Advanced Technology Attachment Programme (ATAP)
Final Project Report

at

Singapore Power Limited (SP Group)

Reporting Period:

01/2023 to 06/2023

by

Ryo Armando Wijaya

Department of Information Systems

School of Computing

National University of Singapore

2023

Project Title: Electric Vehicle Charger Simulator

School Supervisor: Ang Yue Yin

Company Supervisor: Mr Alan Swan

II. Summary

An electric vehicle charging station simulator has been developed that is able to simulate a manufacturer's implementation of existing charging stations in software. This system must be able to maintain state and communicate via web sockets with the Charging Station Management System (CSMS) built by SP Digital, to be able to perform both load testing and automated testing. This system was developed following the Open Charge Point Protocol (OCPP) specification by the Open Charge Alliance. While systems have been developed that can provide a scriptable environment to simulate charging station messages, there is currently no public system that can do so via HTTP requests as well as maintain charger state. This system consists of 4 sub-systems, 1. A charger simulator built on OCPP version 1.6; 2. A charger simulator built on OCPP version 2.0.1; 3. A load testing module that utilized the simulator from system 1 to perform load testing on the company's CSMS server; and 4. A web platform to manage simulated chargers for System 1. System 1 and 2 must be able to be invoked using HTTP and must be able to be hosted either in a local server or using serverless providers.

Subject Descriptors:

Human Computer Interaction

Data Management Systems

Software Creation and Management

Keywords:

Open Charge Point Protocol, Electric Vehicles, Charger Simulator, Testing

III. Acknowledgement

This project was developed with the continuous guidance and support from the members of the CSMS team. More specifically, principal software engineer Alan Swan and senior software engineer Luy John Michael Tee. Despite their busy schedule, they took the time to advise on various aspects of development, such as domain knowledge, CSMS (Cosmos) internal specifications, CI/CD workflow, and general software engineering knowledge questions. I am also grateful to the rest of my helpful colleagues from SP Digital, who aided with various aspects of my internship, including but not limited to help with infrastructure and environment set up as well as general advice.

IV. Table of Contents

II. Summary 1
III. Acknowledgement 1
IV. Table of Contents 2
V. Legend 3
1. Introduction 4
1.1. Background of SP Group and SP Digital 4
1.2. Organizational Structure of SP Digital 4
1.3. Activity of my unit within SP Digital 5
2. Project Overview 6
2.1. How does OCPP work? 6
2.2. Problem Statement 7
2.3. Proposed solution 8
3. Project Methodology 9
3.1. Project Design 9
3.2. Technical Requirements/Deliverables 10
4. Achievement of Objectives and Work Done 11
4.1. Tasks accomplished in the 1st month 11
4.2. Tasks accomplished in the 2nd month 11
4.3. Tasks accomplished in the 3rd month 12
4.4. Tasks accomplished in the 4th month 12
4.5. Tasks accomplished in the 5th month 12
4.6. Tasks accomplished in the 6th month 13
5. Knowledge and experience gained 14
5.1. Challenges faced 14
5.2. Technical and non-technical knowledge gained 14
5.3. Organizational/Industry experience gained 14
5.4 Career Path 15
6. Conclusion 15
7. References 17
8. Appendices 18

V. Legend

Abbreviation / keyword	Expanded / Explanation
CSMS	Charging Station Management System (Software)
CPO	Charge Point Operator. Responsible for managing and setting up the charging infrastructure and charging stations.
EMSP	E-Mobility Service Provider. Responsible for offering the set-up charging infrastructure to customers and handling billing.
Cosmos	The CSMS product offering by SP Digital (Acts as a CPO)
EVA	The EMSP product offering by SP Digital (Acts as an EMSP)
SPM	SP Mobility. The business-facing unit of SP Digital's EV product solutions.
OCA	Open Charge Alliance. A global consortium of public and private electric vehicle infrastructure leaders that aims to promote open charging standards.
OCPP	Open Charge Point Protocol (A charging station AND CPO communication specification by OCA)
OCPI	Open Charge Point Interface. A CPO and EMSP communication specification by the EVRoaming Foundation.
CI/CD	Continuous integration and continuous delivery/deployment
HTTP	Hypertext Transfer Protocol
VM	Virtual Machine
EV	Electric Vehicle
LTA	Land Transport Authority of Singapore

1. Introduction

On the 11th of January 2023, I joined SP Group under its digital subsidiary, SP Digital. Under the Singapore Green Plan 2030 by LTA, Singapore is slated to grow its adoption of EV technology with the goal of reducing land transport emissions. For example, LTA has set a target of Singapore having 60,000 EV charging points by 2030 (Land Transport Authority, 2023). SP Group is one of the players in the EV space here in Singapore, and I have joined the CSMS (Cosmos) team, which focuses on building and maintaining backend and frontend systems for SP Group's EV charging products. In the following sections, I will describe the background of the company as well as the position of the unit that I joined within the company. I will also give a brief account of my onboarding process.

