a similar effect for seven to eight and nine to ten threads or processes.

After evaluating the results and traits of the three control structures, we saw

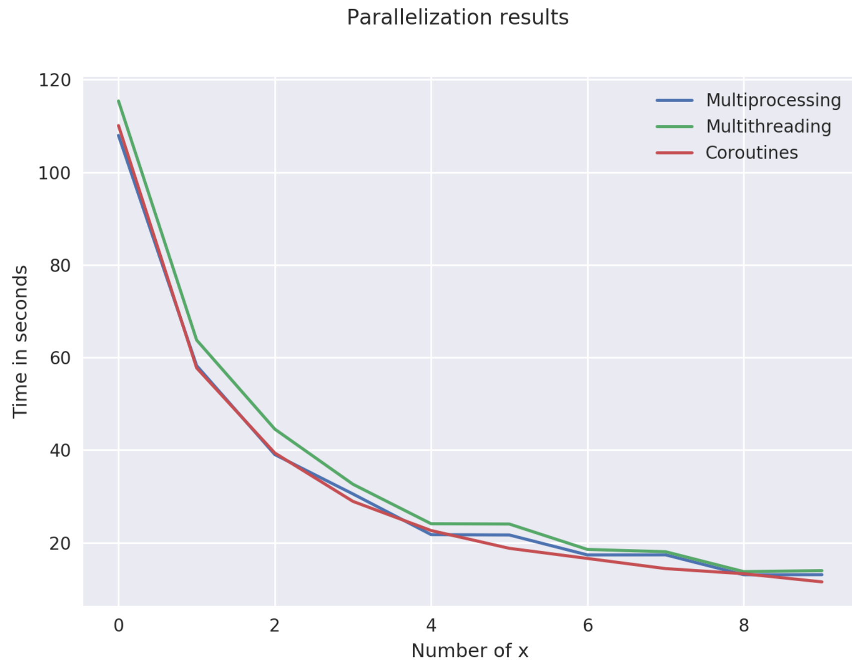


Figure 5.6: Parallelization test results

Nr. of X	Multiprocessing	Multithreading	Coroutines
1	107.943	115.384	110.074
2	58.2275	63.7501	57.7305
3	39.0259	44.5201	39.3660
4	30.5189	32.6135	28.9140
5	21.7344	24.0843	22.6185
6	21.6618	24.0272	18.7778

Table 5.4: Accumulated I/O test results

coroutines as an excellent alternative for our problem. The ability to pass control among themselves is an advantage in the future when the system handles thousands of vehicles. Additionally, we can optimize the database interaction procedure, by extending the pipeline with a bulk write operation.

We were unable to conduct our experiment inside a similar environment as the server, as mentioned earlier, which is why the test result might not reflect the actual performance of the server. However, we believe the trends displayed during the experiment will most likely be similar in both environments, with a tweak to the elapsed time and maximum number possible for either processes, threads or coroutines. Both processes and threads require a higher amount of memory compared to coroutines, which means that the amount used during our test does not apply to the current server configuration.

5.6 Summary

The assembly line design limits the systems ability to support a large number of vehicles. In this chapter, we introduced and discussed a variety of local parallelization techniques to improve the speed of the data gathering station of the assembly line. We created an imitation of the data gathering process using the multiprocessing, multithreading, and coroutines, to simulate the real scenario as accurate as possible. We saw that the different approaches perform nearly identical for our use-case. However, the overhead advantage of coroutines resulted in an advantage compared to the use of processes and threads.

Chapter 6

Tailored Charging Schedules

We designed and developed the Cosinus system to make driving data more accessible and to aid future research into individual and collaborative charging. To encourage EV owners to share their driving data, we have to provide something in return. This chapter takes a closer look at the collected data to present data contributors with general information based on their driving data.

Section 2.4 introduce aggregated information extracted the historical data. Displaying the aggregated information can bring awareness to the EV owners trends. However, the real value lies with combined data sources and finding hidden information in the dataset. During this chapter, we present a simple charging plan to help EV owners to determine when they should charge. Additionally, to ensure some degree of accuracy, we analyzed the driving data find the maximum driving range. Combining mileage trends and driving range estimates will result in a good indication for when to charge the EV in regards to power cost. Complementary figures and source code for the information presented during this chapter are in Appendix B.

6.1 Charging Plan

To assist EV owners in their daily charging decisions and alleviate some of the stress introduced by EVs, we need a charging plan or charging schedule that determine charging times based on the EV owners mileage trends and the areas power trends. By combining the power and driving data presented in section 2.4, we can create a personalized charging schedule for each vehicle.

Figure 6.1 present the result from combining a specific vehicle's data with the areas power trends. For simplicity, we chose vehicle one as a reference. The patterns will differ slightly depending on the car. However, the information presented applies to all EV cars. The most significant difference between

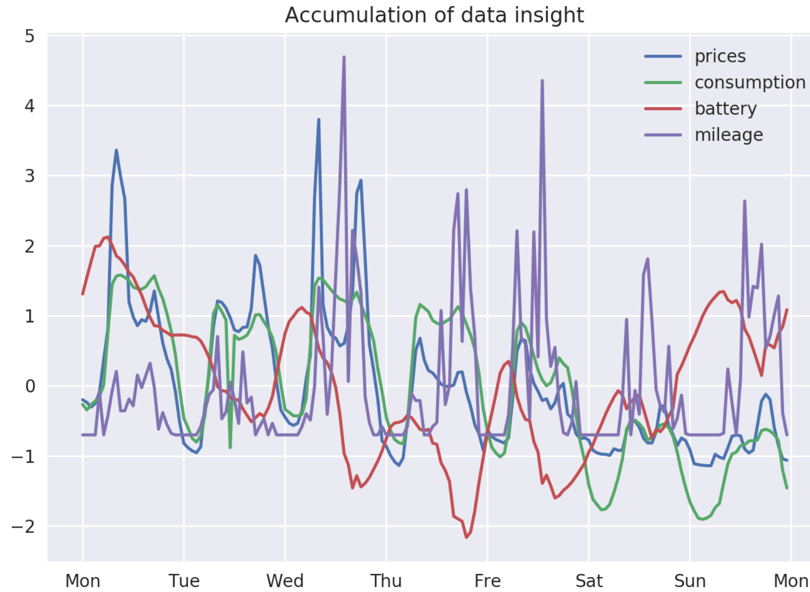


Figure 6.1: Accumulation of trend data

vehicles, however, is due to the battery capacity and average daily mileage.

We normalized the data for easier processing. What we observe, is that mileage trends follow a similar pattern as the power data. Creating a custom charging plan which is of value for the user and power distributors requires us to find time-slots with low usage probability and power trend valleys. The best case scenario is to locate times where mileage, power price, and power consumption valleys overlap. However, considering those who work and drive during the night, the optimal solution is low power trends inside time-slots where the user is less likely to drive.

For the chosen vehicle, we observe what we describe as a general mileage trend. Additionally, valleys of mileage, price and consumption trends overlap every night, giving the algorithm the optimal scenario when we consider power price and EV load. The overlapping valleys provide flexibility when an algorithm decides on charging times. A vehicle with low power demand for a given night can charge on low voltage or discard charging entirely. Additionally, the charging schedule can charge during the valley bottoms, to discourage adding load during the early and late off-peak hours.

Using the accumulated information, we created an algorithm which takes weekly mileage trends and power prices as parameters to determine the optimal charging schedule for a given vehicle. We discarded power consumption

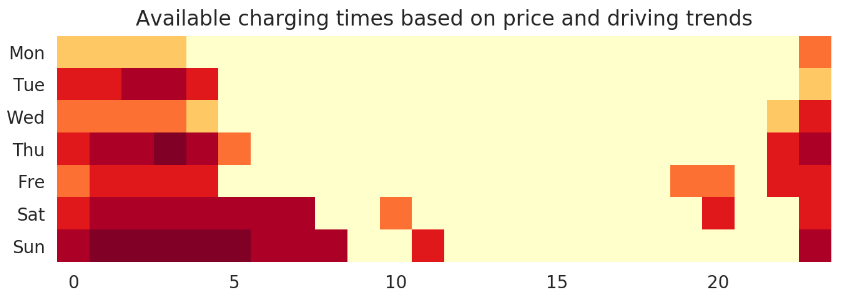


Figure 6.2: Charging recommendation

because it provides little extra information with the current trends experienced. The arguments take the form of matrixes which represent each day and hour and returns a similar matrix with the recommendation. Additionally, we added a third argument to determine charging priority, resulting in more accurate localization of optimal charging times.

Figure 6.2 illustrate recommended charging times with varying charging priorities. Given the previously mentioned data, we get night to Thursday and night to Sunday as the best time to charge. Additionally, the most optimal time to charge at any given day seems to be between 1 and 4 am.

The heatmap visualizes time-slots during the week which is beneficial for charging without obstructing the EV owners daily chores. Darker colors represent a higher benefit in regards to power cost, compared to the lighter shades of red. The charging priority we mentioned earlier adjusts the power cost consideration which decreases in correlation with an EVs power demand. Using the charging priority setting, we can ensure charging below a given threshold for the power price valley, resulting in the avoidance of charging during early and late off-peak hours.

