Development of a discrete event Simulator for IoT applications

Master Thesis

Submitted in partial fulfilment of the requirements for the degree of:
Master of Science in Engineering

to the University of Applied Sciences FH Campus Wien
Master Degree Program: Software Design and Engineering

Author:
Navidreza Nazifi Jafarabadi

Student Identification Number:
1910838028

Supervisor:
DI Dr. techn. Mugdim Bublin

Date:
01.08.2022

Declaration of authorship:

I declare that this Master Thesis has been written by myself. I have not used any other than the listed sources, nor have I received any unauthorized help.

I hereby certify that I have not submitted this Master Thesis in any form (to a reviewer for assessment) either in Austria or abroad.

Furthermore, I assure that the (printed and electronic) copies I have submitted are identical.

Date: ..... Signature:
.....

Kurzfassung

Aufgrund der zunehmenden Verbreitung von Smart-City-Anwendungen und IoT-Geräten werden auch die Komplexität und die Kosten für die Entwicklung dieser Anwendungen steigen, was zu Herausforderungen bei ihrer Realisierung führt. Die Erprobung und Bewertung von Smart-City-Anwendungen in einer Testumgebung vor dem Einsatz in der realen Welt können die Herausforderungen dieser Projekte erheblich verringern. Um diese Projekte zu testen, müssen ihre Entwickler entweder echte IoT-Geräte verwenden, die kostspielig sind und nicht in großem Maßstab getestet werden können, oder Simulator-Tools verwenden. Eines der Probleme für IoT-Anwendungsentwickler besteht darin, dass die meisten dieser Simulatoren für eine breite Palette von Projekten entwickelt wurden oder einige Einschränkungen aufweisen, sodass sie für bestimmte Zwecke angepasst werden müssen, was ausreichende Kenntnisse in der Softwareentwicklung und Programmierung erfordert.

In dieser Arbeit stellen wir einen neuen IoT-Simulator vor, der für die Modellierung von IoT-Geräten auf der Sensorschicht zum Testen von IoT-Anwendungen und IoT-Diensten verwendet werden kann. Unser Ziel ist es, in einem ersten Schritt intelligente Parkplätze zu simulieren und das Tool für andere Smart-City-Komponenten erweiterbar zu machen. Die einfache Konfiguration der Softwareparameter und die Verwendung des entwickelten Algorithmus gehören zu den Merkmalen dieses Simulators. Für diese Studie wurden Stichproben aus der Stadt Melbourne, Australien, verwendet. In dieser Arbeit beschreiben wir den Entwurfs- und Entwicklungsprozess dieses Tools und bewerten die Ergebnisse, indem wir sie mit realen Daten vergleichen.

Abstract

With the increasing proliferation of Smart City applications and Internet-of-Things devices, the complexity and cost of developing these applications will also increase, leading to their realization challenges. The experimentation and evaluation of Smart City applications in a test environment before deployment in the real world can significantly reduce the challenges of these projects. However, to test these projects, their developers must either use real IoT devices, which are costly and cannot be tested on a large scale, or use simulator tools. One of the problems for IoT application developers is that most of these simulators have been developed for a wide range of projects or have some limitations, so they need to be customized for specific purposes, which requires sufficient software development and programming skills.

In this work, we present a new IoT simulator that can be used to model IoT devices at the sensing layer for testing IoT applications and IoT services. We aim to simulate smart parking lots in the first step and make the tool extendable for other Smart City components. The simple configuration of software parameters and the use of the developed algorithm is among this simulator's features. This study used sampled data collected from the city of Melbourne, Australia. In this thesis, we describe this tool's design and development process and evaluate the results by comparing them with real data.

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CoAP	Constrained Application Protocol
Colab	Google Collaboratory
DoS	Denial of Service
EC2	Amazon Elastic Compute Cloud
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ICT	Information and Communications Technology
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
K-S Test	Kolmogorov-Smirnov Test
MAE	Mean Absolute Error
MQTT	Message Queuing Telemetry Transport
MSE	Mean Squared Error
NRMSE	Normalized Root Mean Squared Error
R2	R-Squared (R^2)
RDS	Amazon Relational Database Service
REST	Representational State Transfer
RMSE	Root Mean Squared Error
RSS	Residual Sum of Squares
SSE	Server-Sent Event

Key Terms

Smart City

Internet of Things

IoT Simulation

Modelling and Simulating

Smart Parking

Big data

Contents

1. Motivation and Overview

Population growth has generated significant problems in some cities. Over 55% of the world's population currently lives in cities [1]. It is estimated that this amount will reach 68% by 2050 [2]. Electricity consumption in cities is about 80% of total electricity consumption, of which 60% is used for street lighting [3]. Transportation and traffic are among the other problems of a growing population in cities. According to the report in [4], about 40% of road traffic is caused by cars whose drivers are looking for a parking space.

As mentioned, cities' population and associated problems are increasing rapidly. One of the most important components of a city is its transportation and mobility system. Improving the transportation infrastructure and using innovative ideas can not only improve the efficiency of an urban transportation system and its accessibility, but it can also reduce the consumption of resources. Smart Mobility is one of the most important ways to increase efficiency and quality [5].

Internet of Things is another important component of the Smart City, which is necessary for a city to become smart. The Internet of Things is an information architecture based on the global internet that serves to exchange data, goods and services in a global network. The Internet of Things includes various services, technologies and standards that lead to the provision of innovative services and increase productivity and efficiency in urban systems. As a result, the Internet of Things is expected to be considered one of the poles of the ICT market in the next ten years [6].

Nowadays, various smart applications are widely used, especially in the field of Smart Mobility, including smart parking, smart street lighting, smart traffic control, smart waste management, and so on. Although the transition to Smart City and Smart Mobility is considered a new challenge, it also brings benefits, including improving the quality of life, reducing costs, saving resources and energy, and protecting the environment [5].

However, there are significant obstacles to Smart Cities and their associated infrastructure. Since these infrastructures have a profound impact on the lives of citizens, any disruption brings many problems for city residents (e.g., traffic disruptions, long-term power outages, or overflowing trash cans), which leads to dissatisfaction among the population and also causes enormous financial losses. Moreover, due to the high costs of such projects, their implementation is always under great pressure. Another important challenge related to the Internet of Things is the construction and maintenance of a large network of smart devices (e.g., energy meters, parking lots, street lighting and traffic), through which a large amount of data is sent to the system every day for intelligent decision-making [7]. In any case, the Smart City is a reality, and the demand for smart infrastructure is high due to its efficiency.

An effective solution to minimize these challenges is to use a simulator to model the behavior of Smart City projects and verify that IoT applications, protocols, and network layers can be successful in specific scenarios [8]. The main purpose of an IoT device simulator is to help application developers learn how to use IoT devices without having to buy real sensors or devices and test IoT applications using simulation programs or emulators. The structure of the simulation program allows users to quickly and efficiently

simulate the IoT environment by using custom setting options [9]. The simulation results are used to understand the real system and make appropriate decisions.

For this reason, this master's thesis focuses on developing an IoT simulator for modeling and simulating IoT devices. A simulator for modeling parking lots was also developed as the first step in this direction. In order to identify the requirements of an IoT simulator, various resources in the Smart City field, specifically smart parking lots, were investigated, including studying similar work and reviewing the required parameters for a parking lot, as well as analyzing existing smart parking lot datasets.

1.1 Research Questions

The following research questions are derived from the problem statement. This thesis aims to answer these questions.

- What features should a discrete event simulator for the Internet of Things have?
- What should be considered for the implementation of an IoT simulator?
- How do we develop a multi-purpose simulator or an extensible simulator for the Internet of Things?
- Which parameters are required to simulate a smart parking lot? How can we find out these parameters?
- How realistically can smart parking lots be simulated with different environmental conditions and characteristics?
- How realistically is it possible to simulate the occupancy of a parking lot?
- How accurate is the simulated data?
- How can data sets be generated that are suitable for testing smart parking algorithms and models?

1.2 Research Objectives

The main objective of this work is to develop a Java-based web application with the ability to simulate and model smart parking lots and their occupancy. The simulator will use to generate data for various purposes, such as testing the performance of an IoT application by applying the maximum load in terms of software availability, testing network latency, client-server processing, load balancing between servers, and testing the impact of three Cloud, Fog, and Edge computing model on Smart City applications. Therefore, Smart City application developers do not need to use real sensors or IoT devices to test or deploy an IoT application. The IoT simulator can also be useful for city planning decisions.

The target audience of this project is scientists, engineers and city managers working within the framework of the Smart City.

Below we list other goals that we are pursuing here:

- Configurability: this simulator will be configurable. Therefore, it can be used for different applications and data models. For example, the user can simulate smart parking and store the generated data on a server or in a database.
- Extensibility: the application should be extensible for other Smart City purposes such as street lighting or traffic signals.
- Simplicity and user-friendly: the application should be able to be used by people with little programming knowledge. For example, an analyst or a city manager can easily use the simulator by setting the required parameters individually. For this reason, a suitable algorithm should be implemented by default.
- Reproducibility: The simulator can repeatedly run on specific data sets and produce similar results.
- Default algorithm with sufficient accuracy: developing an algorithm to simulate parking lots is also one of the goals of this work. This algorithm must have good accuracy so that users can get results that are close to the real system.
- Open-source code: This application will be open-source for other developers. The open-source code will lead to the development of the program for other IoT devices and improve its capabilities and quality.
- In addition, this work tries to take further advantages of the simulator, including scalability, manageability, controllability, and interaction

1.3 Thesis Overview

Following this introduction:

Chapter 2 explains the basic concepts, and related work in the field is discussed. This chapter aims to learn more about the different points of view in the Smart City and Internet of Things field.

Chapter 3 presents the efforts to implement the chosen solution to the previously described problem. The following points are explained in this chapter: a general overview of the solution, the analysis and investigation of a data set, the design of an architecture for the simulator, and finally, the details of the implementation of the simulator.

In **chapter 4**, the proposed solution is evaluated through two experiments. On the one hand, the validation of the generated data and, on the other hand, the performance evaluation of the software is concerned. Furthermore, the obtained results are interpreted.

In **chapter 5**, finally, the thesis ends with a conclusion of the whole thesis, the proposed solution and future works.

2. Theoretical Background

This chapter will first explain the fundamental topics related to the IoT Simulation. Such concepts as Smart City, Internet of Things, Big Data, and Simulation will help to better understand the main topic and the problem. Next, in section 2.5, we will review the works in the field of Smart City and Simulation in Smart City. Exploring related works will help us move toward innovation.

2.1 Smart City

The term “Smart City” was first proposed in 1998, but still, today, there is no clear definition for it. This is probably due to its huge scope [10]. However, in 2010, a study by IBM, provided a clearer vision for the Smart City than before, based on the reference in [10]. This study divides smart cities into six main systems based on different and diverse urban infrastructures. It explains how they affect the performance of a city's main functions, such as organization, economy, transportation, communication, and energy.

MIT defines a Smart City as “cities must be viewed as systems of systems, and there are increasing opportunities to deploy digital nervous systems, intelligent responsiveness, and optimization at every level of systems integration” [11].

As mentioned earlier, there is no universally accepted definition of the term “Smart City”. However, it can be generically defined as transforming a city from a conventional state to a sustainable one that brings together all relevant stakeholders and residents. It interacts with various aspects of a city, including social, human, and environmental aspects. It also takes steps toward technological advancement, improving living standards, and managing existing resources [12].

Neckermann explains in [11] that in a Smart City , a combination of data analysis, use of resources and infrastructure, and human resources can improve the quality of life and meet the needs and demands raised in a city. The Smart City is purposeful, efficient, and creative and provides a new experience for its residents.

In [13] and [14], Anthopoulos et al. analyzed Smart City models well and finally presented an ecosystem with eight components for the Smart City . These eight components are listed below and shown in graphical form in Figure 2.1, taken from [13].