1.1. Background of SP Group and SP Digital

Singapore Power Limited (SP Group) is a state-owned utilities company. It is the sole electrical grid and gas grid operator in Singapore, serving millions of residential, commercial, and industrial customers. SP Group operates several subsidiaries, including SP digital, which is dubbed as the digital arm of SP Group. The primary goal of SP Digital is to spearhead digital transformation in the field of sustainable energy in an effort to deliver green energy tech solutions. One of the many products/business units that SP Digital offers is called GET (Green Energy Tech) Mobility. In accordance with the SP Digital website (2023), GET Mobility is "a platform for managing different brands and models of charging stations remotely. It enables Charge Point Operators (CPO) to onboard and configure EV charging stations, set up tariff rates and keep track of transactions, and perform monitoring and troubleshooting activities." Also referred to as SP mobility (SPM), its vision is to provide the largest and most accessible EV charging network in Singapore. As the digital arm of the corporation, it is the responsibility of SP Digital to design, create, and implement the software functionalities of the product and deliver it to the corresponding business unit, which is SP mobility.

1.2. Organizational Structure of SP Digital

As a subset of its parent organization, SP Digital has a much smaller headcount. This allows it to maintain a relatively flatter but still logical organizational structure. SP Digital is headed by managing director Ivan Tan, who reports directly to the Group CEO of SP Group. Under him lies the directors of the various business units of SP Digital, including engineering, project management, product, and others. Director Lee Chun How heads the EnergyTech engineering teams in SP Digital. Some of these teams include the CSMS team, Commercial & Industrial team, and the Infinity team (in charge of the mobile application for EV charging). My supervisor, principal software engineer Alan Swan, heads the CSMS team of which I am interning under.

1.3. Activity of my unit within SP Digital

As mentioned before, I am part of the CSMS team. The name of the project that this team manages is called Cosmos, whereas CSMS is more of an industry term. The more specific responsibility of this team includes engineering and development of 2 main items. One is the Cosmos central OCPP server, which is the backend system for which charging stations can connect to. This server acts as a platform to monitor and manage charge station information, infrastructure stability, energy, tariffs, authentication tokens, maintenance, billing information, and more. The other item is called Cosmos Web, which is the frontend dashboard where users can interact with Cosmos.

Charge stations currently connect to Cosmos following the OCPP 1.6 specification using the WebSocket protocol. Once connected, Cosmos is able to send and receive WebSocket messages to and from the charging stations. This allows the charging stations to validate any necessary information with Cosmos, such as authentication and billing information.

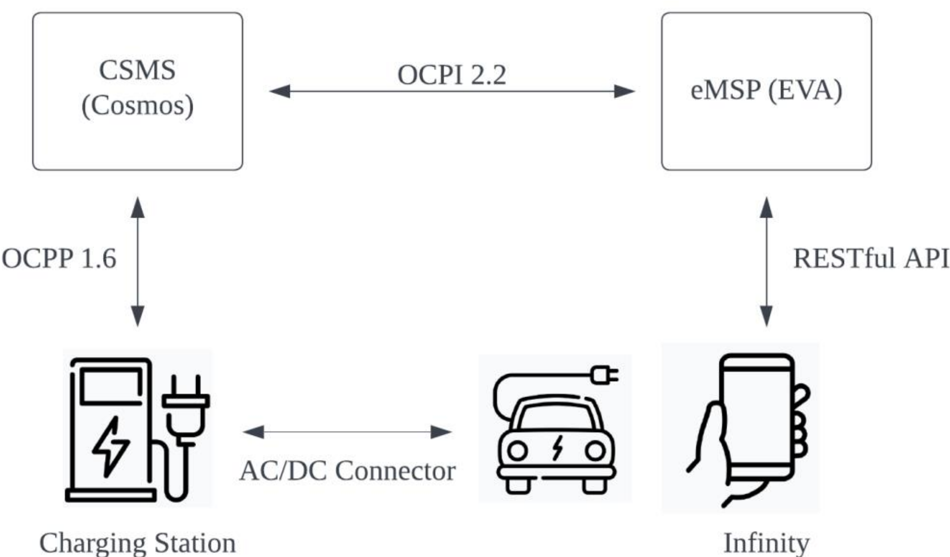

Figure 1: SP Digital EV Charging Solution Overview

Cosmos also interfaces with other systems to be able to provide the full charging solution product. The general flow of operations when a customer attempts to start EV charging can be seen in figure 1 above. The customer first plugs the vehicle into the charging station, which can charge with either alternating current or direct current. The customer can manage the charging of the vehicle via the Infinity app. The Infinity app communicates with EVA via RESTful HTTP, which is the backend system that handles payment and customer subscriptions. EVA communicates with Cosmos via OCPI 2.2 to exchange information such as billing information or other instructions. Cosmos in turn communicates with charging stations over OCPP 1.6 to manage their operations.