The most significant weakness of the generated charging plan is that we currently can't adjust the charging priority automatically. The charging schedule at its current state attempts to find time-slots which minimize charging cost, instead of ensuring that the user has enough power for their daily chores. Charging priority is, therefore, adjusted by the EV owner to prevent power deficiency.

6.2 Automated Charging

Before we discuss how the charging priority can be adjusted automatically, we would like to discuss the process of automatic charging and its current

challenges. Generating a good schedule itself is a considerable obstacle. However, there is an even more significant problem which needs to be mentioned, which is the ability for third-party systems to control the charging process itself.

Our current knowledge indicates that advanced EV's such as Tesla model S and BMWi3 is capable of remote user controlled toggling of the vehicles charging state. Another significant feature is the ability to schedule charging, which blocks the charging until charging is scheduled.

These two features result in the ability for vehicle owners to follow the generated plan manually. A problem with manual toggling and scheduling is that it is easy to forget to turn on charging when we relax before bed. It is also impossible to expect people to wake up at 1 am to turn on charging, and again at 6 am to turn it off. This problem is invalidated if the charging schedule feature is used, which leads to a new challenge — remembering whether or not the schedule is configured correctly from day to day. Additionally, reconfiguring the schedule frequently can be inconvenient for the user.

Which brings us to the question: is it possible to automate the scheduling process? The answer to that question is, yes, but it depends. Depending on the available features, a third-party system can actively check a vehicle charging state, and toggle the charging procedure by mimicking the mobile application.

Actively checking and changing the charging state for large quantities of vehicles result in scaling problems, which brings us to the next topic, scheduled charging. Tesla can schedule charging start, which is a great feature. It gives third-party systems such as Cosinus the ability to adjust the schedule at regular intervals automatically.

There are a wide variety of car manufacturers and brands, where everyone has their API which operates differently from their competitors. Many of those API's are not intended for third-party use and supporting every brand would be near impossible.

Achieving automatic charging scheduling on a scale which is beneficial for power distributors will, therefore, a solution which does not require active charging state validation and toggling. Directly adjusting multiple brands scheduling is a doable task when most of the population use the same brands. However, the best solution would be interaction with the chargers directly to adjust the charging schedule.

Chargers which are connected to the cloud and interacts with other entities have recently surfaced in an attempt to solve the same problem as us. The number of charging station brands is much lower than the number of vehicle brands. To achieve fully automatic management of a vast quantity of vehicles will, therefore, require the co-operation with charging stations to more

effectively ensure charging during off-peak hours.

6.3 Estimating Driving Range

In the previous section, we introduced a charging plan which use a priority argument to finds time-slots during the week to charge a vehicle. One of the weaknesses of the generated plan is the requirement of manual input. During this section, we will take a closer look at the historical data, to find the estimated driving range for a given vehicle, which will be used to change the priority value automatically.

The estimator will then, to some degree, be able to take the driving environment and style into consideration to give a realistic estimate of the vehicles current capacity. We took a closer look at two features which directly describe the cars battery consumption rate: the current battery charge and odometer reading at the time of collection. The hourly data will provide information regarding travel distance and charge used during the travel.

We took a fairly basic approach, by analyzing the information given different timespans, but with similar techniques, we designed four different analyzing approaches which work on a flexible time series. Looking at the data from an hourly, daily and longest possible perspective, we were able to find an estimation for the driving range at max capacity.

We named our four approaches (1) **Hourly**, which is the most straightforward approach. Hourly looks at the difference in odometer and battery charge usage from hour to hour, and produce an estimate based on the hourly changes. (2) **Daily Simple** is the second approach, which expands the timeframe from hourly, to daily changes, in an attempt to iron out small and inefficient journeys which are likely to pull down the result. Further, we created (3) **Daily** and (4) **Greedy**, which extract every journey present in the presented dataset. Dividing the dataset into smaller pieces would result in a more effective filter, to remove noise.

We designed the four approaches to be reasonably flexible, giving us the ability to add and adjust the filter by sending a complementary dictionary which specifies which field we want to filter on, and how aggressive it should be. Additionally, giving the flexible timeframe which the four approaches operate under, we can adjust the input, giving us a day to day, week to week or month to month. In other words, the approaches operate on an arbitrary timeframe.

For simplicity, we chose to use a vehicle which we are familiar with, which also have the most substantial amount of data in our system as a test subject for fine-tuning and an indication of whether or not our approaches provide a

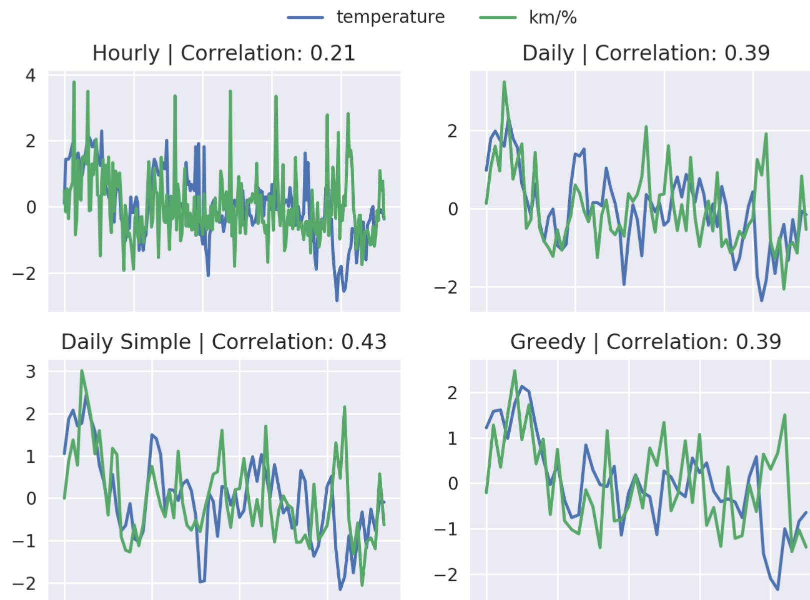


Figure 6.3: Correlation between battery efficiency and temperature

reasonable result. The vehicle in question is a Tesla Model S that has a driving range at max capacity of approximately 150 km during sub-zero temperatures and around 220 km during warmer temperatures.

Our first course of action is to verify the correlation between the produced data points, from our approaches, with the recorded temperature. Battery capacity is affected by the temperature in its environment, in addition to other factors such as degradation [30]. The correlation check will provide a pointer to how accurate the produced estimates are.

Figure 6.3 provides a visualization of the correlation between temperature and the extracted information from our approaches. Hourly is more prone to noise, and performs the worst, as we suspected. Similarly, Daily, which looks at trips taken each provides a more similar trend between the two, as some of the noise is flattened out. Greedy, which operate similarly as Daily, produced an equally positive correlation. Finally, we have our Daily Simple approach, which provided the best result when comparing extracted information with the recorded temperature. Most likely due to noise being less aggressive when looking at a timeframe compared to individual journeys.

With noise, we refer to small and ineffective trips which consume a lot more power than usual. Noise is created by trips where the driver frequently accelerates and decelerate, in addition to really small journeys where the

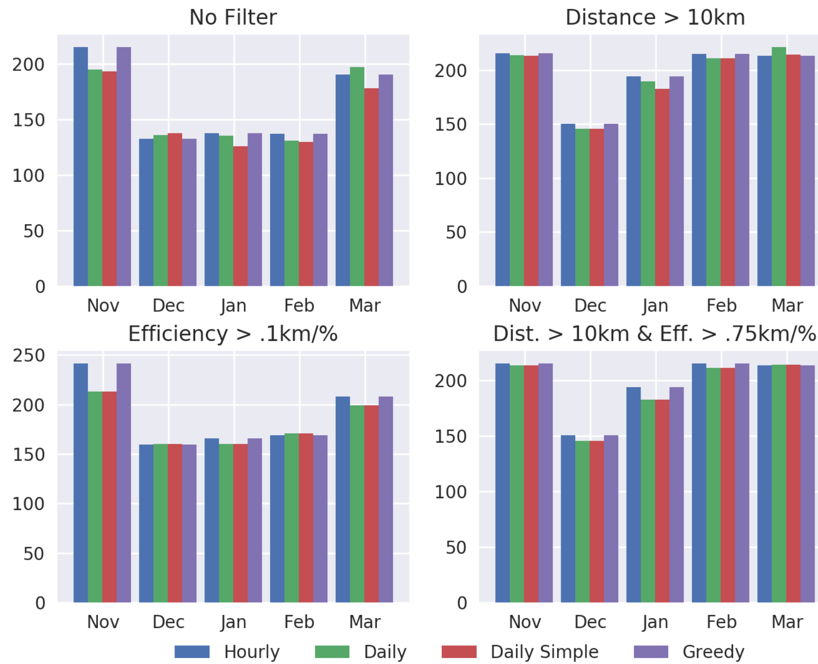


Figure 6.4: Estimated monthly driving range

procedure of ignition consumes a significant amount of that journeys battery usage.