1. Smart Infrastructure
2. Smart Mobility
3. Smart Environment
4. Smart Services
5. Smart Governance
6. Smart People
7. Smart Living
8. Smart Economy

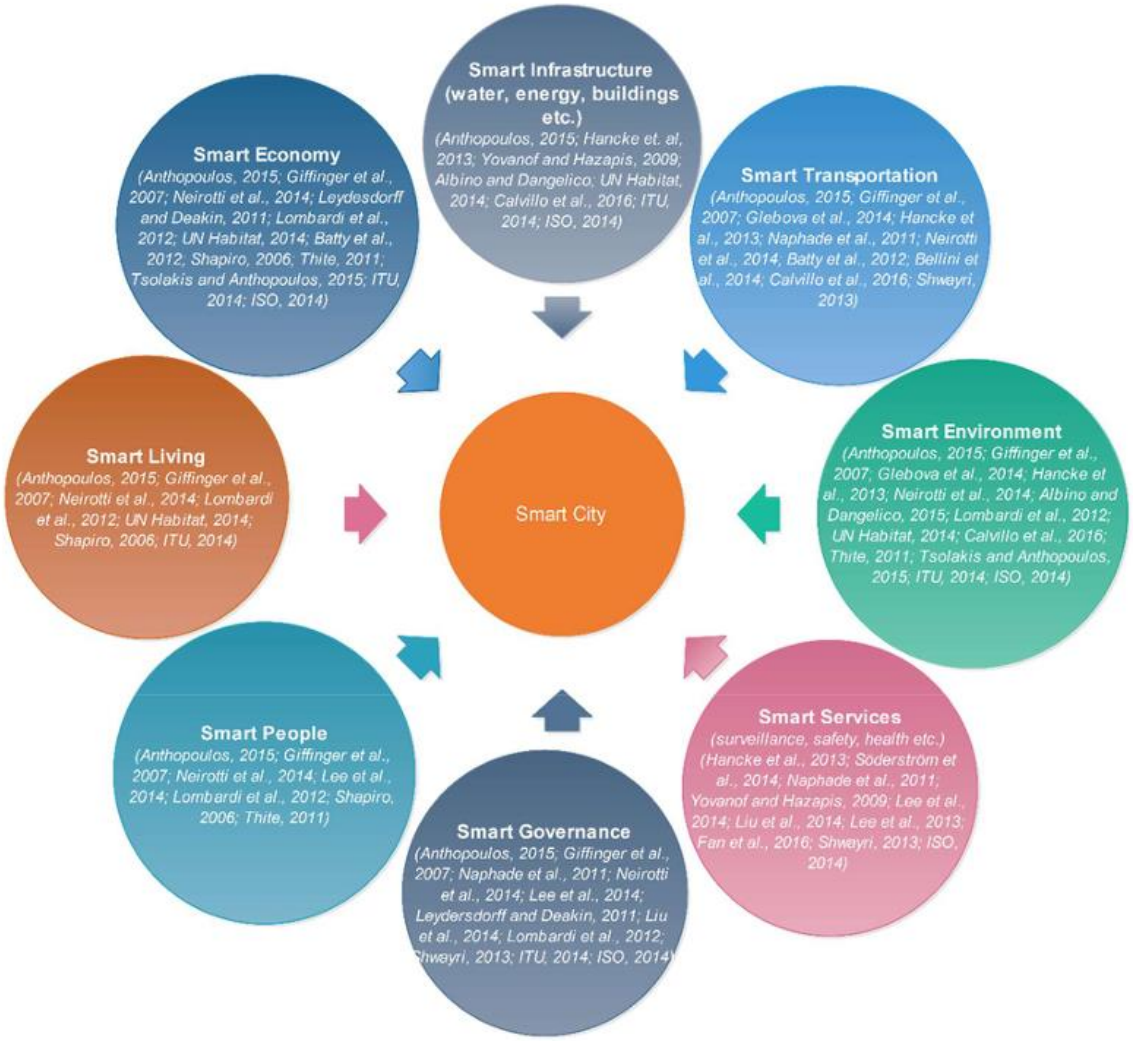


Figure 2.1. Smart City components, Source ([13], Fig. 2.3.1, p.13)

2.1.1 Smart Mobility

In [11], the author likens Smart Mobility in the Smart City to the blood circulation system in the human body. One of the most important parts of cities is the transportation system. For this reason, Smart Mobility is considered one of the most crucial issues in a Smart City, directly affecting various aspects of citizens' quality of life. The concept of Smart Mobility covers a wide spectrum of urban transportation systems and their logistics with the support of ICT. This can take many forms, including carpooling, car sharing, public transportation, walking, biking, and more. It also includes some elements such as technology, mobility infrastructure, mobility solutions and people [15].

2.1.2 Smart Parking lot

One of the subsets of Smart Mobility is the smart parking system, which uses innovative goals to make the parking process easier and more efficient. Since it is an intelligent system,

it becomes easier, cheaper, and safer to manage. Furthermore, since data is collected quickly and analyzed in a central system, it is possible to get good information about traffic congestion and vacant and occupied parking spots. With this information, the time and effort required to find a parking spot will be reduced, reducing traffic congestion and pollution emissions [16]. Also, here it is acknowledged that the smart parking system is considered the most advantageous, modern and innovative method.

2.1.3 Internet of Things

One of the most important components of a Smart City is intelligent devices or the Internet of Things. The term “Internet of Things” was first used by Kevin Ashton in 1999 [17]. The Internet of Things is one of the most recent and important technologies in our world today, connecting physical devices via the Internet and enabling them to interact with each other.

This term comprises the two words “Internet” and “things”. “Things” (physical objects) are equipped with technologies such as sensors and software to send or receive data to other devices and systems via network protocols [18]. On the other hand, the “Internet” is a vast global network of servers, computers, and smartphones connected via network protocols and can send, receive, or communicate data [19].

2.2 Big Data

Big Data generally refers to the process of storing large amounts of data and analyzing it. Storing large amounts of data or accessing it for analysis is not new. In the early 2000s, computer scientist Doug Lane formulated his definition of Big Data for the first time, based on his 3-V model (Variety, Volume, Velocity). According to this model, Big Data should have these three characteristics. In short, this model expresses that the amount of data is growing rapidly in different forms, types and content [20], [21].

Big Data can play an important role in obtaining valuable information for decision-making purposes. Decision-making processes are changing drastically due to the digitalization of systems. Therefore, the dialogue and interaction between city managers and data experts seem very important. Because the combination of the Internet of Things and Big Data brings new and interesting challenges to achieve the goals of Smart Cities. [22] and [23] have discussed the impact of Big Data on the Smart City.

2.3 Cloud, Fog and Edge computing

Cloud computing refers to a delivery model in which data storage, servers, applications and more are delivered over the Internet. The delivery of information technology services in Cloud computing, such as software, is in the Cloud, not in physical space. Therefore, there is no need to install and maintain servers and other IT infrastructure locally, and there is also no need to have the in-house expertise to manage them [24].

Edge computing enables some data to be processed locally and sent to the Cloud. Edge computing is performed directly at the outermost edge of the network near devices or small

local systems. This is done to reduce network load and server response time and to achieve high scalability. On the other hand, local devices (Edge nodes) can intervene to control or adjust the local system through decisions made in the Cloud and sent to it as commands [25], [26].

Fog computing is a processing layer between the Edge and the Cloud. The goals of creating Fog are similar to the Edge, except that the Edge is closer to the end devices, and Fog acts as a Cloud for the Edges. It is called Fog because it is closer to the Earth than Clouds and has structures similar to Clouds [25], [26].

In addition to the three sources mentioned above, Tanwar compared these three concepts (Cloud, Fog and Edge) and summarized them in the form of a table. In this comparison, various aspects such as data processing, architecture, scaling, usage, storage and latency are considered (Source: [27], Table 1, p.182).

2.4 Simulation

Simulation means the recreation of real events in a controlled environment. Since some tests in the real world are costly and, in some cases, dangerous, using a simulator can largely reduce these risks. A simulation can be a specific system, a mathematical model, or a computer programming model that simulates real-time events. The Cambridge dictionary defines a simulator as: “a bit of equipment that is designed to represent real conditions” [28]. According to Laura Adler [29], a Harvard PhD student in sociology and researcher at Smart City Solutions Program, “The most fundamental benefit of simulation is the ability to mitigate the problem of 'unintended consequences' by using realistic models to predict effects”.

Many simulators allow researchers to configure and manage the simulator through a graphical user interface (GUI). Other advantages of simulators include scalability, controllability, repeatability, and interaction [28].

2.4.1 Model and Modelling

The first step in simulation is modeling, i.e., creating a model that represents the characteristics and features of an object in the real world or a system. A model represents a simple representation of a reality that can correspond exactly or with some approximation to the real system [30]. For example, if you want to build a house, you can create several models of it and find the best one. Without modeling, this is not possible because you can't build several houses and then choose one in reality. Another example: If you want to test the efficiency of an IoT application, you can model the IoT devices and use a simulator to create different IoT devices that continuously interact or communicate with the IoT application.

2.4.2 Discrete-event Simulation

One of the types of simulation models is discrete event simulation. In this book [31], discrete event simulation is described in detail in chapter 18. First, it has classified the simulations according to the two criteria of state space and time evolution. An image of this classification is shown in the figure below.

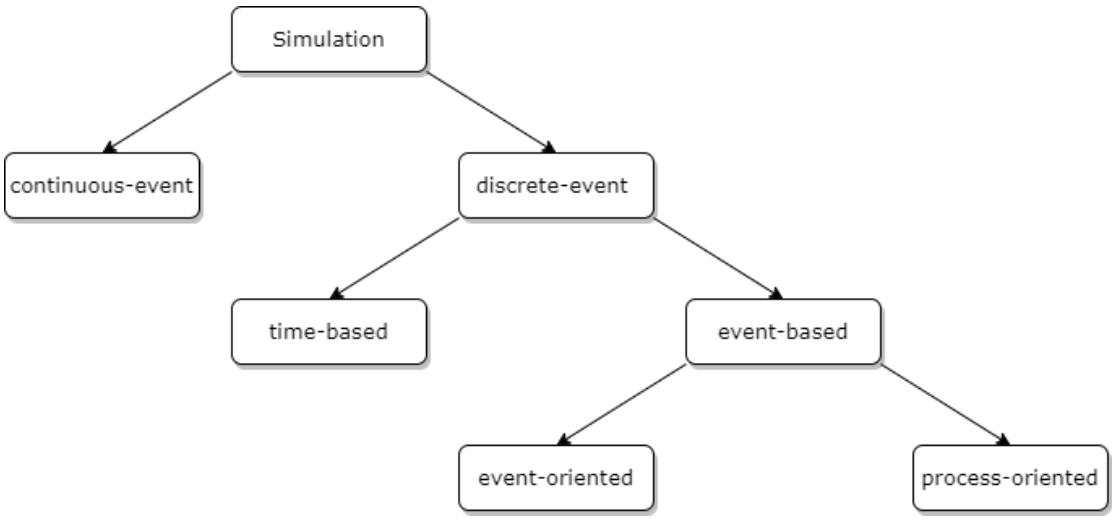


Figure 2.2. Classifying simulations. Source: ([31], Figure 18.1, p.412)

In general, simulations are divided into two categories: discrete and continuous. In continuous event simulations, the state changes continuously. It depends on time, while in discrete simulations, events occur in discrete time jumps, and the system's state does not change between these jumps. The distinction between continuous and discrete systems applies to real dynamical systems and their simulations. Discrete systems are systems in which events are not bound to time and occur in discrete time jumps [31], [32].

In [31], two types of discrete event simulations are defined: time-based and event-based. Time-based simulation (also called synchronous simulation) involves fixed times in which an indefinite number of events can occur in each unit of time. In contrast, event-based simulation (asynchronous simulation) involves time jumps. This means that in each time jump (a specific time point), an event occurs, and each event leads to the occurrence of the next event. Figure 2.3 shows the main steps of a discrete event-based simulation.

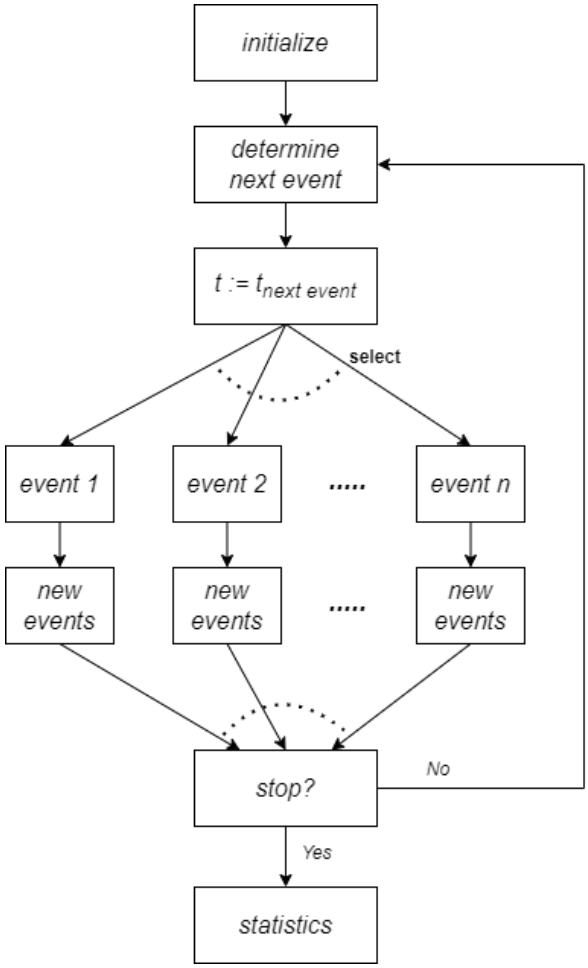


Figure 2.3. Diagram of the actions to be taken in an event-based simulation.
Source: ([31], Figure 18.4, p.416)

2.5 Literature Review

In these two studies [28] and [33], multiple IoT simulators were thoroughly evaluated and compared from different aspects. The results of these two works will assist researchers and developers in choosing an appropriate simulator. Reference [28] compares and evaluates twenty-six simulators, emulators, and testbeds for the IoT (e.g., IoTSim, IoTIFY, iFogSim, NetSim, Bevywise, and FIT IoT-LAB), and Reference [33] reviews and analyzes IoT simulation tools such as Tossim, Cooja Simulator, NS-3, IoTify, NetSim, J-Sim, OMNeT, and Bevywise.