2. Project Overview

The overall project consists of **4 sub project modules**:

1. An OCPP 1.6 Charger Simulator that is HTTP triggerable and can be hosted either locally or serverless.
2. An OCPP 2.0.1 Charger Simulator that is HTTP triggerable and can be hosted either locally or serverless.
3. A Locust application for load testing.
4. A web platform to manage simulated chargers and facilitate OCPP message exchange with a central server for the OCPP 1.6 Charger Simulator module.

2.1. How does OCPP work?

To recap, the purpose of this application protocol is to provide an open protocol for CPOs to communicate with charging stations. The industry wide adoption of this protocol can lead to better interoperability of charging stations and charging vendors. To meet the protocol standards (e.g. to be certified as OCPP compliant), CPOs will also have to prove that they meet a certain security and operational standard.

As mentioned before, the OCPP specification also provides the conventions, terminologies, abbreviations, operation requirements, use cases and use case descriptions, datatypes, enumeration types, as well as charging station configuration variables, for each logical module of the specification. As an example, in the context of starting a charging transaction, the specification details a detailed description of the flow of messages that needs to happen, paired with a sequence diagram of the operation, as seen in figure 2 below. The specification details the prerequisites and postconditions for the transaction to take place, such as the need to send and receive an `*Authorize*` request and confirmation from the central server before the transaction can take place. The specification also details alternative flows to take and specific values that the central server or charging station should receive from one another if certain requirements or pre-conditions are not met.

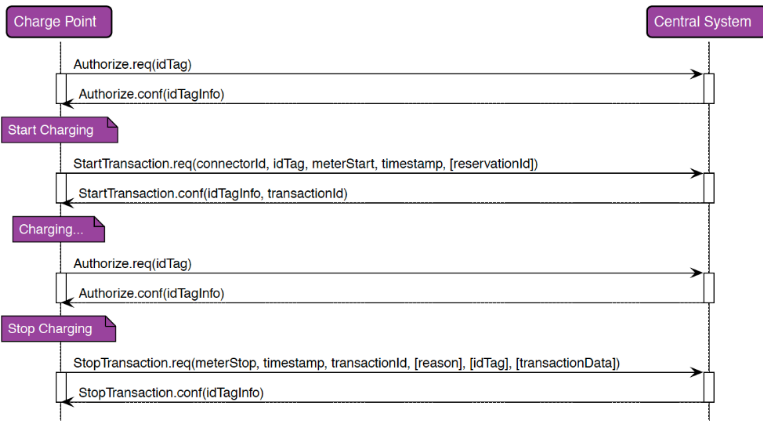

Figure 2: Simplified Sequence diagram of a charging transaction in OCPP 1.6

How is the `*Authorize*` or `*StartTransaction*` request and response WebSocket message formatted? A request and response to and from the central server and the charging station contain the following structure respectively:

Request: ***[<MessageTypeId>, "<UniqueId>", "<Action>", {<Payload>}]***

Response: ***[<MessageTypeId>, "<UniqueId>", {<Payload>}]***

where `*MessageTypeId*`, `*UniqueId*`, `*Action*` and `*Payload*` indicates whether it is a request or response, a unique identifier, the name of the operation (for example `*StartTransaction*`), and finally a JSON object containing the arguments relevant to the `*Action*`. As seen in figure 3, the specification also provides the relevant datatypes for each `*Payload*`.

FIELD NAME	FIELD TYPE	CARD.	DESCRIPTION
connectorId	integer connectorId > 0	1..1	Required. This identifies which connector of the Charge Point is used.
idTag	IdToken	1..1	Required. This contains the identifier for which a transaction has to be started.
meterStart	integer	1..1	Required. This contains the meter value in Wh for the connector at start of the transaction.
reservationId	integer	0..1	Optional. This contains the id of the reservation that terminates as a result of this transaction.
timestamp	dateTime	1..1	Required. This contains the date and time on which the transaction is started.

Figure 3: Payload for a StartTransaction request

In the specifications for OCPP 1.6, operations are divided into logistic sections, such as *Core*, *Firmware Management*, *Local Auth List Management*, *Reservation*, *Smart Charging*, and *Remote Trigger*. A full table can be seen in **Appendix A**. Each module encapsulates descriptions, operations flows, and payload information to fulfil the specific requirements of the module. As an example, `*Authorize*` and `*StartTransaction*` belonged to the *Core* module, while other modules such as *Reservation* may contain operations such as `*ReserveNow*`.

2.2. Problem Statement

The need for an OCPP 1.6 simulator rose from both the lack of any automated testing capabilities for messages to and from Cosmos and charging stations as well as the requirement to perform load testing for Cosmos. These are 2 separate requirements.