Following the verification between extracted data points and temperature, we began to apply filters and experiment with a variety of timeframes. Figure 6.4 illustrate a small subset of the results produced from our tests when we looked at the data from in a month to month perspective. We see excellent results based on our knowledge of the car, from the raw output (no filter). With an estimated range of approximately 150 km for sub-zero temperature and 220 km for more optimal conditions, we see the approaches generating resembling numbers. Especially considering how driving environment can result in worse efficiency than systems might be able to predict.

While the results were satisfying, considering how hills and acceleration and deceleration effects the battery consumption negatively, we felt that the output was a little pessimistic. We, therefore, began to experiment with filters, to reduce the amount of noise in the dataset.

Removing data-points based on a variety of criteria affected the output positively, where the estimated range rose, just as desired. The method of removal did, however, change the result in different ways. Removing data-points based on distance traveled provided the most volatile and optimistic results. Filtering based on consumption rate, on the other hand, resulted in

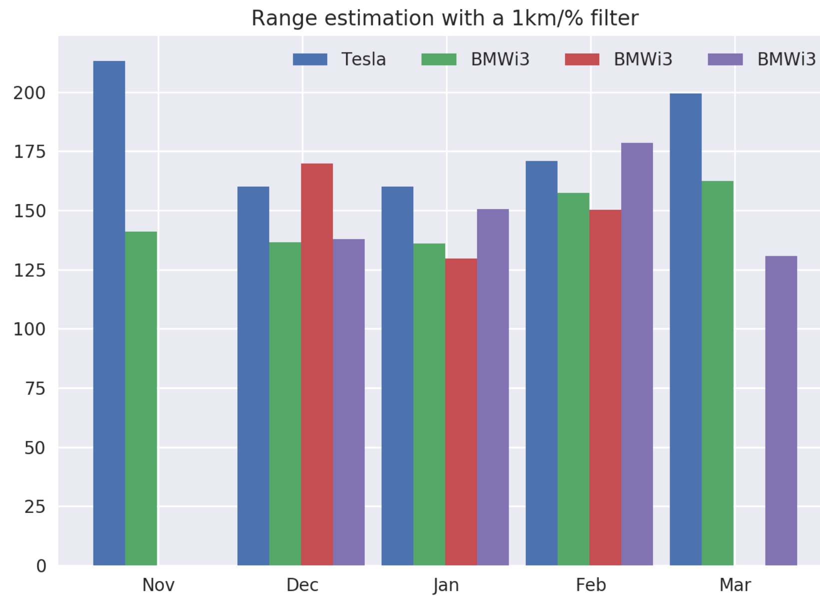


Figure 6.5: Estimated monthly driving range

a much more stable change, where the result looks more realistic and authentic to our knowledge of the performance of the vehicle. Additionally, combining distance and consumption rate filters yielded similar results to applying only a distance filter, which informs us that filtering based on distance is too aggressive, especially for vehicles which experience little use.

Following the application of different filters, and finding what we believe to be a threshold for filter aggression, we ran the Daily Simple algorithm with a consumption rate filter of one on a few other vehicles in the system. The other cars are BMWi3's, which has a lot of holes in the dataset, caused by instability in the external BMW API. The results, however, seems reasonable, given that pre-2018 performs a little bit worse compared to the Tesla Model S we used.

Figure 6.5 visualize a few other vehicles registered in the Cosinus system. The results seem reasonable, especially considering the instability we experience at regular intervals. Our method of pre-processing of the data is, therefore, a considerable factor in the result. The most significant finding is that the filter is too aggressive for one of the vehicles, as seen by the missing bar for March.

Our dataset covers the timespan of November 1st - March 7th, which indicates that a weeks worth of information is not sufficient if we want to apply filters on the data-points extracted by Daily Simple. Another essential artifact found in the dataset is unrealistically low consumption rate, where the vehicle

was able to travel up to ten kilometers each battery percent consumed. We believe the introduction of such spikes is due to our pre-processing method to patch up holes in the data. The consequence, however, is that our the pre-processing might have introduced a lot of fake data-points which is not easily detected.

6.4 Discussion

With the historical data collected by Cosinus, we were able to distinguish an EVs driving range and their owner's mileage demand. Understanding mileage trends and driving range allow for more flexible charging where charging can elapse over a few hours or be discarded entirely for a given day. We get a better grasp of how much power an EV requires to operate normally while adding the minimum necessary stress on the power grid.

Weekends are the best time of the week to charge an EV considering power cost and power loads generally experienced. An optimal charging schedule should, therefore, aim to charge as little as possible during weekends, and retrieve a full charge during the weekends. Minimum charging necessary is the optimal charging method, but will only an acceptable strategy for EVs with enough capacity for two or three charging cycle during the weekends. EVs who has a high power demand will benefit from starting charging after midnight, but won't be able to take full advantage of weekends.

A minimum charging approach will look at the EVs demand from day to day and determine whether charging is necessary that night. If not, we will charge the vehicle for a few hours on low voltage, to prolong the need a full charge. An added benefit of charging a small subset of EVs during power consumption valleys is the weighted aspects. EVs with high power demand will begin charging earlier, while EVs with a low power demand will begin their charging cycle closer to the valley bottom depending on the need. Further, by actively profiling an area's power load, we can prevent vehicles with low power demand from charging, to avoid components from overloading.

Additionally, we can combine the minimum charging approach with voltage manipulation and weighted strategies, as presented by both Richardson et al. [36] and Clement-Nyns et al. [8] Creating a priority list each night and filtering out vehicles with low power demand will result in more leeway for existing algorithms to ensure proper utilization of the power grid.

The most exciting opportunity with a minimum charging approach is the ability of intertwined charging and combining techniques. Vehicles with high power demand can charge unobstructed, while EVs with a low power demand can alternate charging, resulting in a high number of vehicles receiving some

level of power during the charging process.

However, a minimum charging approach won't necessarily be the best choice when we consider battery health. Frequent charging can result in quicker deterioration of the battery. However, low voltage charging is better for the batteries health, which can have the possibility of offsetting the disadvantages of more frequent charging. It is, however, essential to understanding that frequent charging might result in increased maintenance costs as the batteries can deteriorate faster than regular.

On the other hand, valleys will be filled up, resulting in better utilization of the power components and power distribution lines. Better use of the power delivery chain will result in more efficient use of money, from a power distributors point of view, which can result in lowered power prices.

6.5 Summary

In summary, we analyzed the driving data and combined the data with power trends to generate a weekly charging schedule with day to day adjustment. From overlapping mileage and power price valleys, we determine the optimal time of day to charge the vehicle without interfering with the EV owners daily chores. Further, we use the driving data to estimate a maximum driving range, to make day to day adjustment on the charging scheduler. Additionally, the driving range estimator draws a picture of the EVs performance from month to month.

Chapter 7

Client Application

With the Cosinus server and basic analytical algorithms in place, we need a client application for users to interact with the system. Either to contribute to the data collection procedure or to view the collected data. This chapter investigates two different cross-platform frameworks, react native and flutter, and highlights advantages and disadvantages. We design a simple architecture to gain insight into the learning curve of the two technologies and their ease of use.

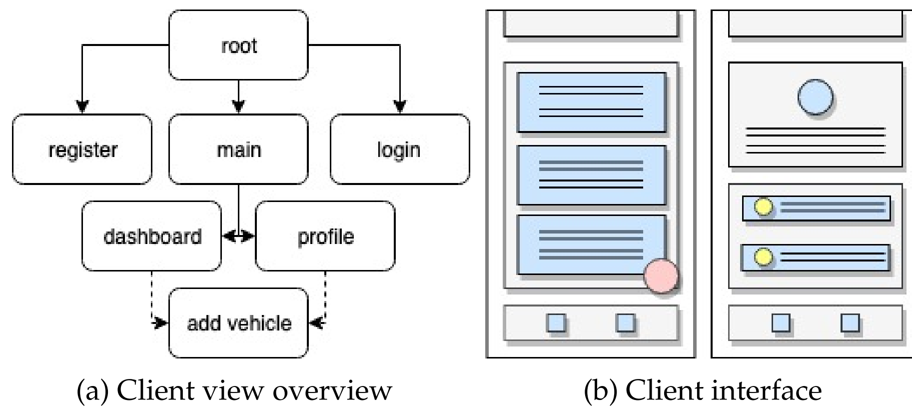
This chapter begins with an introduction to the client app architecture and the frameworks, react native and Flutter. Afterward, we present the results, in regards to usability and maintainability. Finally, we discuss the strengths and weaknesses of both frameworks, highlighting what we believe to be excellent use-cases for the two technologies.

7.1 Architecture

We designed the architecture to include regular interface elements and use patterns. With features such as one-time sign in, navigation between views and interaction with an external API (The Cosinus system). Figure 7.1 provides an overview of the interface layout and navigation flow.

On startup, the application validates the presence of an authorization token and determines which view to present the user. A landing page giving the user two options, log in or registration, is presented if no valid JWT is present. On the other hand, for validated users, a dashboard is displayed, with a navigation bar at the bottom to switch between two views, the dashboard page, and a profile page.