In this master thesis [34], Belchior implements a simulation of the IoT based on the advantages of a visual programming language (VPL) used to simulate an IoT system with physical and virtual devices. In conducting the literature review for this thesis, he extracted 1400 publications from Compendex, Scopus, and IEEEXplore on simulation and the Internet of Things and analyzed twenty of them in detail.

Based on the analysis and comparisons made in the previous works, some practical simulators are reviewed here in detail.

- **IoTIFY** is a commercial cloud-based simulator that tests application functionality and performance and offers a scalable IoT device simulation. It supports popular default protocols such as MQTT, HTTP, CoAP, AMQP and LWM2M. Users can create with IoTify realistic device models with some custom parameters such as sensors, energy and battery. A big advantage of IoTify is rapid development and testing for a huge scope establishment. Some disadvantages are that it can be costly to determine how one element affects another, to make the underlying assumptions, and build the model itself ([28], [33], [34] and [35]).
- **Bevywise** is an enterprise and commercial GUI-based simulator used to test the MQTT and IoT applications to determine the performance level of these applications. It can simulate thousands of MQTT clients to test MQTT applications. It is also configurable to send real-time and dynamic messages in flat text or JSON format or store them in a database. The user can connect Bevywise to clouds such as Azure IoT Hub, AWS IoT Core, or any other MQTT server. A functional feature of Bevywise is its extensibility to implement custom algorithms using Python. Three disadvantages are that more memory is needed, anomalies and analysis take more time ([28], [33] and [36]).
- **OMNeT++** is an open-source discrete event simulator for communication networks, perceptual networks and distributed systems. As defined on the official website [37], “OMNeT++ is an extensible, modular, component-based C++ simulation library and framework, mainly for building network simulators”. It actually provides a simulation IDE based on the Eclipse platform; therefore, the user can take some advantages, such as the extension by a C++ editor and the GIT integration. In addition to the open-source plug-ins, there is a commercial version (OMNEST) with support. The free version is sufficient for educational purposes ([28], [33], [34] and [37]).
- **iFogSim** is an extension of CloudSim for modeling Edge and Fog environments. Since Fog and Edge computing have been introduced as solutions to the data processing problem in IoT and accelerate the processing time between Cloud and devices, iFogSim has been developed to simulate infrastructure with similar characteristics to Fog. Hence, iFogSim can simulate IoT, Edge, and Fog computing environments. However, an important point about iFogSim is that it does not support direct communication between two devices at the same level since Fog devices are assumed to have a hierarchical organization ([28], [34] and [38]).
- **IoTSim** has extended the functionality of CloudSim to simulate multiple IoT applications and big data processing through the MapReduce in cloud-based environments. MapReduce is a method developed by Google and is used as a distributed parallel computing framework for rapidly processing huge amounts of both structured and unstructured data. IoTSim supports Big Data processing systems like MapReduce to help IoT researchers and developers to evaluate the performance of IoT-based applications ([28] and [39]).

StreetlightSim is an open-source simulation introduced in [40]. This simulator can be used to model street lighting systems and evaluate their performance in terms of energy efficiency and benefits. This simulator is also expanded by combining OMNeT++ and SUMO tools. Dizon and Pranggono also used StreetlightSim to analyze different models of

smart streetlights and to represent different lighting schemes. The results of the simulations illustrate reliable data of the time-based schemes in StreetlightSim [41]. Although it is only used for street lighting and cannot be used for parking lots, the review of these publications provides a good overview that is appropriate for developing the project and its future.

In this master thesis [42], Pahr defines the Smart City and explains the measures that can facilitate life in a city. In addition, He also developed a simulation with a smartphone application as part of his project to demonstrate “how data on a smartphone are collected and autonomously driving shuttles are calculating the most efficient way through the city traffic on their own”. This tool aims to illustrate the impact of IoT technology on improving traffic, using resources more efficiently, and increasing the safety of a Smart City .

In this work [32], Carina Pilch has implemented an event-based simulator for modeling and evaluating hybrid Petri nets with random variables. The concepts presented in the fundamentals chapter and the structure of the work were among the most important points that attracted our attention. In addition, Pilch used the SSJ library implemented at the Université de Montréal to generate the random numbers.

Here, the question arises why we need to implement a new simulator instead of using existing simulators. Indeed, according to the goals stated in section 1.2, the simulators presented here cannot meet all of our goals. Considering this, we decided to design and implement a simulation to achieve these goals.

3. Development of the Simulator

This chapter presents a technical solution to the problem raised using a software engineering process. First, section 3.1 describes the preparatory works before the implementation, which are useful for a better understanding of the solution, such as the definition of the requirements, the implementation environment, and an overview of the system. Then, section 3.2 explains the used dataset, which is required to better understand the smart parking environment and develop a model. Next, section 3.3 describes software architecture, and finally, in section 0, the most important features of the software and its main parts are described.

3.1 Preparatory work

Before explaining the methods and steps for developing the simulator, it is necessary to define the basic requirements for software development, explain the programming environment, and provide an overview of the system and work process.

3.1.1 Requirements

According to the MoSCoW definition explained in [43], this method is used for project management to prioritize project requirements under four rules based on their importance and impact. In Appendix A, there is a list of all functional requirements for the simulator application.

The requirements in the MUST criteria include items that are completely mandatory for the program. SHOULD contains requirements necessary to obtain the appropriate software. COULD contains requirements that are desirable but not mandatory. In addition, the fourth criterion of the MoSCoW method is WON'T, i.e., the functions that have not been implemented in the current work but are desirable in the future. Some of the WON'T requirements are explained in detail in section 5.2.

3.1.2 Environment

This simulator has been developed with Java 11 using the IDE IntelliJ Idea on a 64-Bit Windows 10. The following technologies are also used for the development of this tool:

- Spring Boot 2.5.8
- Vaadin 14.8.4
- Maven 3.6.3
- Hibernate 5.6.9
- MySQL 8.0.29 (or MariaDB 10.6.7)

Spring Boot is an open-source framework that reduces the complexity of Java programming based on the principle of “convention before configuration”. Vaadin is a modern technology for Java web application development. Vaadin combines Web 2.0 experience with the

stability and performance of the Java platform. Since 2007, this framework has been open source under the permissive Apache License, version 2.0, except for the documentation, which is licensed under the Creative Commons CC-BY-ND 2.0 [44]. Since the development of this software required Pro components of Vaadin, such as charts, the Vaadin Prime subscription under an educational license was used. However, it is planned to implement these components or the whole frontend part of the simulator in the future using another frontend technology like ReactJS.

3.1.3 System Overview

An overview of the overall structure of IoT projects is shown in Figure 3.1. In such projects, the applications are usually located in the Cloud or the Datacenter, and the IoT devices or sensors are directly or indirectly connected. New technologies such as Edge and Fog computing reduce client-server communication latency and server processing time. As shown in Figure 3.1, IoT devices are connected to the Edge nodes, which are usually local servers or devices, and then they interact with servers at the Fog layer. Section 2.3 explains the Edge and Fog computings and their differences.

This work aims to design a discrete event simulator to replace IoT devices in the sensing layer, as shown in the figure. By doing so, not only can the smart applications in the Cloud layer be tested, but also the capability of Edge and Fog nodes without incurring high costs.

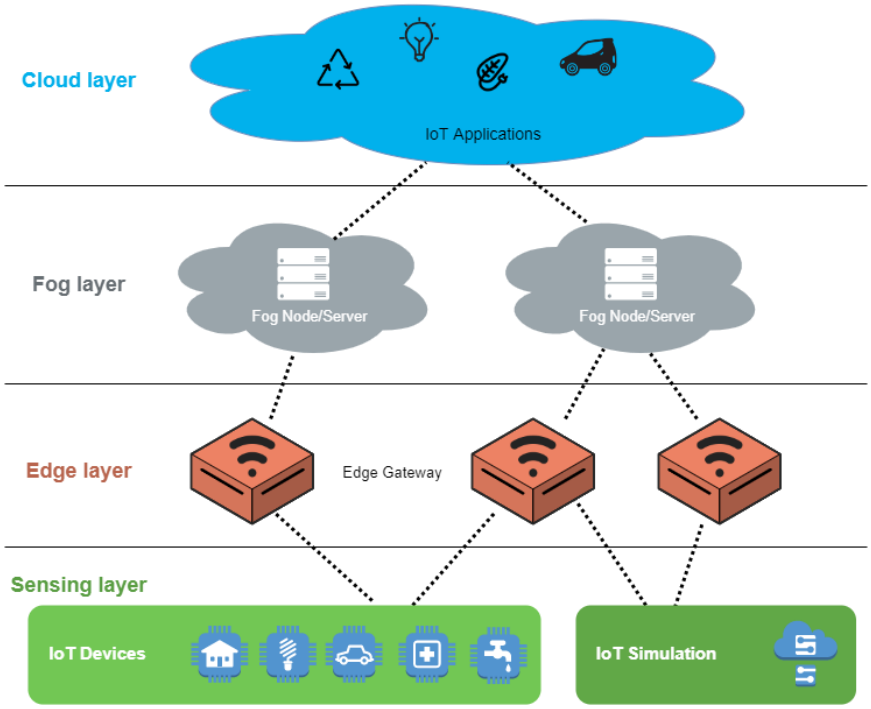


Figure 3.1. IoT System Overview

3.1.4 High-level process flow

The high-level simulation process flow is shown in Figure 3.2, which includes all stages for the simulation of IoT devices. The main stages of this simulation project are discussed below.

Problem Definition:

The definition of the problem is very important because it helps obtain favorable results. In this case the problem definition may involve, for example, analyzing the utilization of an existing parking lot or the impact of constructing a new parking lot on traffic volumes in the area. It is important to note that the original goal of this project is to simulate smart parking lots. However, this problem-solving method can be used in the same way to simulate other IoT devices, such as streetlights.

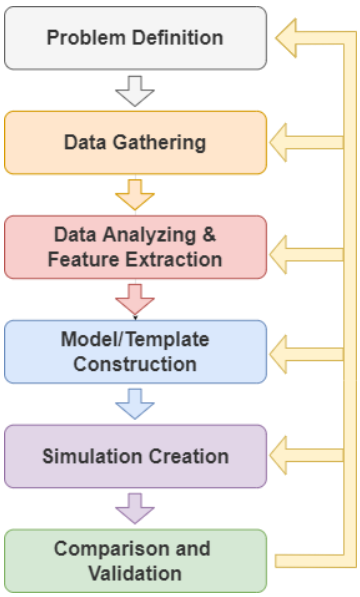


Figure 3.2. High-level process flow

Data Gathering:

At this stage, data can be collected using various methods, such as sampling with physical sensors or using existing datasets with similar characteristics. In this project, data from the city of Melbourne, Australia, collected in 2019 using on-street parking sensors, was used [45].

Data Analyzing and Feature Extraction:

In order to build an accurate model, it is necessary to have a good understanding of the data. In this stage, we can explore and extract these features by using different data preprocessing techniques. In this project, Google Colab and Python language were used to analyze the data.

Model Construction:

This means designing a model for simulation using the information obtained in the previous stage. One of the objectives of this master's thesis is to develop a smart parking model that enables the user to simulate a parking lot using the parameters derived from data analysis.

Simulation Creation:

The next step is implementing a simulator or using an application to simulate a defined model. The main task of this work is to implement an IoT simulation application. Thus, when the user defines a model or creates a template, he can configure and simulate it. Finally, the generated data is logged in a database or transferred to a server.

Comparison and Validation:

The last step is to analyze the results and compare them with the original data (if available). Any of the previous steps can be repeated afterward to make corrections.

3.2 Dataset

One way to gain knowledge about the requirements of a Smart Parking lot is to review and analyze existing datasets. For this purpose, only one of several datasets belonging to the City of Melbourne [45] was selected for this work. In addition, on the one hand, measures were taken on these data to obtain sufficient knowledge for implementing the simulator algorithm; on the other hand, it was used for modeling the parking lot with the simulator. In the following, this dataset is presented, and the analysis and processing steps are described.

However, we believe it is necessary to mention another dataset from the City of Frankfurt am Main that was used for the first version of this project [46]. Initially, very useful information was obtained from the dataset of parking lots in Frankfurt am Main. However, since the data format of the City of Melbourne is very similar to this work, and there is no access to the new version of the data of Frankfurt am Main, it was decided to use the data of the City of Melbourne.

3.2.1 Data description

We worked on a dataset of on-street parking spots captured by in-ground parking sensors. The dataset is taken from the open data portal of the city of Melbourne in Australia [45]. This data includes approximately 42.7 million parking events in 37 areas (123 streets) of Melbourne in 2019. Of course, due to the execution timing of this project, it was possible to use the 2020 and 2021 data. However, due to the COVID-19 pandemic and successive lockdowns in those years in Australia, there was a possibility of irregularities and unforeseen changes at different times. Therefore, the 2019 data seemed more appropriate for this project in all respects.