Firstly, assertion of correct OCPP messages back and forth must be done manually either with a hardware charging station or with the existing in-house local charger simulator. The existing simulator application, as well as other public charger simulator solutions such as the scriptable `docile-charge-point`, does not maintain charger state or enable triggering via serverless HTTP. This makes it not suitable for configurable complex operation flows.

Secondly, due to the growing number of charging stations that needs to be managed, it has become a requirement to be able to find out the **maximum load of charging stations** Cosmos can handle in terms of connection and communication. For example, in the event of an internet outage, it is a probability that all the managed chargers will attempt to reconnect at the same time. Cosmos must be able to handle this without failing. Knowing the maximum load that Cosmos can take will allow the team to configure the optimal pod settings (Kubernetes) such as memory and CPU resources to allow for the best price to performance ratio.

The need for an OCPP 2.0.1 simulator on the other hand rose from the future requirement of migration to the newer OCPP 2.0.1 (Cosmos is currently an OCPP 1.6 server). OCPP 2.0.1 brings additional security features, better smart charging capabilities and many other improvements to the protocol. According to cloud-based CSMS company E-Drive (EDRV, 2022), most CSMS and hardware charging station vendors still use OCPP 1.6, but are slowly shifting to OCPP 2.0.1. When the in-house migration/development of an OCPP 2.0.1 eventually starts in SP Digital, it would be beneficial to have a software simulated charger to be able to test it with.

2.3. Proposed solution

To address the problems, the above 4 sub-project modules as discussed in 2. Project Overview are required. Both the OCPP 1.6 and 2.0.1 charger simulator must be able to be hosted both locally and serverless for the purpose of load testing and automated testing respectively. These 2 simulators are meant to be built with the viewpoint of a charging station manufacturer. This means that the systems must be able to maintain state as well as communicate in a (specification accurate) manner with the central server, which is Cosmos. The 3rd module (Locust) is meant to compliment the 1st module (OCPP 1.6 simulator). As a load testing framework, locust can simulate a large number of concurrent requests to HTTP endpoint URLs. The URLs that will be used for this module will be exposed by the 1st module. The 4th module is meant to be used for ad-hoc testing by the developer team.

For the technology stack, I used the MIT licensed Python implementation of OCPP by \`mobilityhouse\`, a European EV company with SP Group investment, as it was one of the few with full OCPP 2.0.1 support. In summary, I decided to build the entire project using \`**Python 3.9**\`, \`**ocpp release 0.16.0**’ by \`mobilityhouse\`, \`**FastAPI v0.92**\` for the main client applications (1st and 2nd module), \`**Azure Functions v4.x**\`, \`**Azure Cosmos DB v4.3.0**\`, \`**Docker**\`, \`**Jenkins**\` (for CI/CD), \`**Next.js**\` (for the web platform) and finally \`**Locust v2.15.0**\` as the load testing tool. A full discovery on why I selected \`FastAPI\` to host the main bulk of my HTTP clients can be found in **Appendix B**.

3. Project Methodology

As an example of a use case that the OCPP 1.6 and OCPP 2.0.1 simulator will have to fulfil, there exist an operation, much like the `*Authorize*` and `*StartTransaction*` operations previously, called `*BootNotification*`. This is a message that is sent by a charging station to the central server (Cosmos) whenever it is started up or when connection is made. It includes the details of the charging station, such as firmware version and other relevant identifiers and allows the central server to monitor, track and configure the behaviour of the charging stations.

The user of the simulator should be able to call a specific HTTP endpoint hosted by the `FastAPI` application and pass in a charging station ID that is both registered with Cosmos as well as created in the simulator Cosmos DB database (by another endpoint previously). The application will then send and receive the `*BootNotification*` request and response to and from Cosmos using the data stored in the simulator database. This is similar to many of the other OCPP operations that can be seen in the provided OCPP specifications inside 7. References.

3.1. Project Design

Figure 4 below shows the high-level architecture of the project. HTTP requests can be made by the user either through the locally hosted `FastAPI` application itself, or through the serverless hosted `Azure Functions` provided ASGI wrapper (that allows the endpoints to be hosted on an `Azure Functions App` which forwards requests to the same `FastAPI` layer through an ASGI communication interface). The `FastAPI` application can communicate with Azure Cosmos DB through its SDK for the purpose of storing charger information and maintaining charger state. The `FastAPI` application then communicates with the central server over OCPP (WebSocket) and change charger state or return a response to the caller based on the operation.