The dashboard page provides the user with quick information such as battery level and temperature and odometer reading, while the profile page



(a) Client view overview (b) Client interface

Figure 7.1: Client Application Architecture

offers more in-depth information related to each vehicle. Additionally, both pages provide quick access to add cars to the system. We initially had a desire to include event registration, to give EV owners the ability to inform the charging schedule of mileage outside general trends. However, a feature capable of handling such events requires more time than we have available.

We believe the current architecture will provide much insight into the capabilities of the two frameworks. It offers a variety of interface elements which exists in much mobile application. Additionally, shifting between views will give us an indication of the performance.

7.2 Cross-platform frameworks

There are different cross-platform frameworks as mentioned in chapter 2.6. The various concepts strive for a common goal, quick development of mobile applications with acceptable speed. They differ, however, in their approach. Web apps are essentially websites, with little to no connection to the underlying hardware. Hybrid and generated apps, on the other hand, provides a tighter coupling between the application and device, resulting in better-performing apps, which can provide better user experience, compared to web apps.

We decided to investigate the frameworks, react native and flutter, which is a hybrid alternative and generated app alternative. Response time and local storage are essential for a good user experience, which is why we discarded web apps as an alternative, because of the significant overhead associated with web applications. [20, 46].

React native and flutter are backed by Facebook and Google respectively and have a lot in common. Both use the reactive paradigm, which is a declarative

paradigm concerned with application state. The use of the reactive paradigm makes it possible to express both static and dynamic layouts in an object-oriented manner while keeping the application state as the core driver for interface behavior.

The difference between the two frameworks lies with the programming language used and accompanied tools and libraries. React native leverage web technology and compiles down to native widgets while Flutter compiles to native code [13]. The difference between compilation level provides Flutter with a performance advantage, bringing it closer to the device hardware.

Another difference between the two is the learning curve associated with the frameworks. React native is similar to web programming and the popular web framework react in many ways. They are giving developers who are familiar with the web a quick and easy entry point into mobile app development. Flutter, on the other hand, use a new programming language created by Google, Dart, which can create some initial difficulties for some. However, with the react developers will have a natural entry point into Flutter, with similar design patterns.

The most significant difference between the two, on the surface, is the complementary interface libraries which come with the frameworks. React native provide fundamental building blocks, which is easily configurable by the developer. The interface components which arrives with react native have no design elements applied to them, and it is up to the developers to maintain a consistent feel and look throughout the application. Flutter, on the other hand, is shipped with pre-designed components following their material design guidelines.

7.3 Navigation

Navigation between views is an essential feature to provide a rich user experience while avoiding massive view controllers. In the architecture proposed by section 7.1, either application state or user actions trigger a change of view. Both frameworks represent current views as a stack and provide a global object to manipulate the stack. The interface for stack manipulation is also similar, with pop, push and replace operations.

The use of both is also reasonably similar. A first route is defined in the root of the project, resulting in a default entry point during start-up. Afterward, a new view is pushed on the stack as illustrated by figure 7.2. The difference between the twos navigation capabilities lies with platform support and feature set. Flutter provides a more extensive feature set, out of the box, giving developers the option to create pre-defined routes which will result in more

```
// react native
this.props.navigator.push({
    component: SecondView,
    title: 'Foo Bar',
});

// Flutter
Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SecondView()),
);

// Flutter with predefined route
Navigator.pushNamed(context, '/second');
```

Figure 7.2: Navigation Examples

maintainability. All views can be defined in the same file, resulting in a quicker replacement process of view components.

The most significant difference, however, is the platform support. React native rely on third-party libraries to support both iOS and Android [33], while Flutter has built-in navigation support for both platforms. Including third-party libraries into the equation results in similar feature sets, the consequence, however, is the introduction of a new dependency. Introducing third-party dependencies has the possibility of adding rouge extensions into the project, which can result in security concerns.

7.4 Building Interfaces

React native and Flutter uses reactive-style views which result in similar organization of the code-base. The reactive-style encourages the definition of reusable components, where we define interface elements as generic elements that receive their functionality from the parent. The result is a pyramid hierarchy with data travels mainly in one direction.

The difference between react native and Flutter is the extent of the predefined components. Flutter components are pre-styled, which makes it quick in use in addition to streamlining the application's design. React native, on the other hand, give the developers all the responsibility for the look and feel of the app, resulting in much more freedom for creativity.

```
// react native
_getIPAddress = () => {
    fetch("https://httpbin.org/ip")
        .then(response => response.json())
        .then(responseJson => {
            this.setState({ _ipAddress: responseJson.origin });
        })
        .catch(error => {
            console.error(error);
        });
};

// Flutter
_getIPAddress() async {
    final url = Uri.https('httpbin.org', 'ip');
    final httpClient = HttpClient();
    var request = await httpClient.getUrl(url);
    var response = await request.close();
    var responseBody = await response.transform(utf8.decoder).join();
    String ip = json.decode(responseBody)['origin'];
    setState(() {_ipAddress = ip;});
}
```

Figure 7.3: HTTP request example

7.5 Device and Network Communication

Interacting with the device and the Internet is straightforward thanks to asynchronous operations and support for promises. Writing and reading from the local storage is reasonably quick and does not create complications for the user under regular use. We did, however, notice a slight delay between reading from local storage to updating the interface. The consequence is the requirement of a loading screen to hide the quick switch between two views as a result (The landing page appears for a few milliseconds, which is then replaced by the dashboard).

The HTTP libraries which accompanies the frameworks are easy to use. Both make use of promises to handle the request without blocking the main thread, resulting in good user experience and transition between user actions and app response.

7.6 Programming Language

The programming language used in react native, and flutter development is Javascript and Dart, as mentioned earlier. We won't go into detail about the differences in performance and syntax between the two. However, it is essential to know about what we believe to be the most crucial difference between the two and is most likely one of the causes for why flutter is a more structured framework compared to react native, which is quite flexible in use.

Dart is compiled, compared to Javascript which is interpreted in most scenarios. The advantage of code compilation is type validation while compiling, which prevent simple errors such as non-existing variables and mismatch between arguments and return values. The disadvantage, however, is that the language is strict when it comes to combining types. E.g. Javascript lists support strings, numbers and booleans to exist in the same array, while Dart only allow one type for each list defined.

Some form of type-safety can, however, be achieved in react native with a complementary static type checker such as Flow [12] or through the use of the Javascript dialect Typescript [43]. However, using typescript require the use of typescript supported libraries, which introduce extra external libraries into the project.

7.7 Conclusion

Both react native, and flutter is attractive options for mobile app development. React native is similar in many ways to web development, which result in a gentle learning curve. There are, however, a few weaknesses. The accompanied building blocks are the bare minimum necessary, which lead to a lot of time used styling the application instead than perfecting the functionalities.

On the other hand, the lightweight components result in the ability to create several different prototypes. Flutter is similar to react native in many ways, but with more type-safety and structure. The pre-designed components give developers more time to focus on functionality and ensure consistency throughout the application.

The most promising of the two seems to be flutter. Flutter is more refined as a whole, with a much more structured code-base and type-safety, which is an essential tool to prevent simple bugs from occurring. The pre-designed components and layout components will also speed up development time, as less time is required on designing and developing the interface.

7.8 Summary

In this chapter, we took a closer look at two frameworks which uses the same concept of using small building blocks to describe the user interface for a given app state. We developed the application using both frameworks to determine which one would be better to use in a proper app implementation. While the use and feel are similar across both frameworks, we saw clear advantages depending on the use-case. React native is a lightweight framework which shines when it comes to high-fidelity prototypes. Flutter, on the other hand, is much more rigid and structured, which is a desirable trait when it comes to team development and long-time maintenance.

Part III

Conclusion

Chapter 8

Conclusion

8.1 Summary

In this thesis, we explored a variety of strategies to design and develop a vehicle data collection system. We have also analyzed the data to improve our understanding of EV power demand, driving trends and EV driving range.

We began with the design and development of the data collection system, where we prioritized security and maintainability to minimize technical debt. We explored different hosting services and architectural design principles to devise a solution which would not prevent the systems from growing or moving between cloud providers.

We deployed the system inside a Google App Engine instance rather than a Google Compute Engine instance to give Google the responsibility of securing the server environment. A Compute Engine instance allows for full customization of the environment and computing resources. However, maintaining the server environment should be left to experienced engineers. The most significant sacrifice is the loss of configurable elements such as memory and virtual CPU configuration. However, giving Google control over the infrastructure allow for more resources on research and development. Further, the systems only cloud provider dependency is the cron service. Many other cloud providers offer a similar service, which is why we would argue that the system at its current state is still portable across cloud providers.

Further, we experimented with a variety of different architectural layouts to investigate and understand how different variations affected the outcome. We began with abstraction through functions and state-driven objects. Functions did not create sufficient abstraction, and state-driven objects were fragile against changes in its environment. We, therefore, chose a stateless driven design for the system.

To improve charging decisions, we began to analyze the collected driving data to improve our understanding of EVs power demand and capabilities. We found mileage trends and estimate a maximum driving range based on the last 30 days of data. Combining the two, we can roughly estimate when to charge the EV to meet power demand without creating inconvenience for the EV owner. We also leverage the range estimation feature to inform EV owners of the performance of their vehicles throughout the year.