The dataset contains 20 attributes, some of which are important for this work, such as *DeviceId*, *ArrivalTime*, *DepartureTime*, *DurationMinutes*, and *VehiclePresent*. *DeviceId* is a unique number for each IoT device. Arrival and departure times indicate the start and end times of a parking event for the vehicle. They are used not only for occupied spots but also for vacant spots, and *VehiclePresent* indicates the status of a parking spot. Finally, *DurationMinutes* is a specific amount of time between two events in minutes, which is calculated by subtracting the departure time minus the arrival time, and from now on is called “duration time”.

3.2.2 Analysis

Due to a large amount of data and simplifying data analysis, the data was limited to only Queen Street, which included 2.3 million events for 227 parking spots in 2019. After analyzing all the data and understanding the key features, the data volume was again narrowed down to August 2019. This was also the easiest for simulation and comparing the generated data with the original data. In this step, the number of records was reduced to 220292 records, which seems to be an acceptable value for comparison. The Google Colab is the most important tool used in this project for data analysis.

After the data cleaning step, the next step of preprocessing was to select a statistical distribution that best fits the data set. To determine the goodness of fit, the Python package Distfit is used, which, as described on the library's website, can determine the best fit among the 89 theoretical distributions using the Residual Sum of Squares (RSS) and returning the best-fit theoretical distribution with the parameters loc, scale and arg [47] (see Appendix F).

Figure 3.3 and Figure 3.4 show the estimation results of the *distfit* function. Based on these results and the shape of the graph in Figure 3.4, we can conclude that the selected data fit the exponential distribution better than the other statistical distributions. This experiment was also performed for empirical data using the *distfit* method and the Kolmogorov-Smirnov test (K-S) and yielded similar results. The results of this experiment can be found in Appendix C.

	distr	score	LLE	loc	scale	arg
0	expon	0.000009	NaN	0.0	31.985456	()
1	gamma	0.000115	NaN	-0.0	175.210115	(0.34240852592845084,)
2	dweibull	0.00014	NaN	1.0	21.28465	(0.7346479499791447,)
3	beta	0.00015	NaN	-0.0	6070.292102	(0.34398564031160805, 300.00706088173865)
4	norm	0.00017	NaN	31.985456	68.426721	()
5	loggamma	0.000186	NaN	-10320.819252	1681.200159	(472.99410793361085,)
6	pareto	0.000191	NaN	-0.808938	0.808938	(0.4343496282554167,)
7	genextreme	0.000258	NaN	2.090289	14.247144	(-6.815865139618332,)
8	lognorm	0.000267	NaN	-0.0	2.323414	(8.532457678850543,)
9	t	0.000267	NaN	1.012107	1.287194	(0.4009850082790162,)
10	uniform	0.000322	NaN	0.0	1080.0	()

Figure 3.3. The result of the *distfit* method using RSS as the scoring statistic

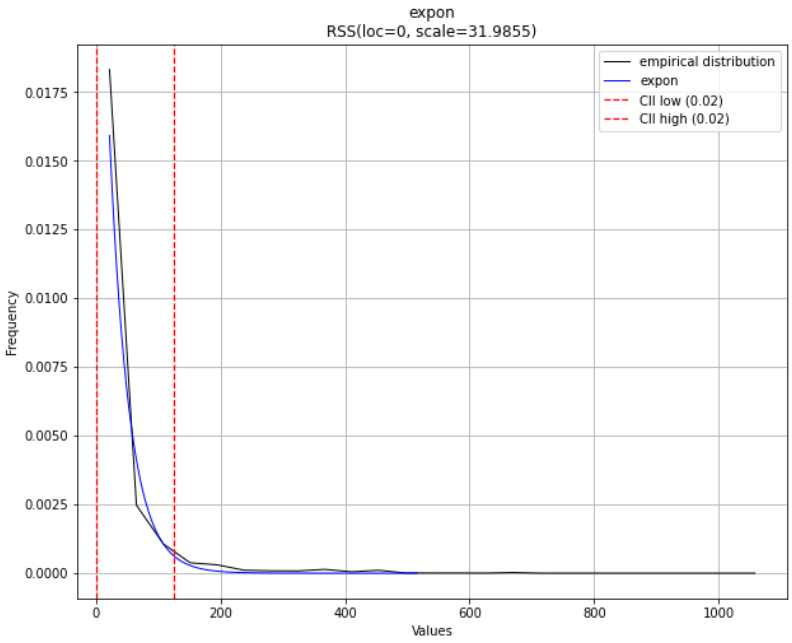


Figure 3.4. Indication of the fit of the empirical data to the exponential distribution

Based on the data distribution, essential parameters such as the mean value can be determined. These parameters are needed to generate random data in the simulator. The most important parameters of the exponential distribution are scale and location. An exponential distribution is defined by the location parameter, which is the lowest possible value, and the scale parameter, which represents the mean time until the event occurs.

The average time can be determined for the entire data or in relation to the events during the day and night. To divide the data into day and night categories, two-time points are required to determine the start of night or darkness and the start of day or daylight. In this project, 7 am and 5:30 pm are considered. These two times are the average times for sunrise and sunset in August 2019 in Melbourne.

Then it is time to calculate the mean time. For this purpose, the data were converted into different shapes, and the mean value was calculated for each data shape. In the following table, the mean time for the duration time (*DurationMinutes* attribute) of the data set is shown not only for the entire data but also for events during the day and night and based on weekdays, weekends and working days.

Dataset	Mon.	Tue.	Wed.	Thu.	Fri.	Sat.	Sun.	Weekend	Workday
All	31.95	31.49	32.18	30.13	30.36	31.43	38.90	34.33	31.11
Days	21.29	21.64	22.69	21.25	21.02	22.60	26.60	24.20	21.52
Nights	52.23	49.24	46.61	45.22	46.45	45.30	60.77	51.03	47.63

Table 3-1. Average duration time of events on weekdays, weekends and workdays

3.3 Design

Software design is a software process used to implement a solution. Software design can be used to avoid the complications and risks that software projects usually have [48].

3.3.1 Simulation Architecture

To solve the problems mentioned in chapter 1, especially to achieve the goals mentioned in section 1.2, an attempt was made to construct an architecture that is as simple and scalable as possible to cover the various goals in the field of IoT device simulation. As seen in Figure 3.5, it is possible to run several simulators in parallel. One or more templates can also be modeled in each simulator. As shown below in the figure, each template is related to an IoT environment. A template represents the characteristics of an environment and the smart devices within that environment. For example, a template can be defined for a parking lot that displays information about that parking lot, including capacity and the average time cars are parked there. The template can also be used to define information about a street, such as the number of streetlights, the distance between them, and the speed limit on that street, to simulate a smart street in the corresponding template.

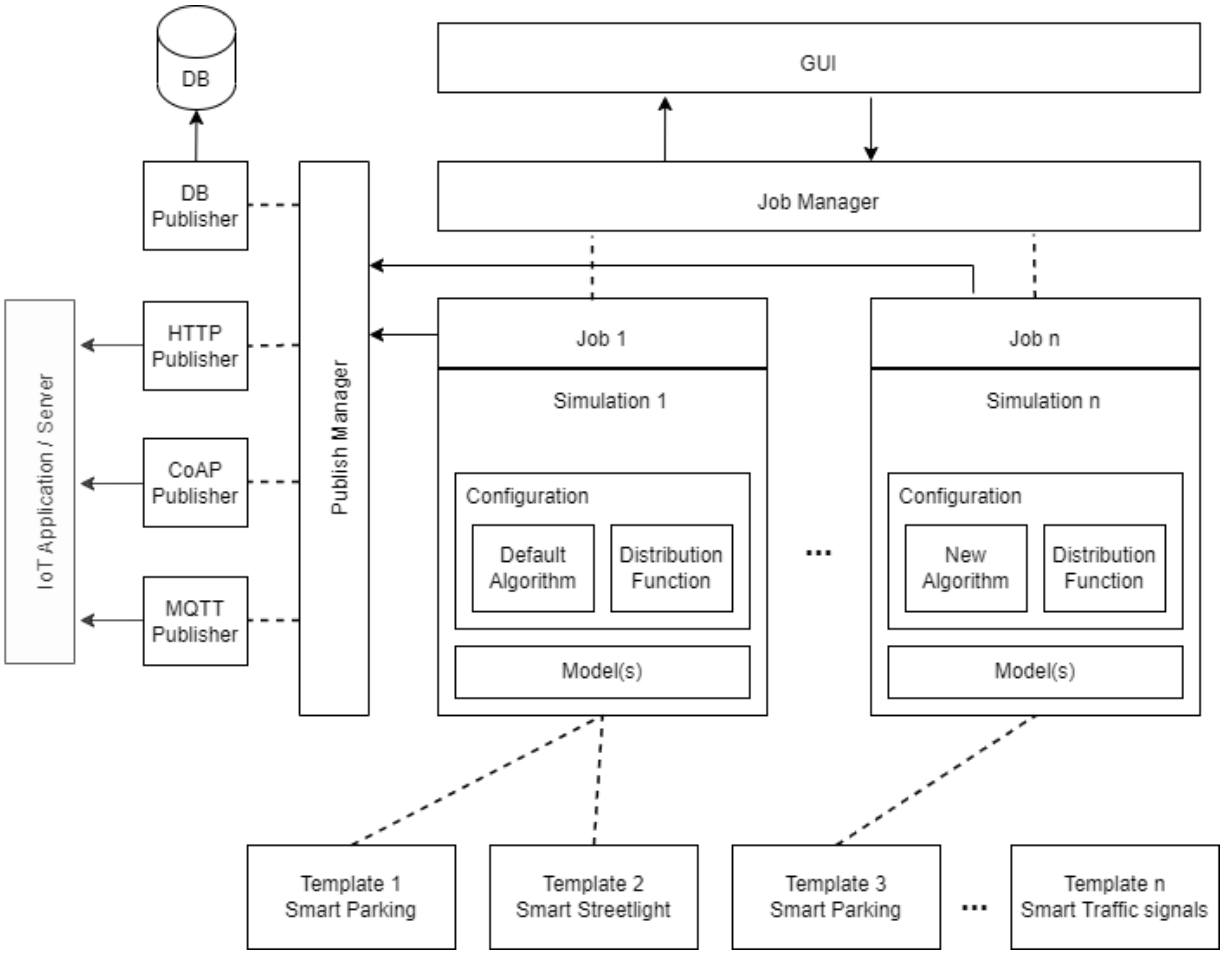


Figure 3.5. The architecture of the IoT simulator application

Moreover, each simulation can be configured independently. The simulation configurations refer to the functionality of a simulator, including the algorithm and the type of distribution function or the operating speed of the simulator. These details are sufficiently explained later.

Finally, each simulator is executed by a job. This can be interpreted as Jobs running the specified algorithm according to the simulator configuration to generate data based on the template definition. The Job Manager is responsible for controlling jobs and managing them through the GUI. Each job transmits simulation results to the publish manager, who determines where and how to transmit and store the data based on the simulator configurations.

3.3.2 Packages and Classes

A package diagram is used to represent the system elements at a high level and illustrate the simulator's structure. The implementation of the simulator is divided into five main packages, as shown in the package diagram in Figure 3.6. Note that only the most important packages and classes are shown in this section.

The View package contains the classes and components for the graphical user interfaces, which are built based on web standards and specifications of the Vaadin framework. The Component package includes the customized components for Vaadin classes used in View. The Data package refers to the data layer of the simulator and contains four sub-packages, as shown in Figure 3.6 (b). Figure 3.7 shows a general UML class diagram of this package.

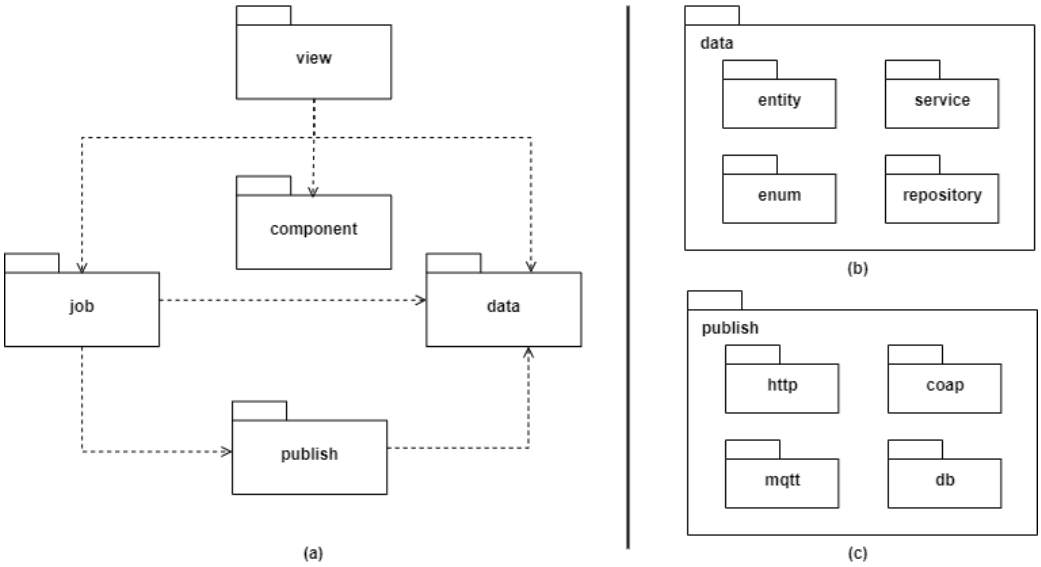


Figure 3.6. Structure of all packages. (a) main packages, (b) sub-packages of the data, (c) sub-packages of the publish.