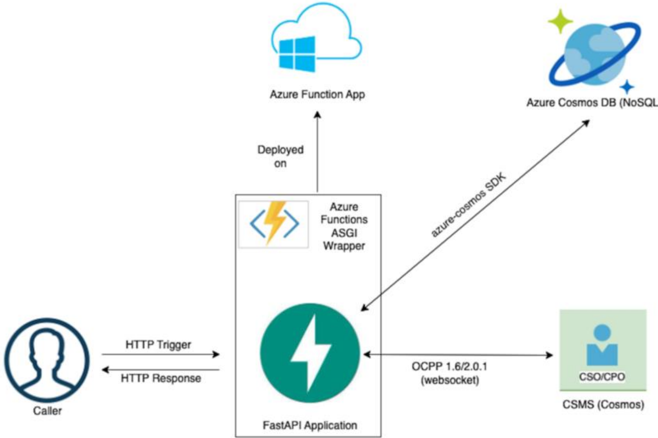

Figure 4: High Level Architecture of the Charger Simulators

3.2. Technical Requirements/Deliverables

As mentioned before, the bulk of the development of this project lies in the implementation of OCPP operations for each charger simulator and how the operations use, manage and modify the existing state of the simulated charger. It was also mentioned that OCPP operations were split into logical modules that encapsulates the function group of each set of operations, such as *Core*, *Reservation*, etc (See **Appendix A**). To streamline the organization of the deliverables, I have decided to use this structure to manage my Jira tickets, application code, as well as my milestones.

The guideline given by my supervisor for my project is: First complete OCPP 1.6 Core operations, then OCPP 2.0.1 Core operations, then work on the rest of OCPP 1.6. Hence, I have drafted some milestones to hit throughout my internship:

A full list of the completed OCPP operations can be found in **Appendix C.**

Date	Milestone
19th February	All OCPP 1.6 Core operations
24th March	All OCPP 2.0.1 Core operations
31st March	Initial load testing results for 2 test cases (ISP outage + simultaneous charging)
11th April	All OCPP 1.6 Local Auth List Management operations + Authorization Cache
11th May	The rest of the OCPP 1.6 operations
11th June	A simple web UI for OCPP 1.6

4. Achievement of Objectives and Work Done

In this section, I will give a brief summary of my development process for each month since the start of my internship up until now. This text entry can be viewed as a journal of my work done as well as the achievement of my self-defined objectives for this internship.

4.1. Tasks accomplished in the 1st month

In the first month, I spent the first 2 weeks going through the onboarding process described in section 1.4. I then did some exploration into the previously unfamiliar technology stack by creating minimal applications that utilizes `Azure Functions` and `Azure Cosmos DB`. I then completed a first draft of the project specifications document, which included the requirements and project design among other things. Due to my lack of experience with Python development (or development in general) as well as my lack of domain knowledge, I also spent some time researching on things like the ideal folder structure for this type of project, usage of environment variables, mandatory tools like `git`, formatters, linters, virtual environments, test-driven development and reading through the existing in-house simulator and Cosmos source code. I then started working on the project (OCPP 1.6 charger simulator) and created a very rough application that is able to perform invocation of a `BootNotification` and `StatusNotification` request using the `Azure Functions` Python Programming Model V1 (an API layer that does not exist anymore in the current iteration of the project). I also attended the Virtual Training Day(s) for Microsoft's Azure Fundamentals and Azure Data Fundamentals programme.

4.2. Tasks accomplished in the 2nd month

In the second month, I continued working on the OCPP 1.6 charger simulator. This month was where things started to become clearer for me with respect to how a Python OCPP application should be developed. I did a lot of refactoring to ensure that my source code is maintainable. This means consistent Python type checking across files, consistent logging, good docstrings and comments for modules and functions, consistent exception handling, using modern Python features (such as Python 3.7's Data Classes), writing unit tests with `pytest`, and more. I also spent time refactoring my code to make it adhere more to SOLID principles, such as decoupling my simulator's core operations to suit the Single Responsibility Principle. While Python is more of a scripting language, it does provide some tools that work well to fulfil these principles. With a previous background in Java, I tried my best to develop this project in a similar way without compromising on the benefits of Python as a dynamic language. After that, I completed most of the remaining OCPP 1.6 core operations. I also attended the exams for both Microsoft's Azure Fundamentals and Azure Data Fundamentals certifications and passed.

4.3. Tasks accomplished in the 3rd month

In the third month, the requirement for load testing came up. Hence, I opted to add to my API layer from the `Azure Functions` Python programming model V2 (I had migrated from V1 to V2 earlier) with another `FastAPI` layer. This application layer was expected to be able to handle thousands of concurrent HTTP requests with each request maintaining relatively lengthy WebSocket connections (to Cosmos). After some time, I realized that it did not make sense to implement new features or make changes to 2 API layers, and I abandoned the `Azure Functions` Python programming model V2 for the ASGI wrapper for `FastAPI`. Concurrently, I have also completed most of OCPP 2.0.1 core operations using a minimal local test server to ensure working message functionality (the Cosmos server does not support OCPP 2.0.1). I also performed incremental load testing on Cosmos in development and quality assurance environments to assess the maximum load the system can take (ISP outage + simultaneous charging). For example, I simulated 50, 100, 150, 200, 500, 1000 chargers connecting and sending `BootNotification` requests simultaneously to Cosmos to mimic an ISP outage and recorded the results. I also spent some time learning about Docker and how to containerize my project. I attended the Virtual Training Day for Microsoft's Azure Cloud-Native Apps. At the end of the month, I recorded a demo video detailing the progress of the project thus far for my supervisor.