Finally, we developed a cross-platform mobile application to interact with EV owners, giving them an interface to register their EV for monitorization and see aggregated data. To compare and ensure that the developed application is maintainable, we decided to develop the mobile app using two options we considered to be great options for the task, react native and Flutter. We found valuable insight into the strength and weaknesses of using the two frameworks and decided to move forward with Flutter. The type-safety, pre-styled components, and UI element hierarchy make Flutter an excellent option for collaborative and longtime development.

8.2 Main Contributions

We have provided a system capable of collecting and retrieving vehicle data to map out changes throughout the day. We have used data collection and storage of vehicle data as a scenario to explore how different architectural choices affect the maintainability of the system and its capabilities. We have also analyzed the collected data to improve our understanding of EV needs and capabilities by using conventional statistical approaches.

In order to achieve this, we have explored a variety of techniques to create loosely coupled components and abstraction throughout the system. The resulting system has a data collection pipeline which can be changed without affecting other parts of the pipeline. The most significant challenge of the system, however, is security. The vehicle data present an accurate representation of the EV owner trends, and must, therefore, be protected appropriately. We used a combination of different techniques to protect user data at different levels of the system. We used input validation at the systems interface, to prevent injection attack and automatically encrypt vehicle data in case database credentials is leaked due to bad key management. We achieved automatic encryption of vehicle data through the use of customized DAOs, to secure the data without affecting the maintainability, resulting in developers to interact with the database like any other ordinary objects.

In section 1.2, we outlined two questions that we can answer as follows:

1. *How should a system be built that can fulfill the task of automatically retrieving and storing driving data?*
   An I/O extensive system which relies heavily on external APIs must adapt quickly to changes in external dependencies. The system will grow to be a car aggregation system with time, with multiple different systems connected. Breaking changes can occur at any given moment, and a flexible system which supports quick changes without affecting the overall functionality of the system is therefore crucial. A flexible design is, however, no excuse to down-prioritize security. The system operates with sensitive data, and must always ensure that data protection stands at the top. The flexible system design will allow for rapid changes and allow the system to grow in features without prior knowledge of the system.

2. *How can the driving data be used for smarter and more economical charging?*
   Mileage trends can be extracted and analyzed to understand specific EVs needs. The analyzed data will provide insight into the EV owners trends and result in more flexibility when deciding the time to charge a vehicle. For individual charging, the data can ensure charging during the most economical hours without affecting the users daily chores. For collaborative charging, the data can introduce strategies such as voltage manipulation and a variety of scheduling algorithms to provide vehicles with enough power to fulfill their regular demand.

8.3 Future Work

For future work, researchers can further improve the scalability of the system. The thesis provided insight into local optimization, with traditional approaches such as multiprocessing and multithreading in addition to coroutines. Additionally, the driving range estimator has substantial overhead.

The local optimization applied during this thesis has its limits. Spawning thousands of coroutines are possible, but will most likely add considerable overhead in regards to managing them, in addition to the memory needed for each coroutine. Future work should, therefore, explore options for distributed optimization of the data collection process to further increase the number of vehicles the system is capable of handling — additionally, a distributed system distribution of network traffic between multiple destinations which prevents bottlenecks from forming.

A distributed strategy which is rising in popularity is data streams with the use of Apache Kafka. Apache Kafka is a distributed streaming platform with three key capabilities. Kafka is similar to a message queue and stores the data

in a fault-tolerant durable way. Additionally, data can be processed as they appear, resulting in real-time processing of events. Kafka introduces exciting concepts which can result in a more responsive and scalable system [22].

Future work can also explore mileage forecasting to support the development of collaborative charging strategies which fulfill an EVs needs without providing a full charge. Driving trends can change depending on the season and weather. A system which is capable of forecasting daily mileage trends with weather data as a supporting parameter will prevent a charging scheduler from underperforming and create inconveniences for the EV owner.

Another area of work is the driving range estimation used to support a charging scheduler. The current implementation analyzes a month worth of driving data to provide an estimate for the current driving range. The method of analyzing only a month worth of data makes the method rigid and incapable of adjusting to rapid changes in temperature. Future work should explore strategies to profile a vehicle based on the accumulated driving data and present an estimate based on the current temperature or future temperatures. A driving range estimator that changes based on weather reports will provide a more accurate estimate for the situation, and increase the accuracy of the EVs power demand forecasting.

8.4 Final Remarks

The comparison between the use of processes, threads or coroutines makes use of time as the only value of comparison. However, memory use is another critical parameter that was left out. We left out memory use due to problems related to monitoring the usage. The results provided by the memory profilers tested provided an increase during the first instances and leveled off for the remainder of the test. We, therefore, determined the results to be unreliable.

Additionally, in retrospect, the comparison conducted for the mobile application should have included more variety. Template based frameworks, for example, is a common approach to client based development, and should, therefore, have been included.

Part IV

Appendices

Appendix A

Monitor Optimization

```
OS: macOS Mojave 10.14.4 18E226 x86_64
Host: MacBookPro12,1
Kernel: 18.5.0
Packages: 64 (brew)
Shell: bash 3.2.57
Resolution: 1440x900@2x
DE: Aqua
WM: Quartz Compositor
WM Theme: Blue (Dark)
Terminal: iTerm
CPU: Intel i5-5257U (4) @ 2.70GHz
GPU: Intel Iris Graphics 6100
Memory: 8192MiB
```

Figure A.1: System specification

```
def busy_sleeping(n):
    for i in range(n*1000):
    time.sleep(0.001)
```

Figure A.2: Source code: Busy waiting

```
1   def primes(n):
        primes = [True for a in range(n+1)]
        primes[0] = False
        primes[1] = False
5       primes[2] = True
        check = 0
        while(check <= len(primes) / 2 + 1):
            while(not primes[check]):
9               check += 1

            flag = check * 2
            while(flag < len(primes)):
13              primes[flag] = False
                flag += check

            check += 1
```