The implementation of the main algorithm of the simulator is in the job package, and its details are explained in the continuation of this section and the next section. The publish package is also related to transmitting or storing the generated data in a database or server.

As explained earlier, this server can be a cloud server or an edge node. Figure 3.6 (c) illustrates that this package includes four sub-packages to support different data transmission protocols and data storage.

Although this project was developed in the first step for the simulation of parking lots, one of the goals of this work is to make the program extensible to simulate other IoT devices and environments. In the following, this capability is explained in more detail using several class diagrams.

One of the most important points in Figure 3.7 is the independence of the information units from each other. There is also confidence that the data structure is scalable and can grow. The information about the parking template is stored in the corresponding table, *ParkingLot*. According to the definition information, each parking lot includes some parking spots. All factors that depend on time are stored in the *TimeBasedData* table, regardless of which template they belong to. The data generated by the simulator for the parking model is stored in a *ParkingRecord* table (simulator output). Information about how to simulate and where and how to store the data is stored in the *Simulation* table. Also, the jobs, which are the executable part of the simulator, are stored independently in the *Job* table. Therefore, different jobs can be executed for one simulation.

Assuming that the next project in the development of this program is related to street lighting, you can create a new table for that template, link it to the *Simulation* table, and store the generated data in an independent table, for example, named *StreetLightRecord*.

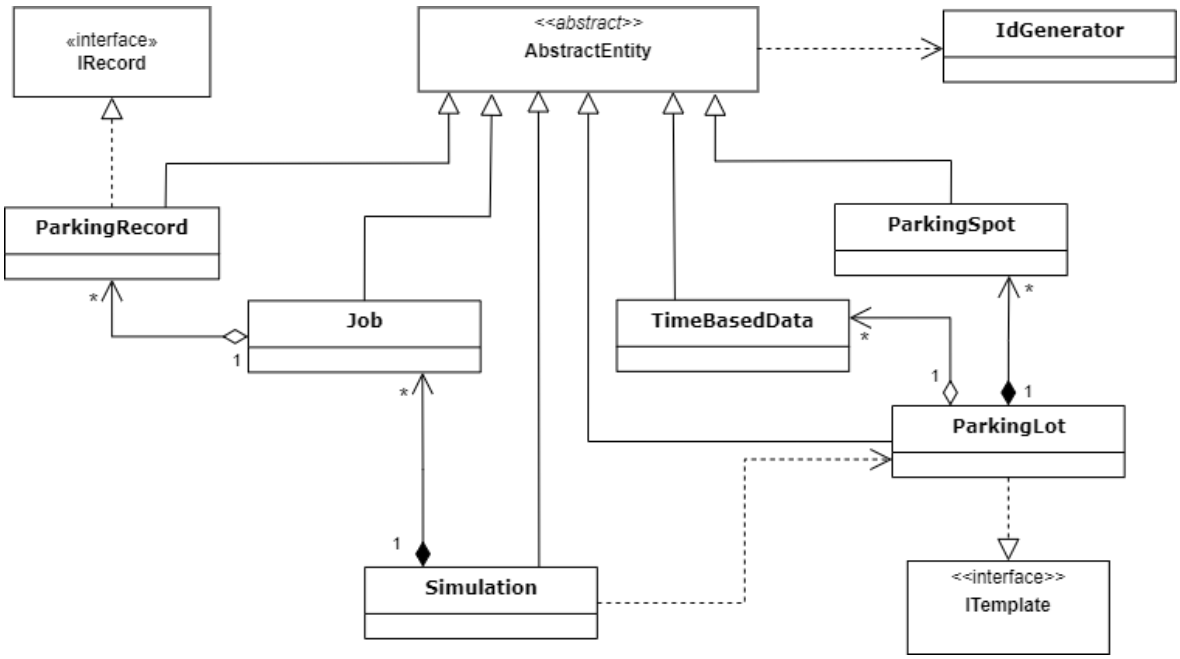


Figure 3.7. Structure of the package data

3.4 Implementation

Once the design process is complete, the simulation software can be implemented in Java. In this section, the details of the solution and the main features of the simulator are comprehensively examined, especially the implementation code of the simulator algorithm located in the job package. Also, it explains the random number generator, data transmission to the server, time in the simulator and the graphical user interface. Furthermore, two diagrams were drawn to show the software's main structure and dynamic aspect, which helps to understand the subject. Figure 3.8 demonstrates the structure of the job package and its connections with the user interface and the publish package, which is responsible for storing and transferring data. Figure 3.9 is an activity diagram illustrating how the parking simulator algorithm works from the moment the user clicks on the "*Start*" button until it is stopped.

3.4.1 Random Number Generation

In the simulation, random numbers distributed according to certain probability distributions are needed. These random numbers are obtained by different methods depending on a specific distribution. This simulator is not exclusive and needs random number generators for different purposes. For example, in the parking simulator, random numbers are used to determine the time intervals (duration) between discrete events. Each event represents a change in the status of a parking spot. In this way, it is determined when each event occurs. For example, if a random number of 15 minutes is determined for parking spot 101, this parking spot must change from vacant to occupied or vice versa in the next 15 minutes.

Therefore, this project utilizes the distribution package of the Apache Commons Math library [49] in Java, which provides a set of random variable generators for different distributions. Appendix B gives an overview of these distributions and their parameters.

3.4.2 Job Package

The Job package is intended for managing jobs and implementing various algorithms of the simulator software but currently contains three classes. The *ScheduledJob* class is an abstract class for creating, managing and scheduling jobs. In this abstract class, not only a syntax for applying special functions in a job has been defined, but also some practical methods have been implemented, including a set of random variable generators for various distributions that determine the next discrete event. In addition, a schedule method is used to execute complex tasks of a job in the background without interrupting the work of the main program. According to the value obtained through the random generator, this method schedules the next execution of the *ParkingTimerTask* (as a *Runnable* class), which is responsible for controlling and changing the state of the spots.

JobManager is a component for managing jobs. As shown in Figure 3.9 (a), after starting a job, i.e., when the user starts a job from the user interface, the corresponding *Job* entity is submitted to the *JobManager*, which maps this *Job* entity to a *ScheduledJob*. From this moment on, the simulation of the selected model for this job begins. Also, when the job is

stopped, the desired job is selected and stopped by the *JobManager*. The *JobManager* has a simple structure and only ensures that there is only one *ScheduledJob* for a job. Therefore, an important task of this component is to map the *Job* entity to the correct instance of *ScheduledJob*. This means that it selects one of the implementations of the *ScheduledJob* class depending on the selected model for the simulation.

The only instance of the abstract *ScheduledJob* class in this project is the *ScheduledParkingJob* class. Here two *start()* and *stop()* methods are implemented from the parent class. Their name describes their function well. At the start, some spots are created and initialized according to the capacity determined in the parking model. Then, using one of the random number generator methods in the parent class, some random numbers are generated and set on the created spots. After that, the smallest generated random number is selected and based on it; a *ParkingTimerTask* is scheduled for this time, an instance of the *TimerTask* class in Java. Part of the implementation of the parking algorithm is in this class. When *ParkingTimerTask* is executed, all the spots are checked, and a new time is randomly generated for the spots whose duration time has passed. While checking the spots, the shortest remaining time for the next state change is selected, and an instance of *ParkingTimerTask* is created again and scheduled for that time. The advantage of this method is that there is no need to create independent threads for all spots, which is expensive in terms of performance. Although all spots are interdependent and located in one thread at first glance, in practice, they are completely independent of each other and only their management is carried out in a single thread.

A running job can be stopped in two ways, either manually by the user or automatically. In the manual variant, the user stops the job obviously via the user interface. In the second variant, the user can specify the end time of a job, and as shown in Figure 3.9 (a), *ParkingTimerTask* checks at the end of its work if this time is set and the time has come, then stops the job. Part (b) of Figure 3.9 shows the process of stopping a job. As mentioned above, the *JobManager* calls the *stop()* method of running *ScheduledJob*, which is done only if a *ScheduledJob* has already been mapped to a *Job* entity and is running. The *stop()* method shuts down the *ExecutorService* with a timeout of 10 seconds and terminates the operation. *ExecutorService* is used to concurrently execute some tasks (*ParkingTimerTask* in this case).

3.4.3 Publish Package

The parking simulator contains discrete events. Each time the status of a parking lot changes, an event is triggered, and when each event occurs, a record can simultaneously be stored in the database or sent to a server. This record includes the start, end, and duration time between these two and the previous status.

As you can see in the structure of the publish package in Figure 3.8, *ScheduledParkingJob* uses a *ParkingPublisher* to manage data publishing. This class is a manager and decides where and how to store the data according to the user's settings.

Three publishers are implemented to transfer records to the server using one of three protocols: HTTP, MQTT, or CoAP. The only JSON format is supported when writing this thesis for transferring data to the server. The *DbPublisher* is only used to store the records

in the local database (MySQL). The idea of extending this class with the possibility of storing data in different databases like MongoDB is not far-fetched.

Also, a *PublishManager* can be used here instead of *ParkingPublisher* for all kinds of simulations. However, since *ScheduledParkingJob* knows its target, it was preferred that each job use its publisher directly. However, this form of implementation could be changed in the future.

3.4.4 Simulation Time

Two features of the simulator are described here. The first feature is the time unit, and the second one is the start and end time of the simulation.

TimeUnit

The time unit allows the user to increase or decrease the tempo of simulation and data generation. This parameter determines how much real-time passes with each simulation time step. By default, its value equals 1, so the speed is the same as the wall clock time. If the time is set to 0.1, each simulation time unit will take only one-tenth of a second, which means that the simulation speed will increase by 90%. If the time is set to 10, the speed will decrease, and each simulation time unit takes ten seconds.

Therefore, this capability can be used depending on the purpose of the simulation. For example, if the real-time simulation is desired, such as when the simulator interacts with real hardware, the value should be set to the default or 1. However, if you only want to generate data, you can increase the simulation speed. In section 4.2, the performance and throughput of the simulator are examined with two different time units

Simulation Start/End Time

The second feature allows simulating data for a specific time in the past or future. In simple words, the user can specify whether the start and end of the simulation should be at the beginning of 2050 or the summer of three years ago. This feature is used in this work. As explained in section 3.2.2, one of the project's goals is to create a dataset close to the City of Melbourne data with similar characteristics. Therefore, this feature is used to generate data for the exact time of the original data, making it easier to compare the data generated by the simulator.