4.4. Tasks accomplished in the 4th month

I spent most of April adding support in my charger simulator for OCPP 1.6 messages (other than the core profile). This includes the Local Auth List profile, Reservations profile, Smart charging profile, Firmware Management Profile, and the RemoteTrigger profile. For April, I gained deep insight into how the other OCPP 1.6 profiles work and how they are implemented in practice (Central Server wise). I also gained an idea of how these profiles can be implemented from the charger software side (manufacturer's implementation perspective), albeit my implementations being an extremely simplified version (its a simulator) and does not deal with memory management or interfacing with hardware.

4.5. Tasks accomplished in the 5th month

Most of May was spent building a Web UI for the OCPP 1.6 simulator. This allows the simulator to be used for ad hoc testing as well as function as an easily accessible demonstration of the simulator's features. This web application is a Next.js (React) one, and involves a home page, charger dashboard page, charger details + edit details page, configurations page, simulation configurations page, operations page, and logs page. The user is able to CRUD chargers, edit charger details and state, connect to a desired OCPP central server to simulate a hardware charger and exchange OCPP messages all while preserving the subsequent logs. One week of May was also spent helping the

company's reliability tech team set up the locust_loadtesting and the simulator server modules for future load testing needs.

4.6. Tasks accomplished in the 6th month

In my final month I planned to complete 3 main tasks. 1. Deploy my applications on the company's AKS cluster; 2. Prepare for my offboarding presentation; 3. Perform my offboarding clean-up tasks (clean up, update, and make concise all documentation for the 4 modules I have done `ocpp16`, `ocpp201`, `locust_loadtesting`, `ocpp16-web`, including on confluence. This is to make it easy to setup any of these applications in the future and continue development should there be any need to).
In this month I gained knowledge on how to set up a CI/CD pipeline with Jenkins, Github with webhooks, and Azure services (Vault, AKS etc). Through this I got an idea of how these pipelines are typically designed and the good practices involved at each stage.

5. Knowledge and experience gained

In this section, I will discuss the challenges I have faced thus far and go into detail on the knowledge gained or improvements I have made.

5.1. Challenges faced

The first challenge I faced was my lack of development experience in general. I have never done more than the basics in terms of version control (never made a PR), never used a linter or formatter, never done test-driven development, etc. These took some time to read about. The next challenge I faced was unfamiliarity of the technology stack. This led to some ramp up time for me to repeatedly research and refactor my project to suit to language conventions and programming best practices. For example, I had to self design and develop a Next.js web application from scratch to support my OCPP 1.6 simulator module for charger CRUD, OCPP operation facilitation, and viewing of logs. A related problem I faced was the lack of online resources for my specific project and technology stack. Serverless OCPP state-maintaining implementations are not many in the wild. I also discovered a weakness of mine where in my excitement, I tended to deliver the minimal implementation of a feature, only to have to refactor it later to add things. These also resulted in increased development time.

5.2. Technical and non-technical knowledge gained

Thus far, I have learnt the importance of in-depth research and design of a project before I start it. This includes looking into all the available tools and options and choosing the one which is a better fit and combination for the rest of the technology stack or project requirements. I have also learnt the basics of good software development, including some must-use tools and best practices described above. Unrelated to development, I have also learnt about some of the different roles and responsibilities possible in my chosen future career path (Information Technology) and the common practices in the field from my colleague. More technically, I have become comfortable with working with Python. I have also gained some experience with Azure products and how to utilize them. I gained knowledge on how to set up a CI/CD pipeline with Jenkins, Github with webhooks, and Azure services (Vault, AKS etc). Through this I got an idea of how these pipelines are typically designed and the good practices involved at each stage.

5.3. Organizational/Industry experience gained

I have gained insight on how software teams are run in big organizations. This includes experiencing established development processes such as an Agile system, learning about the interaction and communication between the various infrastructure/development units in the company, and witnessing the hierarchical decision-making process. I have also gained valuable domain knowledge on how EV charging processes typically take place and how it is implemented with OCPP.