Figure A.3: Source code: Find all prime numbers below n

Appendix B

Tailored Charging Schedules

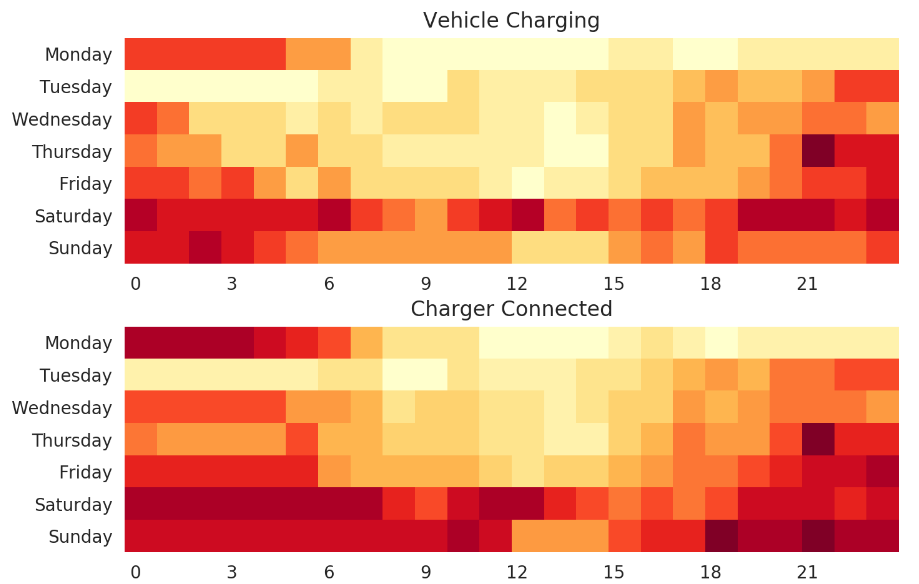


Figure B.1: Heatmap of vehicle one charging trends

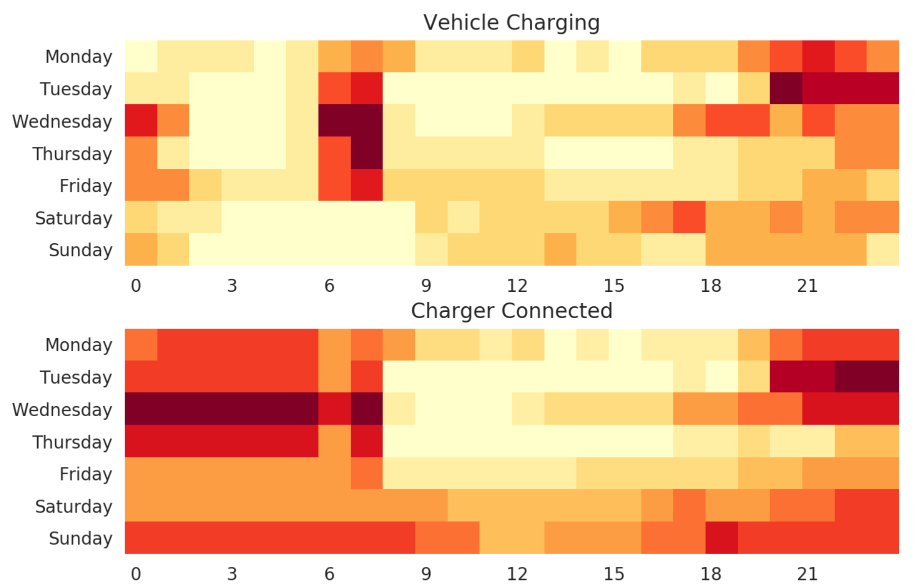


Figure B.2: Heatmap of vehicle two charging trends

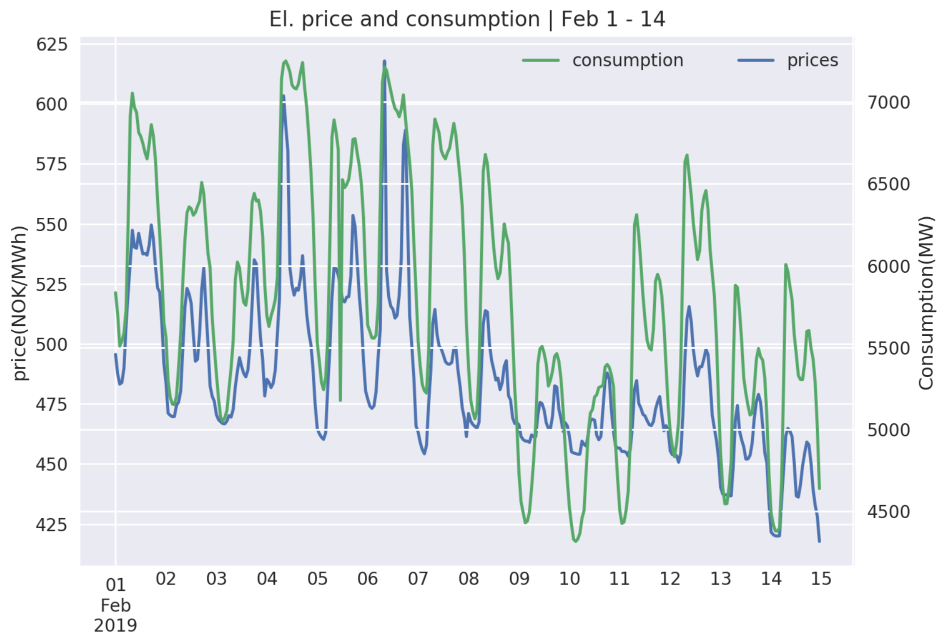


Figure B.3: Power prices and Consumption data for the first two weeks of February