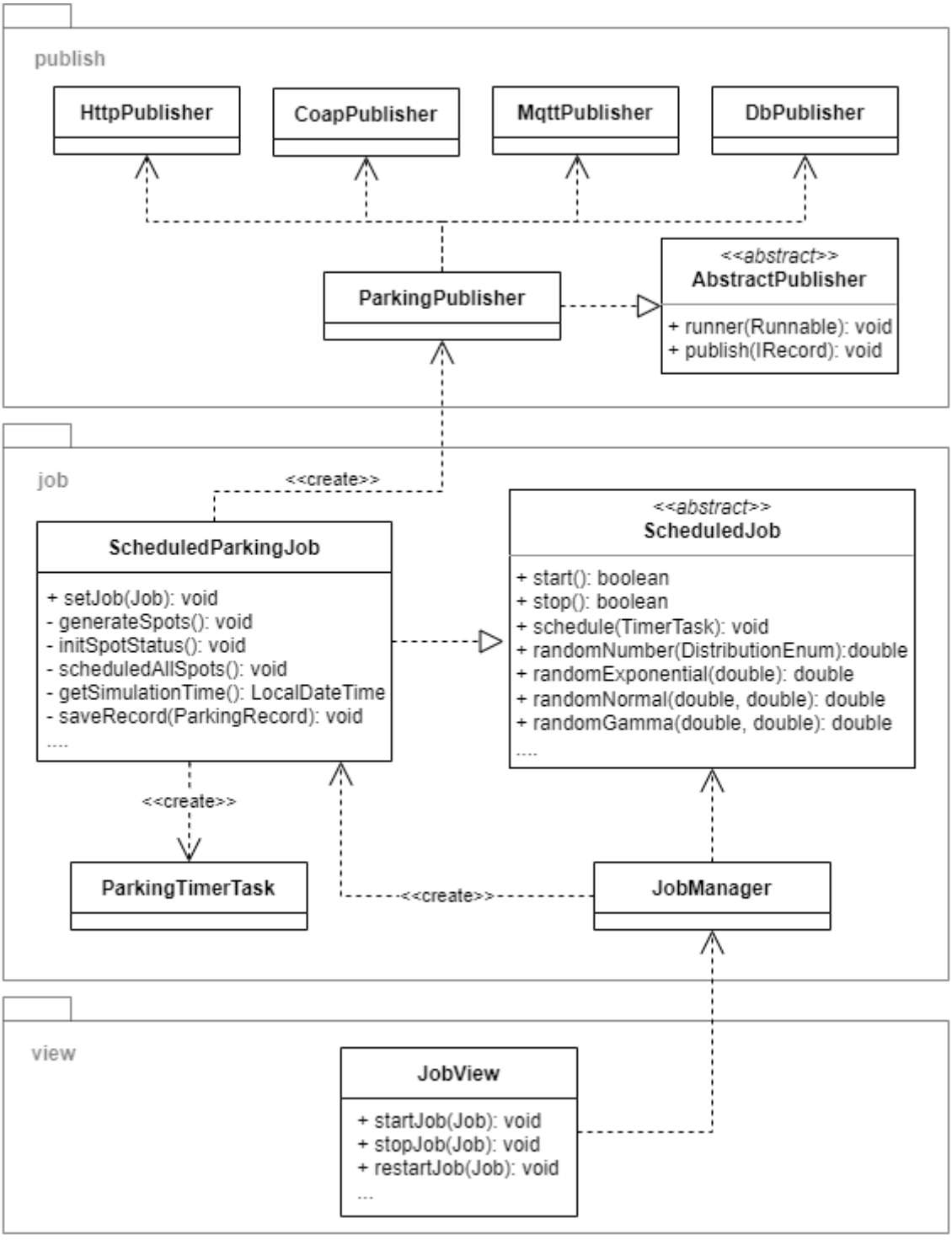


Figure 3.8. Structure of the job and publish packages and their connections

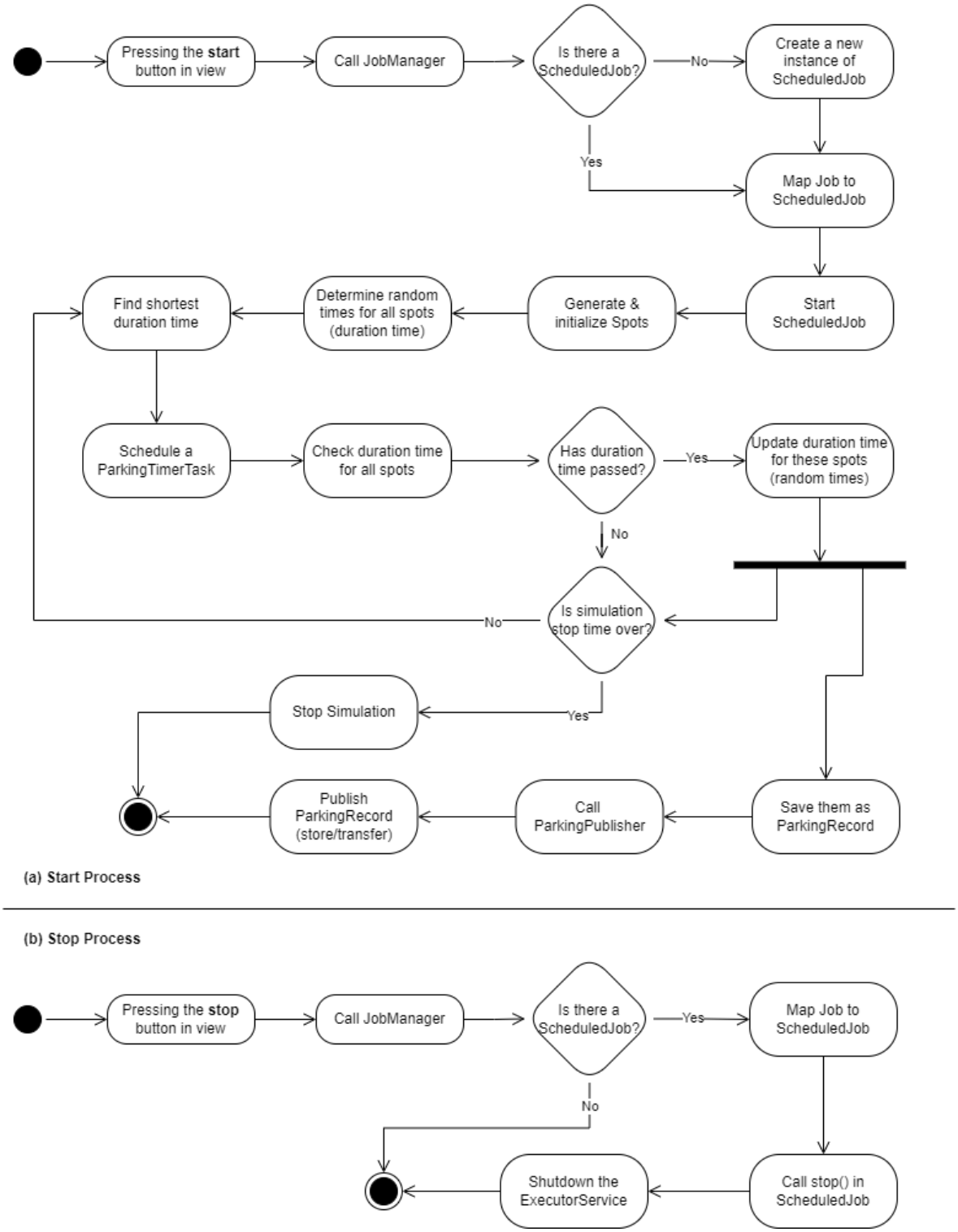


Figure 3.9. Activity Diagram: (a) The process of starting a job, (b) The process of stopping a job

3.4.5 User Interface

The user can work with the simulator through a web-based user interface. An attempt has been made to design and implement a user-friendly and comfortable graphical user interface. In this section, we introduce how to use the software by explaining some practical parts of the software.

Parking Template

The template is a part of the software that allows the user to define his model. Since there is only one template for modeling parking lots in this version, the parameters used to define a parking lot are described here. Figure 3.10 shows the details of the parking template.

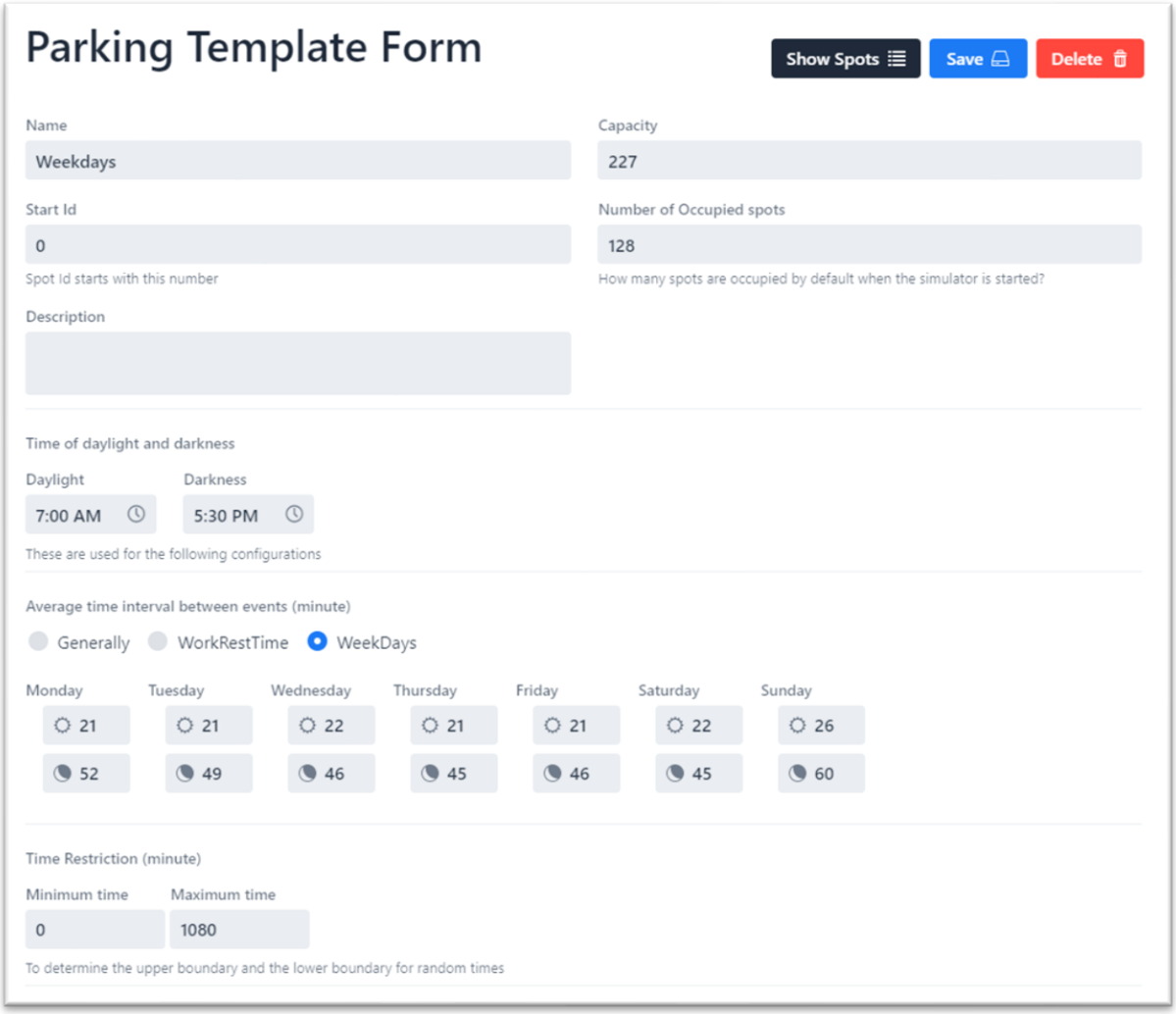


Figure 3.10. User Interface - Parking Template

In addition to the *Name* and *Description* fields, which differentiate the templates from each other, the user must specify the *Capacity* or the number of spots in the parking lot. It is

possible to specify the "*Number of Occupied spots*" in the field, i.e., how many spots are occupied at the beginning of the simulation. Field "*Start Id*" has no operational use and is used only for naming spots as a device ID. Device IDs are identified by prefixing the name of the parking lot and a number that starts with the *start ID* value. For example, according to the information in Figure 3.10, the naming order will be as follows: Weekdays_0, Weekdays_1, ..., Weekdays_226.

The *Daylight* and *Darkness* fields divide the 24 hours of a day and night into two parts. This is to simulate what happens in a parking lot during daylight when there is more traffic and during darkness when cars usually do not move and spend more time in the parking lots.

In the next part, the time interval between events can be estimated as day and night. In this part, there are three different ways to set the time interval. First, as shown in Figure 3.10, each value can be entered for the days of the week, which gives more accurate results. Two other options are also shown in Figure 3.11. The user can also generally enter a number for all days and hours or, if necessary, based on working days and weekends. Averaging time based on holidays can be an attractive option not implemented in this version.

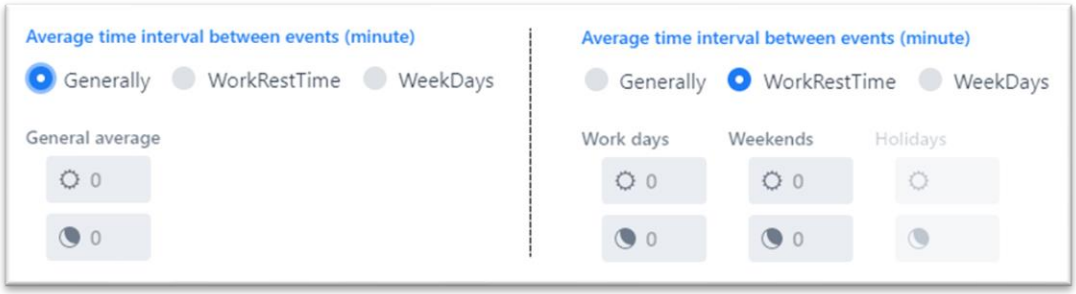


Figure 3.11. User Interface - Average time interval between events

Finally, the *Time Restriction* determines the minimum and maximum thresholds for the random numbers generated based on the time averages given in the previous part.

Simulation Configuration

In this part of the application, the user can configure a simulation. This configuration contains the application's parameters to simulate the selected model in the *Template* field. The *Time Unit* is a field to change the speed of the simulation functionality, which we explained in detail in section 3.4.4. With the *Simulator Algorithm* and *Distribution Function,* the user can choose an algorithm for simulating a model (in this case, a parking lot) and the probability distribution function for generating random numbers. In the current version, only one algorithm is selected by default. The default algorithm is developed in this work and is described in section 0.

In the "*Simulation Start/End Time*" part, you can specify whether the data recording should occur at the current time or at a specific time in the past or future. We have explained the details of this parameter extensively in section 3.4.4. Finally, in the *Scheduling* field, the start time of the simulation is specified. It should be noted that this parameter is

fundamentally different from the previous one, "*Customize time*". Here it is specified that after clicking the *Start* button of a job, this job must be executed immediately at the same time or later after a certain time. While the previous field only defines the time when the events should occur (e.g., the days and hours when the vehicles are parked).