5.4 Career Path

My goal has always been to work as a software engineer, building software products. Throughout my internship I have learned a great deal about my future career path. These include experiencing the workflow of a proper agile team and the common tooling used to achieve clean code and speed up mundane processes. Working specifically in the electric vehicle industry has also opened my eyes into the emerging tech in this semi-niche sector and taught me a lot about how things are implemented here. I found it interesting, and given the opportunity, I would jump at a chance to work in the same sector again after graduation.

6. Conclusion

To end off, I will describe a short summary of the current functionality of the project:

[For OCPP 1.6 and OCPP 2.0.1 Charger Simulator] There are 5 types of API requests a user can call to simulate charging station behaviour:

1. **Service APIs**
    a. These are admin APIs to manage charge point and its details in Cosmos DB (including evse(s) connectors, configurations/variables, etc.).
    b. *Example: Create Charge Point, Update Configurations*
2. **Standalone OCPP operations**
    a. These are single OCPP operations that a user can provide configurations for and trigger to enable a request and response from Cosmos, subjected to the current state or configurations/variables of the charge point.
    b. *Example: `Heartbeat` or `StatusNotification`*
3. **Chainable OCPP operations**
    a. These are single OCPP operations but can be chained in a custom order. This is useful to test certain operational flows of OCPP messages and can be achieved as the modified state of the charge point (if any) is saved after each operation.
    b. *Example: `Heartbeat` to `StatusNotification` and then `Heartbeat` again in a single HTTP request.*
4. **Pre-defined OCPP operations**
    a. These are a string of pre-defined operations where one or many of the included operations would not make sense as a standalone operation, or where it encompasses a logical or typical application flow.
    b. *Example: StartStopTransaction, with the specified configurations, this HTTP request sends `StartTransaction`, `StopTransaction`, `Authorize`, `MeterValues` and `StatusNotification` requests all in one HTTP call to simulate a charging transaction.*
5. **Central server to charge point OCPP operations**

a. These operations are not HTTP triggerable by the user, but when the simulated charge point is connected to the central server, any supported OCPP messages will trigger the desired processes and return the “correct” response.
b. *Example: `Reset` or `RemoteStartTransaction` or `RemoteStopTransaction`*

- Each of the 5 operations above will return a unique `operation_id` where OCPP message logs can be filtered and retrieved via another API, each with a sequential `message_id`.

[For the locust load testing module]

1. Each of the test modules contain 6 test trials. For each test trial, the number of chargers to use and the maximum length of the trail can be specified in the `.env` file.
2. Each test module can be run with a single command as described in the project `README.md`
3. The output of each test module will be outputted to a log file.
4. Locust is capable of generating a great many concurrent request through lightweight thread-like python coroutines called `greenlets`. However, ensure that there exists enough registered chargers with the same ID prefix inside both the Cosmos PostgreSQL database as well as inside the simulator’s Cosmos DB database.

- A sample report of the Locust load testing results can be found in **Appendix E**.

[For the OCPP 1.6 web platform module with Next.js]

- Users can manage CRUD of simulated chargers, charger OCPP configurations, facilitate OCPP message exchange with a central server, and view message logs.
- Central server Web Socket URL can be configured.

7. References

Open Charge Alliance. (2020). (tech.). *Open Charge Point Protocol JSON 1.6* Retrieved April 9, 2023, from https://www.openchargealliance.org/downloads/.

Open Charge Alliance. (2017). (tech.). *Open Charge Point Protocol 1.6 Edition 2*. Retrieved April 9, 2023, from https://www.openchargealliance.org/downloads/.

Open Charge Alliance. (2020). (tech.). *Open Charge Point Protocol 2.0.1* Retrieved April 9, 2023, from https://www.openchargealliance.org/downloads/.

Panion. (2022, March 2). *E-mobility service provider (EMSP) and Charge Point Operators (CPO)*. PANION. Retrieved April 9, 2023, from https://www.panion.org/emsp_and_cpo/

Open Charge Alliance. (n.d.). *Home*. Open Charge Alliance. Retrieved April 9, 2023, from https://www.openchargealliance.org/

Wikimedia Foundation. (2023, April 4). *SP Group*. Wikipedia. Retrieved April 9, 2023, from https://en.wikipedia.org/wiki/SP_Group

Land Transport Authority. (2023, March 14). *Our EV Vision*. LTA. Retrieved April 9, 2023, from https://www.lta.gov.sg/content/ltagov/en/industry_innovations/technologies/electric_vehicles/our_ev_vision.html

Digital, S. P. (n.d.). *SP digital - empowering the future of Energy*. SP Digital - Empowering The Future Of Energy. Retrieved April 9, 2023, from https://www.spdigital.sg/

Electric car charging station in Singapore. SP Mobility. (2022, December 9). Retrieved April 9, 2023, from https://www.spmobility.sg/