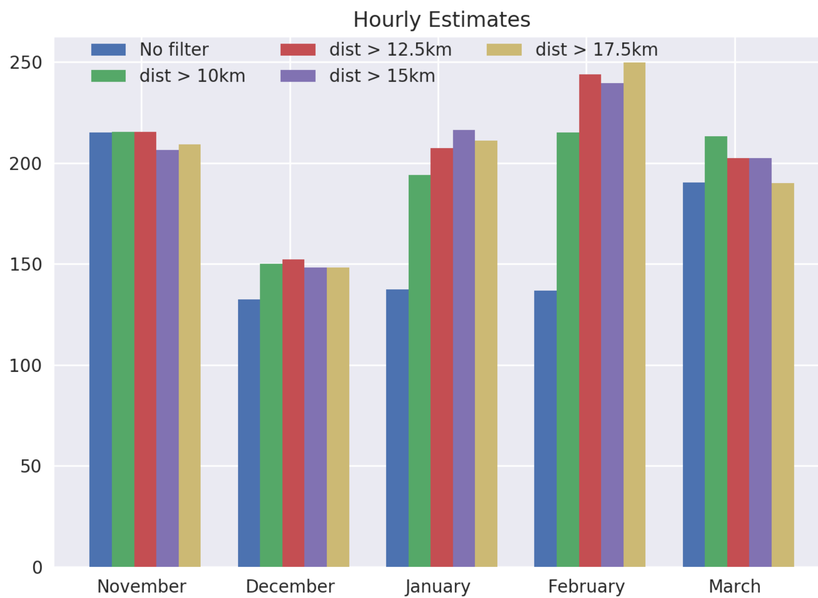


Figure B.4: Hourly with distance filters

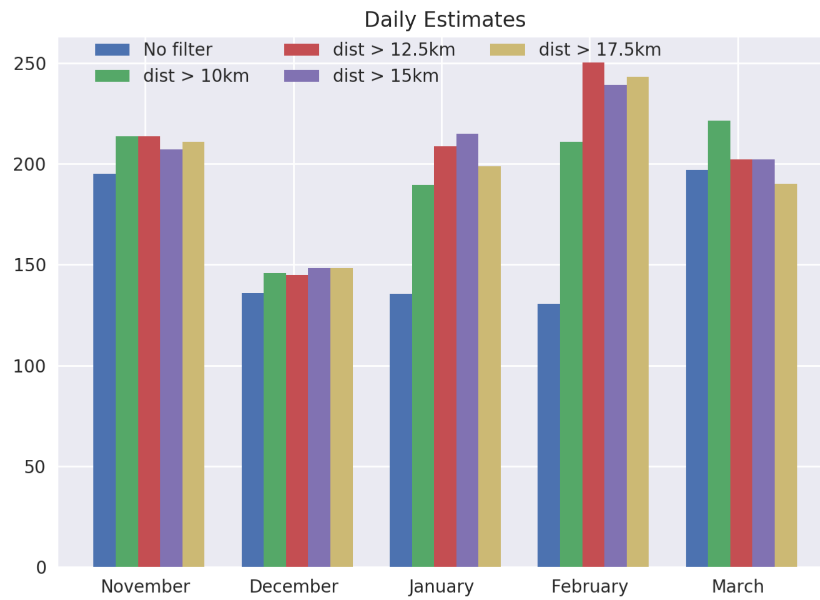


Figure B.5: Daily with distance filters

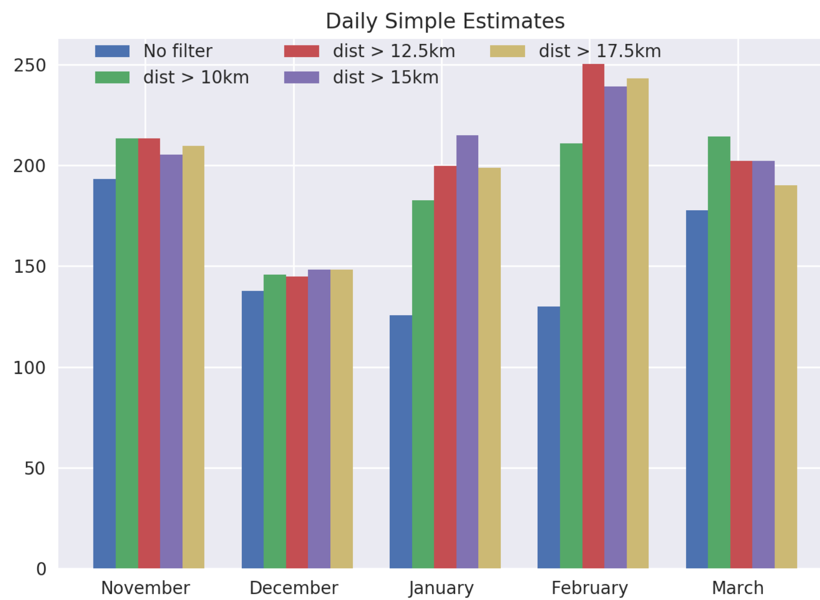


Figure B.6: Daily Simple with distance filters

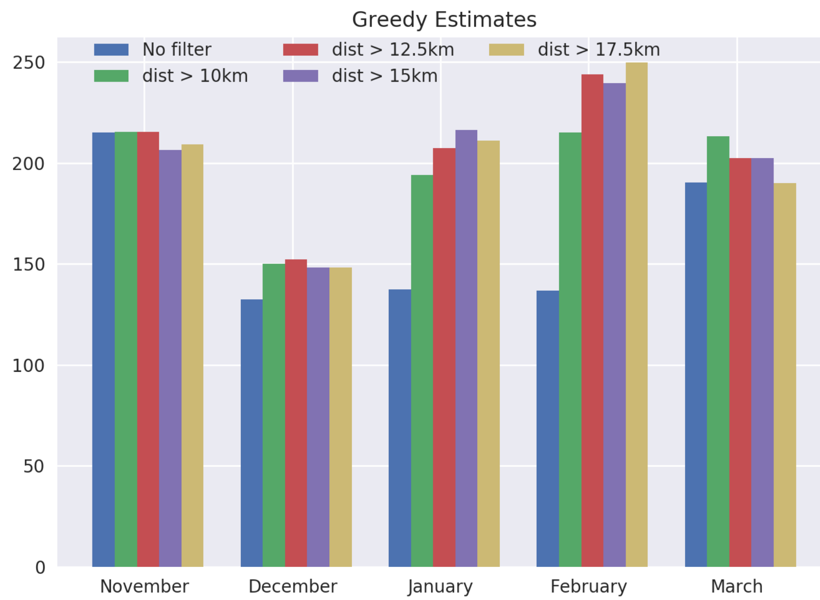


Figure B.7: Greedy with distance filters

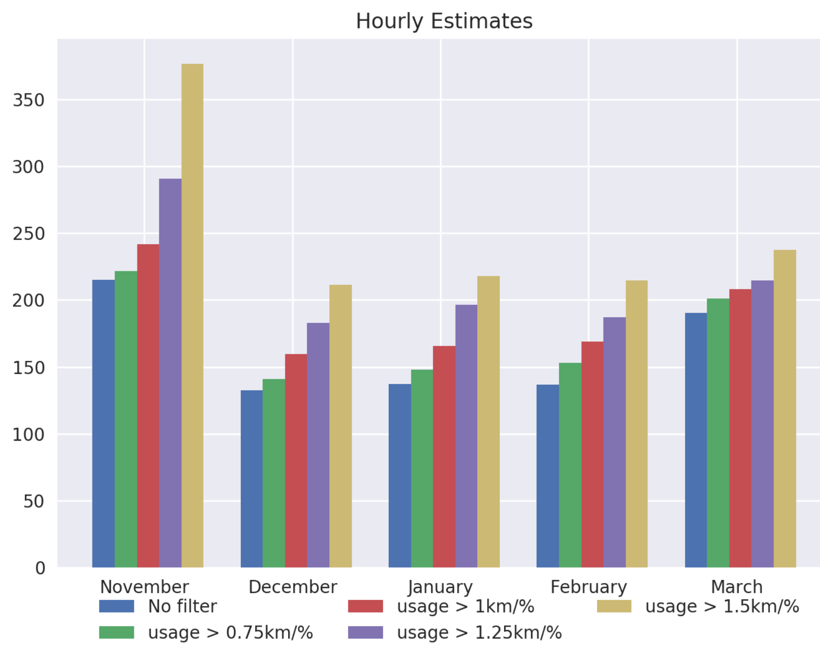


Figure B.8: Hourly with efficiency filters

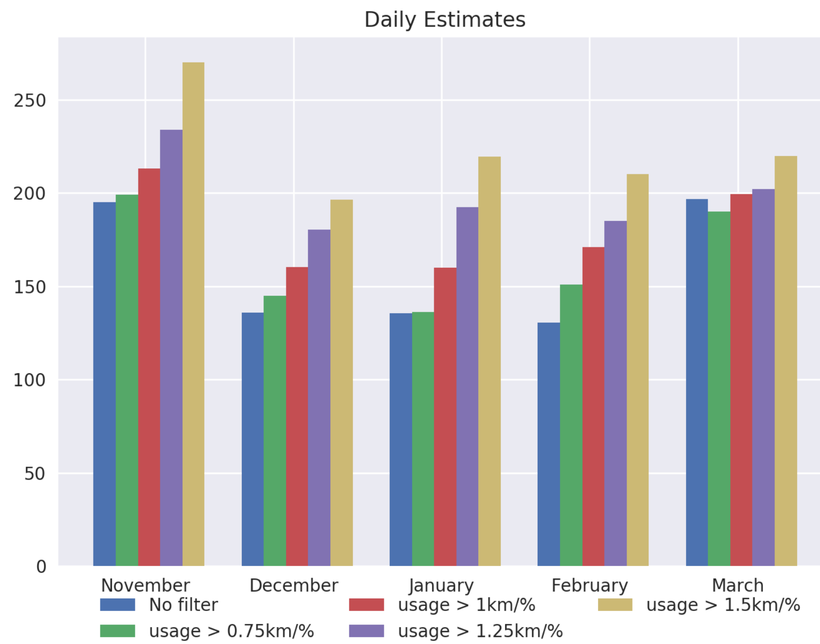


Figure B.9: Daily with efficiency filters

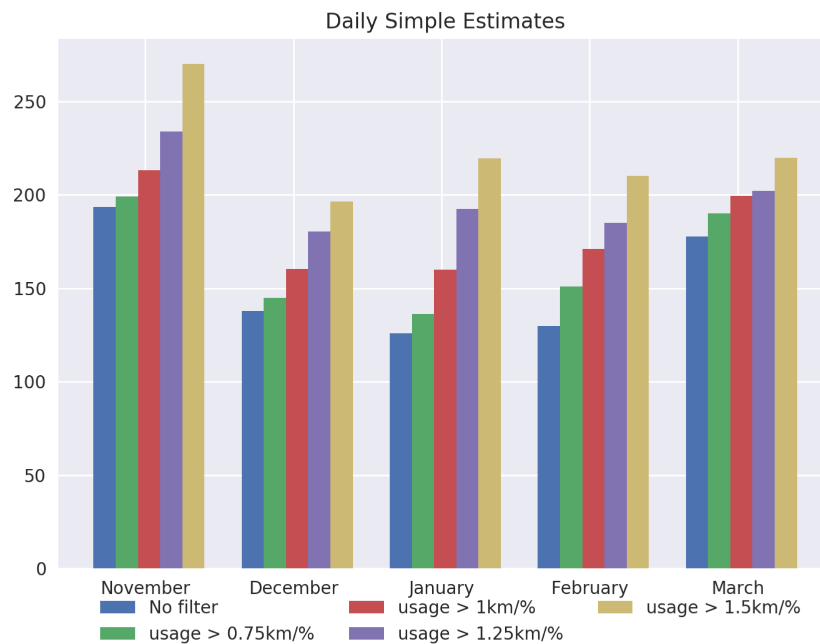


Figure B.10: Daily Simple with efficiency filters

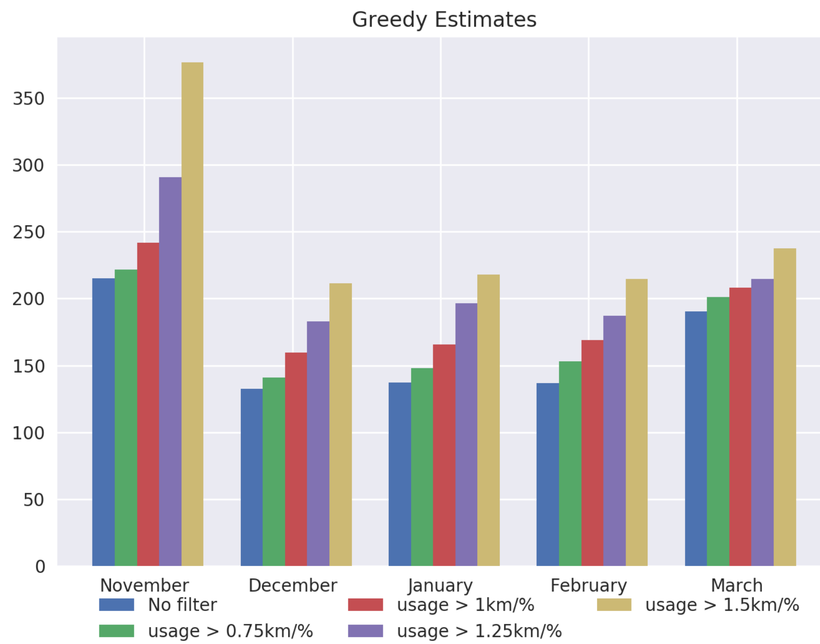


Figure B.11: Greedy with efficiency filters

Figure B.12: Hourly with distance and efficiency filters

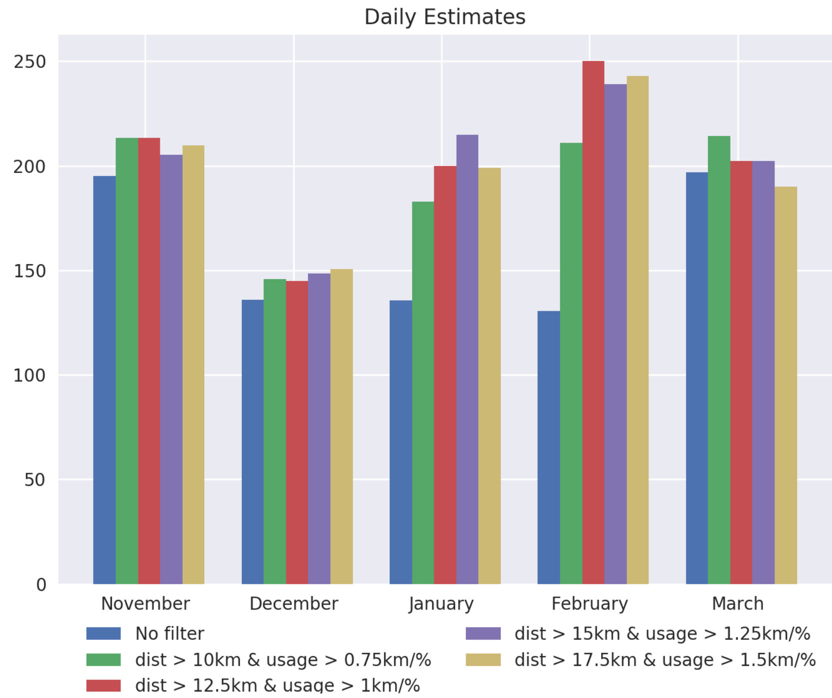


Figure B.13: Daily with distance and efficiency filters

Figure B.14: Daily Simple with distance and efficiency filters

Figure B.15: Greedy with distance and efficiency filters

Appendix C

Client Application

```
Widget _buildRow(Map<String, dynamic> vehicle) {
    Color state = _getStateColor(vehicle['state']);
    return ListTile(
        title: Text(vehicle['nickname']),
        leading: Icon(Icons.blur_circular, color: state,),
        onTap: () {
        Navigator.of(context).push(
            MaterialPageRoute(builder: (context) => Vehicle(metaData:
                vehicle))
        );
        },
    );
}
Widget _buildVehicles() {
    if (this.data == null || !this.data.containsKey('vehicles')) {
        return Placeholder(color: Colors.amber);
    }
    List<dynamic> vehicles = this.data['vehicles'];
    return ListView.builder(
        padding: const EdgeInsets.all(16.0),
        itemBuilder: (context, i) {
            if (i.isOdd) return Divider();
            final index = i~/2;
            if (index < vehicles.length) {
                return _buildRow(vehicles[index]);
            } else if (index == vehicles.length) {
                return ListTile(
                title: Text('Add Vehicle'),
                leading: Icon(Icons.add_circle),
                onTap: () {Navigator.of(context).push('/addvehicle');},
                );
            } else {
                return null;
            }
        },
    );
}
```