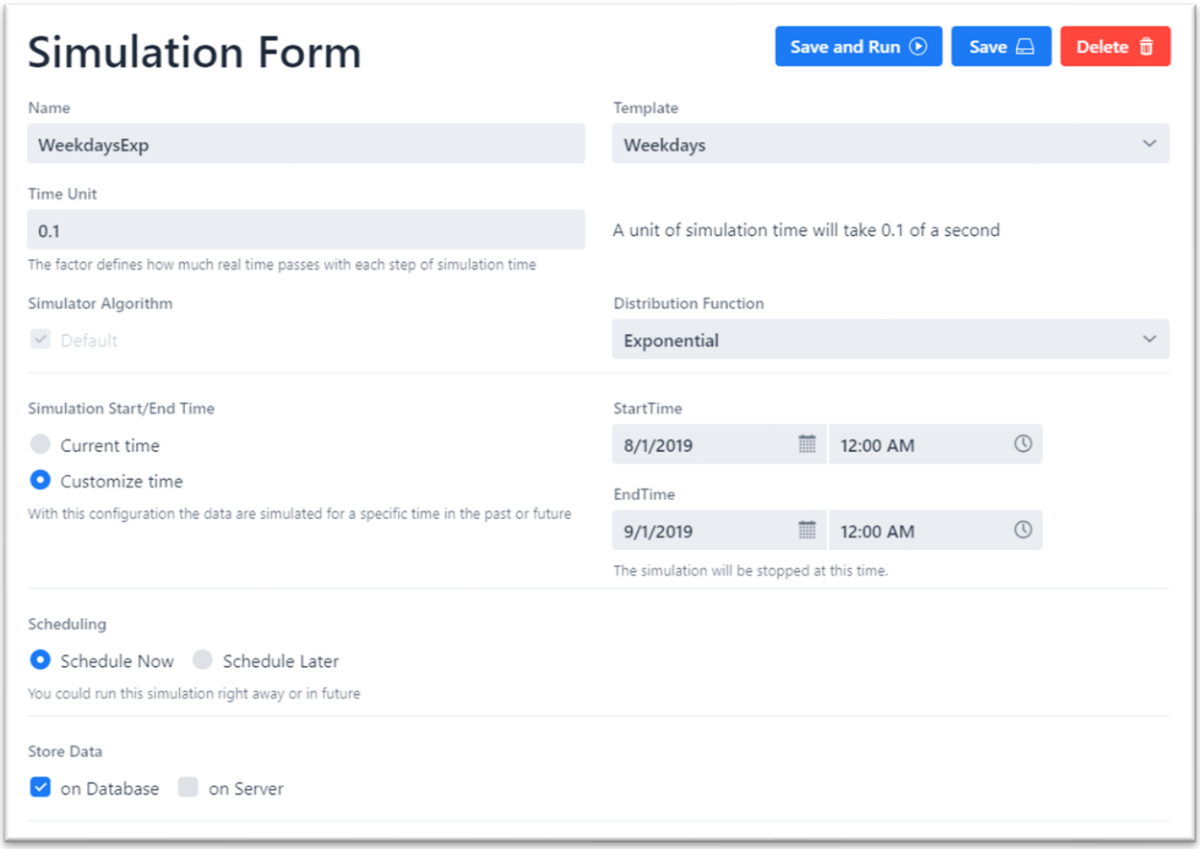


Figure 3.12. User Interface - Simulation Configuration

In the "*Store Data*" part, the method and location of data storage are specified. It is possible to choose both the database and the server options. If you select the "*On Database*" option, the data will be saved in the local database to which the simulator is connected. The "*On Server*" option allows you to specify the information about the target server (Figure 3.13). Therefore, the generated records for each event will be stored on the server.

On Server

Indeed, data transmission to a server means that events are sent to Smart City applications. Therefore, this is an important feature of the simulator. In order to test the usability of this feature, a helper server based on Spring Boot and MongoDB has been developed with the ability to communicate over various protocols such as MQTT, HTTP, and CoAP and with the support of the JSON format and REST API. The obtained results were successful. In this way, the designed IoT simulator can send data related to the event (ParkingRecord) in

JSON data format using one of the selected protocols. The target server will store the received data in the MongoDB database.

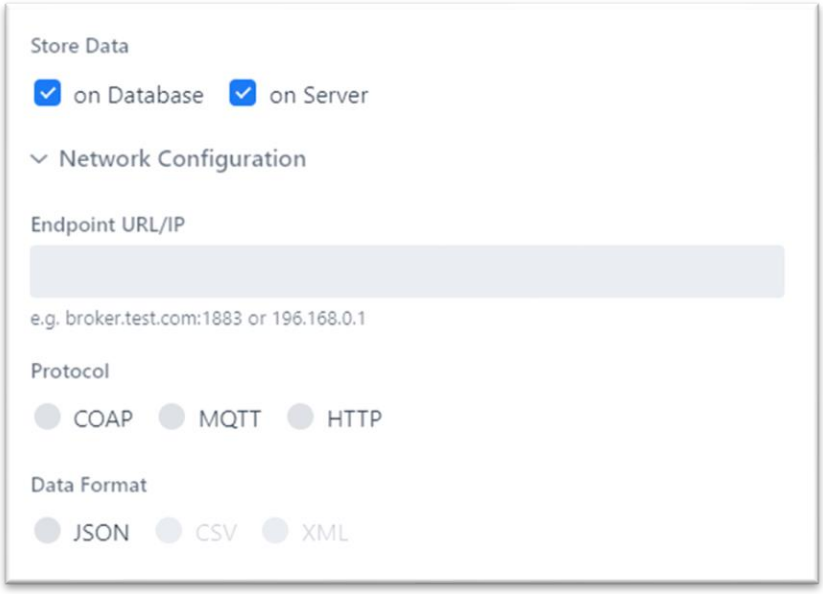


Figure 3.13. User Interface - Store generated records on server

Job List

Each job is intended for only one simulation; however, for one simulation, several different jobs can be executed simultaneously. To run a job, the "*Save and Run*" button must be pressed in the form of a simulation, Figure 3.12 top right. In addition, executed or running jobs can be managed on the Jobs page, Figure 3.14, where you can not only manage the execution of a job using the "*Start*", "*Stop,*" and "*Restart*" buttons but also view the simulation results in the *Explore*.

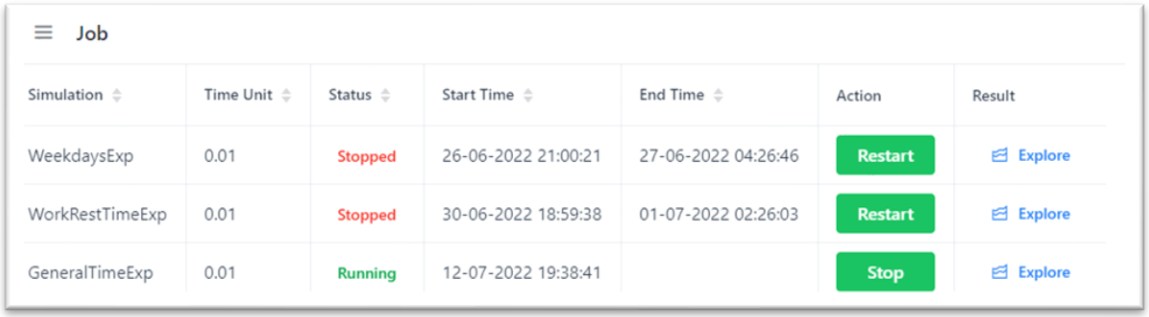


Figure 3.14. User Interface - Job management

Job Report

On the report page for each job, you can view information such as the current status of the job, the start and end time of the job, and the current time spent in the simulation (based on

the time unit and time period selected for the simulation), the number of occupied and vacant spots in the parking lot. It is also possible to review, in the form of three diagrams, the average duration time and the number of requests in different days, weeks, months and years. It is also possible to download data in CSV format. An overview of this page is illustrated in Figure 3.15.

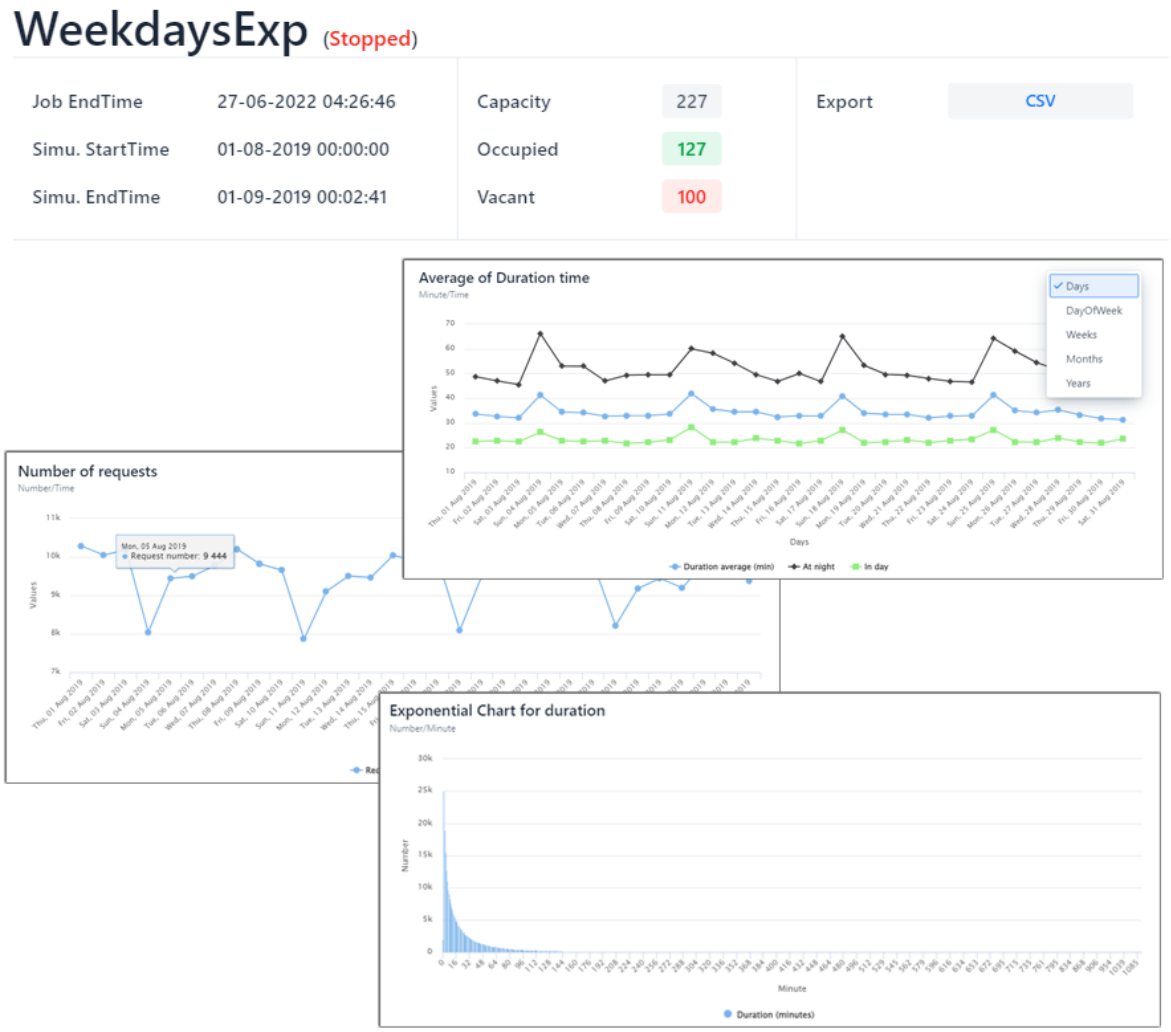


Figure 3.15. User Interface - An overview of the report page

4. Results and Discussion

This chapter deals with the evaluation of the developed solution. In section 4.1, a case study is defined to measure the features described in chapter 3 and the achievement of success in meeting the project objectives in section 1.2. For this purpose, the simulation results were compared with the original data. In the continuation of the second case study in section 4.2, the performance and effectiveness of the software are measured under different conditions. Finally, in section 4.3, we summarize the obtained results.

4.1 Validation study

The first case study is the simulation of on-street parking on Queen Street in Melbourne for August 2019. Based on this, two parking templates were defined and simulated using the average numbers for days and nights of the week identified in Table 3-1; the first template is for weekdays (Simulation1), and the second for weekends and working days (Simulation2). This test aims to simulate the occupancy and request rates in different hours and days and compare them with the original data.

Since the data analysis in section 3.2.2 showed that the original data followed an exponential distribution, an exponential random number generator was also chosen. Thus, two things need to be tested here. First, whether the generated data also fit into the exponential distribution, and second, how close the characteristics of the generated data, such as the mean, are to the real data. For this purpose, we also used some methods such as Distfit, Boxplot and error metrics.