E-Drive. (2022). *What is The open charge point protocol (OCPP)?* Home. Retrieved April 9, 2023, from https://www.edrv.io/ocpp

ms-learn. (2023, March 17). *Python developer reference for Azure Functions*. Python developer reference for Azure Functions | Microsoft Learn. Retrieved April 9, 2023, from https://learn.microsoft.com/en-us/azure/azure-functions/functions-reference-python

Mobilityhouse. (2023, January 4). *Mobilityhouse/OCPP: Python implementation of The open charge point protocol (OCPP).* GitHub. Retrieved April 9, 2023, from https://github.com/mobilityhouse/ocpp

8. Appendices

Appendix A

An explanation of each functional group can be found below:

Profile name	Description
Core	Basic Charge Point functionality comparable with OCPP 1.5 [OCPP1.5] without support for firmware updates, local authorization list management and reservations.
Firmware Management	Support for firmware update management and diagnostic log file download.
Local Auth List Management	Features to manage the local authorization list in Charge Points.
Reservation	Support for reservation of a Charge Point.
Smart Charging	Support for basic Smart Charging, for instance using control pilot.
Remote Trigger	Support for remote triggering of Charge Point initiated messages

The grouping of all messages in their profiles can be found in the table below.

Message	Core	Firmware Management	Local Auth List Management	Reservation	Smart Charging	Remote Trigger
Authorize	X					
BootNotification	X					
ChangeAvailability	X					
ChangeConfiguration	X					
ClearCache	X					
DataTransfer	X					
GetConfiguration	X					
Heartbeat	X					
MeterValues	X					
RemoteStartTransaction	X					
RemoteStopTransaction	X					
Reset	X					
StartTransaction	X					
StatusNotification	X					
StopTransaction	X					
UnlockConnector	X					
GetDiagnostics		X				
DiagnosticsStatusNotification		X				
FirmwareStatusNotification		X				
UpdateFirmware		X				
GetLocalListVersion			X			
SendLocalList			X			
CancelReservation				X		
ReserveNow				X		
ClearChargingProfile					X	
GetCompositeSchedule					X	
SetChargingProfile					X	
TriggerMessage						X

Appendix B

Preliminary investigation for API layer**:**

This layer will be running thousands of of HTTP calls making web socket connections and will need to be performant, scalable, and preferably modern and easy to use. `Flask`, `Tornado`, `FastAPI` and `aiohttp` were considered.
Detailed comparison of options with a modified version of a decision matrix (Max 4pts each category)

Worst to best (1-4 pts)	**Performance and Scalability (according to TechEmpower benchmarks)**	**Ease of use (Syntax simplicity, lightest learning curve, modern python syntax support, documentation and popularity)**
Flask (1+4+1) 6/12	(1pt) 4th best performant. Very lightweight and suited for small-medium applications, not designed for high traffic. Routing using decorators/blueprints. Async requests handled with eventlet.	(4pts) Best usability, popularity and API design. Simple syntax. Good documentation and very widely used.
FastAPI (4+3+3) 10/12	(4pts) Best performant. Built on top of Starle tte ASGI (lightweight). Fast data serialization (with Pydantic). Leverages newer python type hints for routing and serialization.	(3pts) 2nd best usability, popularity and API design. Automated documentation (Swagger UI). Good documentation and easy to use.
Tornado (2+2+4) 10/12	(2pts) 3rd best performant. Uses event loops to handle async requests.	(2pts) 3rd best usability, popularity and API design. More complex class-based syntax for defining endpoints.
aiohttp (3+1+3) 7/12	(3pts) 2nd best performant. Built on asyncio. Supports routing with regex.	(1pt) 4th best usability, popularity and API design. More complex syntax for defining endpoints (uses a middleware system). Good documentation.

Since mobilityhouse's OCPP library already handles WebSockets with the python `websockets` library, WebSockets support might have minimal impact. Therefore I chose to prioritize ease of use and scalability/performance and try going with FastAPI first.

Appendix C

Completed OCPP operations by logical module:

OCPP 1.6

All operations

OCPP 2.0.1

Only core oeprations

Appendix D

[Removed]

Appendix E

Sample load testing results:

[Environment]

- **FastAPI application hosted on** - Azure VM
- **Load Generator** - Macbook Pro x86
- **Target System** - Cosmos (QA Environment)
- **Timestamp** - 31/3/23 3.30pm

[Test Case]

- **Description** - Simultaneous successful Boot Notification requests to simulate ISP outage
- **Link** - Create Connection Boot Notification 1 + `keep_alive_time = 5` Close Connection
- **Configurations**

```
• params = {
      "operation_interval": 0,
      "keep_alive_time": 5,
  }
```

Number of Concurrent Chargers	Success Rate (Trial 1)	Success Rate (Trial 2)	Success Rate (Trial 3)	Average Rate	Runtime in seconds (avg)	Max VM load (CPU/RAM left)	Report
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	py.html
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	68.html
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	04.html
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]