Figure C.1: Flutter: rendering list of vehicles

```
export class VehicleButton extends React.Component {
    render() {
        const { state, nickname, created, brand } = this.props;
        ...
        return (
        <View>
            ...
            <TouchableOpacity
                { ... this.props }
                style = {styles.paddedButton}
                onPress = {_ => this.setState({visible_modal: true})}
            />
            <View style={[styles.statusIcon, color]}></View>
            <Text style = { styles.buttonText }>
                { nickname }
            </Text>
            </TouchableOpacity>
        </View>
        )
    }
}
{vehicles.map((vehicle, i) => (
    <VehicleButton
        key={'vehicle−'+vehicle['id']}
        {... vehicle}
    />
))}
<AddVehicleButton
    addHandler={this.props.addVehicle}
    fetchProfile ={this.props.fetchProfile}
    error={error}
    errorMsg={errorMsg}
/>
```

Figure C.2: react native: rendering list of vehicles

Bibliography

[1] *App Engine Cron Jobs*. online: accessed: 12/12-2018. URL: https://cloud.google.com/appengine/docs/flexible/python/scheduling-jobs-with-cron-yaml.

[2] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[3] *Authentication hacking*. online: accessed 9/11-2018. URL: https://www.acunetix.com/websitesecurity/authentication/.

[4] David M Beazley. *Python essential reference*. Addison-Wesley Professional, 2009.

[5] Andrew D Birrell. *An introduction to programming with threads*. Digital. Systems Research Center, 1989.

[6] Kevin Bull. *Could Electric Cars Threaten the Grid*. Online: accessed 25/5-2018. URL: https://www.technologyreview.com/s/518066/could-electric-cars-threaten-the-grid/.

[7] Rajkumar Buyya, S Thamarai Selvi, and Xingchen Chu. *Object-oriented Programming with Java: Essentials and Applications*. Tata McGraw-Hill, 2009.

[8] Kristien Clement-Nyns, Edwin Haesen, and Johan Driesen. "Analysis of the impact of plug-in hybrid electric vehicles on residential distribution grids by using quadratic and dynamic programming". In: *World Electric Vehicle Journal* 3.2 (2009), pp. 214–224.

[9] Aline Dresch, Daniel Pacheco Lacerda, and José Antônio Valle Antunes Jr. *Design science research: A method for science and technology advancement*. Springer, 2014.

[10] Enova. *Marketevaluation*. Online: accessed 12/4-2018. 2017. URL: https://www.enova.no/om-enova/om-organisasjonen/publikasjoner/rapport-markedsutviklingen-2017/.

[11] Council of the European Union Presidency. *DATAPROTECT 97*. online: accessed 9/11-2018. 2015. URL: http://data.consilium.europa.eu/doc/document/ST-9565-2015-INIT/en/pdf.

[12] *Flow - Static type checker for javascript*. online: accessed 30/4-2019. URL: https://flow.org/.

[13] *Flutter for react native devs*. online: accessed 29/4-2019. URL: https://flutter.dev/docs/get-started/flutter-for/react-native-devs.

[14] Ian Foster. "What is the grid?-a three point checklist". In: *GRIDtoday* 1.6 (2002).

[15] Armando Fox et al. "Above the clouds: A berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), p. 2009.

[16] David Gale. "The law of supply and demand". In: *Mathematica scandinavica* (1955), pp. 155–169.

[17] *GDPR Key Changes*. online: accessed 9/11-2018. URL: https://eugdpr.org/the-regulation/.

[18] Joel Gibson et al. "Benefits and challenges of three cloud computing service models". In: *Computational Aspects of Social Networks (CASoN), 2012 Fourth International Conference on*. IEEE. 2012, pp. 198–205.

[19] Marc Hassenzahl and Noam Tractinsky. "User experience-a research agenda". In: *Behaviour & information technology* 25.2 (2006), pp. 91–97.

[20] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. "Evaluating cross-platform development approaches for mobile applications". In: *International Conference on Web Information Systems and Technologies*. Springer. 2012, pp. 120–138.

[21] *Injection Attacks*. online: accessed 9/11-2018. URL: https://www.acunetix.com/blog/articles/injection-attacks/.

[22] *Introduction to Apache Kafka*. online: accessed 30/4-2019. URL: https://kafka.apache.org/intro.

[23] Harvey Jones. *What's put the spark in Norway's electric car revolution?* Online: accessed 17/10-2018. 2018. URL: https://www.theguardian.com/money/2018/jul/02/norway-electric-cars-subsidies-fossil-fuel.

[24] Michael Jones, John Bradley, and Nat Sakimura. *Json web token (jwt)*. 2015.

[25] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. "Technical debt: From metaphor to theory and practice". In: *Ieee software* 29.6 (2012), pp. 18–21.

[26] Kevin Kunzelman and Sterling Hutto. *Common session token system and protocol*. US Patent 6,041,357. Mar. 2000.

[27] Zhen Li and Eileen Kraemer. "Programming with concurrency: Threads, actors, and coroutines". In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 1304–1311.

[28] *Model S REST API*. online: accessed 12/12-2018. URL: https://teslamotorsclub.com/tmc/threads/model-s-rest-api.13410/.

[29] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting coroutines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.2 (2009), p. 6.

[30] Gang Ning and Branko N Popov. "Cycle life modeling of lithium-ion batteries". In: *Journal of The Electrochemical Society* 151.10 (2004), A1584–A1591.

[31] Open Web Application Security Project. *OWASP Top 10 2017*. online: accessed 1/11-2018. 2018. URL: https://www.owasp.org/index.php/Top%5C_10-2017%5C_Top%5C_10.

[32] S Rahman and GB Shrestha. "An investigation into the impact of electric vehicle load on the electric utility distribution system". In: *IEEE Transactions on Power Delivery* 8.2 (1993), pp. 591–597.

[33] *React Native Navigation between screens*. online: accessed 29/4-2019. URL: https://facebook.github.io/react-native/docs/navigation.

[34] *Registered vehicles*. online: accessed 16/1-2019. URL: https://www.ssb.no/transport-og-reiseliv/statistikker/bilreg.

[35] Protection Regulation. "The General Data Protection Regulation". In: *INTOUCH* (2018).

[36] Peter Richardson, Damian Flynn, and Andrew Keane. "Optimal charging of electric vehicles in low-voltage distribution systems". In: *IEEE Transactions on Power Systems* 27.1 (2012), pp. 268–279.

[37] Fariza Sabrina et al. "Processing resource scheduling in programmable networks". In: *Computer communications* 28.6 (2005), pp. 676–687.

[38] *Sensitive Data Exposure*. online: accessed 9/11-2018. URL: https://blog.detectify.com/2016/07/01/owasp-top-10-sensitive-data-exposure-6/.

[39] Mohammad Shahidehpour, Hatim Yamin, and Zuyi Li. *Market operations in electric power systems: forecasting, scheduling, and risk management*. John Wiley & Sons, 2003.

[40] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[41] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*. Vol. 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[42] *Tesla JSON API*. online: accessed 12/12-2018. URL: https://tesla-api.timdorr.com/.

[43] *TypeScript - Superset of JavaScript*. online: accessed 30/4-2019. URL: https://www.typescriptlang.org/.

[44] *What is IaaS*. online: accessed 10/12-2018. URL: https://azure.microsoft.com/en-in/overview/what-is-iaas/.

[45] *What is PaaS*. online: accessed 10/12-2018. URL: https://azure.microsoft.com/en-in/overview/what-is-paas/.

[46] Spyros Xanthopoulos and Stelios Xinogalos. "A comparative analysis of cross-platform development approaches for mobile applications". In: *Proceedings of the 6th Balkan Conference in Informatics*. ACM. 2013, pp. 213–220.

[47] Qi Zhang, Lu Cheng, and Raouf Boutaba. "Cloud computing: state-of-the-art and research challenges". In: *Journal of internet services and applications* 1.1 (2010), pp. 7–18.