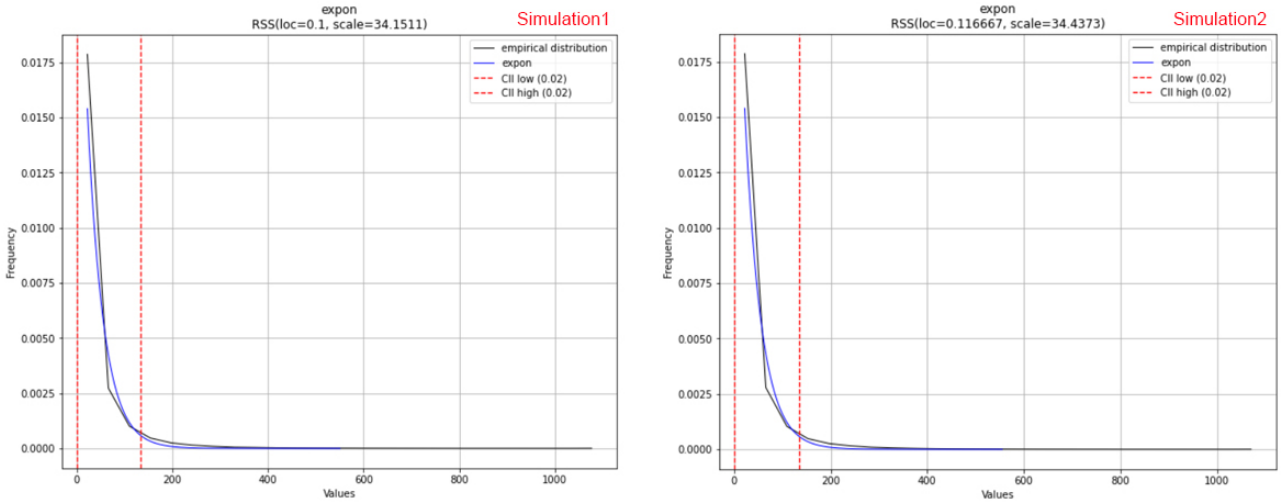


Figure 4.1. Distribution of simulated data fitting the exponential distribution

Figure 4.1 illustrates the results obtained with the *distfit* method for the generated data. According to this, both simulation results fit an exponential distribution. The results of both fitting processes in detail are presented in Appendix D.

The following table also compares the properties of the two simulated models with the original data. The obtained results show that the simulator was able to generate similar data as expected (in this case, the data of Melbourne in August) but do not match completely, which should also be noted. Looking at the comparison in the Table 4-1, the difference between the simulated data and the original data can be seen in two parameters: Count (number of requests) and Mean (average time between two events). Here the difference in the number of generated events is about 75 thousand events higher than in the original data. The second point is the difference in the average of the generated data, which is slightly higher than the average of the original data.

It is assumed that this mean difference depends on the number of requests. Thus, if the number of data generated can be managed and actually limited, an average that is closer to the original data can be generated. The second assumption is that the reason for the higher average time between events is the processing time. A better result will probably be achieved if the existing algorithm can be improved.

Params	Original	Simulation1 (Weekdays)	Simulation2 (Weekends/Workdays)
Count	220292	295371	292734
Mean (Scale)	31.98	34.25	34.55
Location	0	0.1	0.116
Max	1080	1098.55	1092.10
Min	0	0.1	0.116
25%	1	4.4	4.48
50%	5	13.9	14.11
75%	33	37.58	38.23
RSS score (Fit to expon)	0.000009	0.000008	0.000009

Table 4-1. Comparing the properties of the generated data with the original data

Here, a boxplot was used to show the data distribution. This plot displays much information about how the data is spread out in a small space (for more information, see Appendix F). Figure 4.2 and percentile values in Table 4-1 indicate that there is scatter at the end of the original data (longer periods), and the density of short duration times is much higher. At the same time, in the generated data, there are almost values for all numbers on the x-axis. This causes the results of the simulator to deviate slightly from the real data. The reason is that in the original data, the volume of events with short duration is much larger than that of events with long duration. Although this trend is normal for an exponential distribution, the difference between columns 1 and 2 or between columns 2 and 3 in the histogram is very large and significant (see Appendix E).

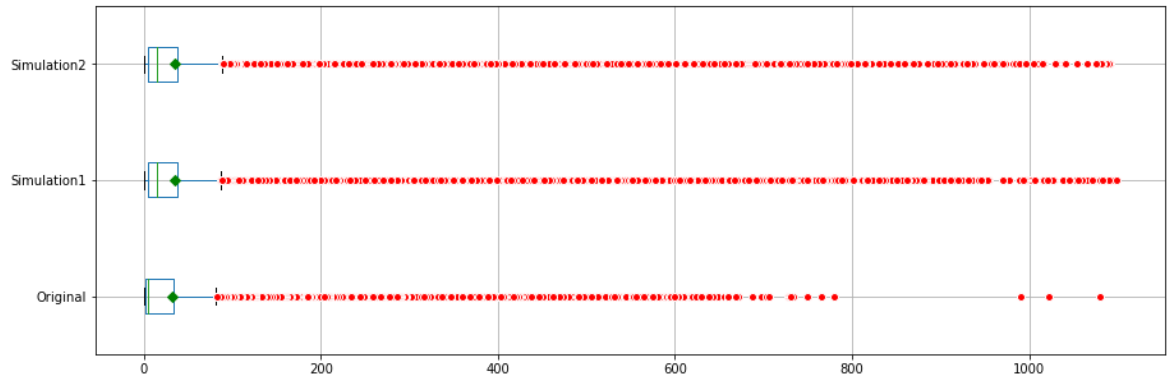


Figure 4.2. Boxplot - comparison of the number of events in reality and the simulations

Therefore, this difference can be interpreted as meaning that the data distribution tends toward shorter durations in the original data. In contrast, in the simulator-generated data, the trend of the data distribution appears to be more balanced.

The accuracy of the generated data was evaluated using three error metrics: MAE (mean absolute error), RMSE (root mean squared error), and R2 (R-squared). See definitions of error metrics in Appendix F. Using error metrics, we can check the average distance of the generated data from the real value. For example, the error value 7 of MAE shows that the simulator data has an average distance of 7 from the real data. According to the explanations in Table 4-1 and Figure 4.2, obtaining these values by error metrics was not far-fetched.

Metric	Original vs. Simulation1	Original vs. Simulation2
MAE	7.069	7.012
RMSE	14.425	14.098
NRMSE	0.451	0.440
R2	0.955	0.957

Table 4-2. Results of comparison of data models with error metrics (MAE, RSME and R- Squared).

Another way to compare data is to observe their rate of change over time. The following chart shows the mean duration time in 31 days. As shown in the figure, the results of Simulation1 are closer to the original data rate on different days. This is because the events in Simulation1 are generated based on average values for each day of the week and are, therefore, more accurate than in Simulation2. For example, in Simulation1, a longer time is specifically set for events on Sundays, 38.90 minutes on average. However, in Simulation2, an average time of 34.33 minutes is applied for Saturday and Sunday (weekend). The chart below shows that the peak values for Simulation1 on Sundays appear exactly like the original data.

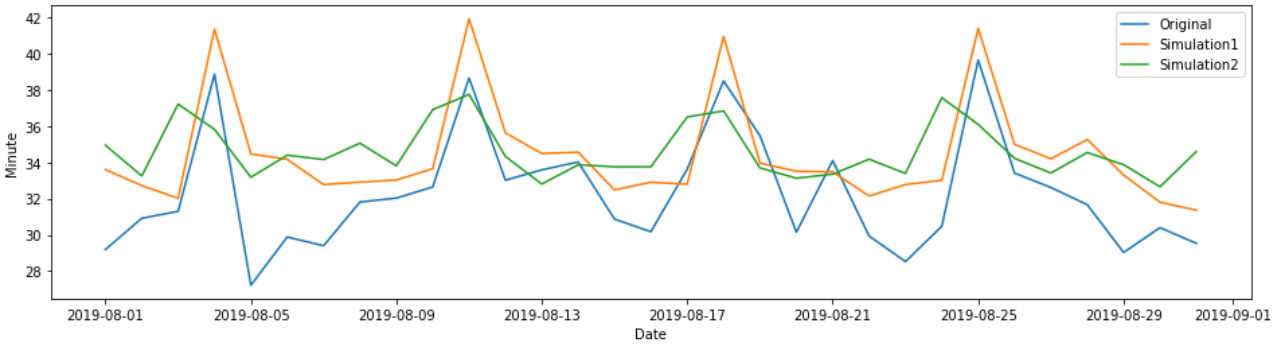


Figure 4.3. The mean duration time in 30 days. (Blue) Original data, (Orange) simulated data by Simulation1, (green) simulated data by Simulation2.

One of the things that can be checked here is the number of generated events. From a different perspective, they can be called requests because a request is sent to the server when each event is triggered. The following charts compare the number of requests with the original data for both Simulation1 and Simulaion2. Although the number of simulation requests is higher than that of the original data (e.g., 295371 vs. 220292 in Simulation1), their upward and downward trends are almost similar. This is due to the possibility of configuring the simulator for different days.

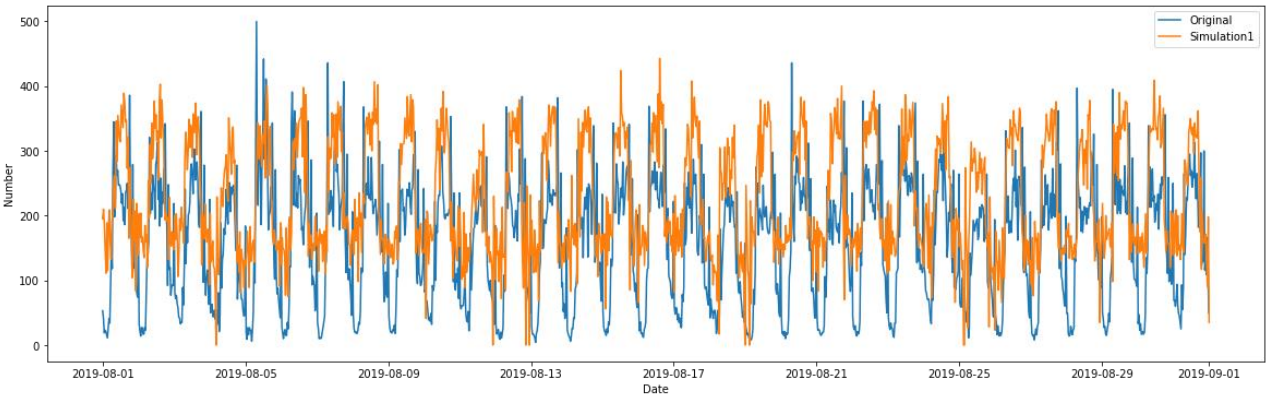


Figure 4.4. Number of requests during one month for Simulation1

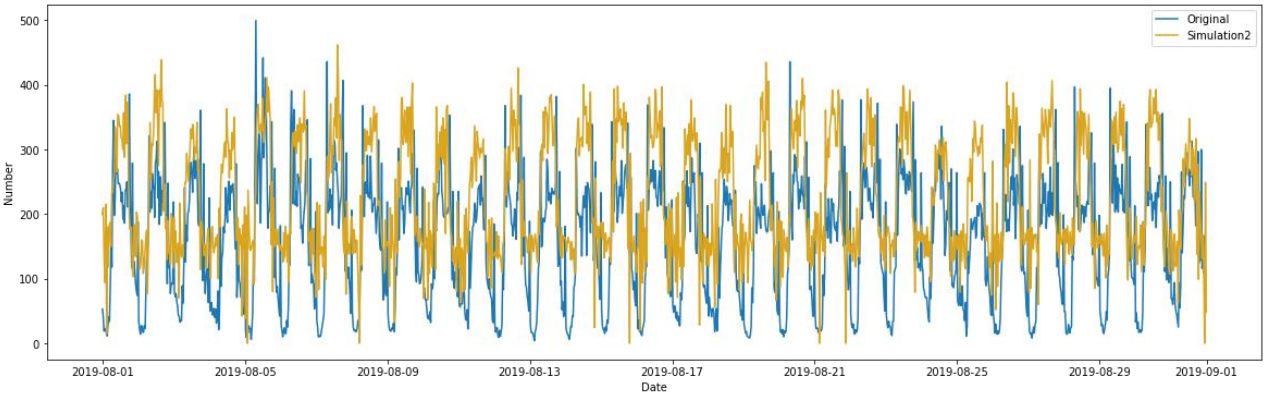


Figure 4.5. Number of requests during one month for Simulation2

4.2 Performance study

In addition to evaluating the results of the simulator, the case study is expanded to test and verify the simulator's performance. For this purpose, the simulator application was deployed in three different Amazon environments (using the AWS Free Tier [50]), each of which had its database. All three environments also had the same configurations, as shown in Table 4-3. In the next step, a parking model was created with the same characteristics for all three developed applications.

AWS Config	RDS	Elastic Beanstalk
Instance Type	t3.micro	t2.micro
vCPUs	2	1
Architecture	x86_64	i386, x86_64
Cores	1	1
Threads per core	2	1
Memory (GiB)	1	1

Table 4-3. AWS configuration for performance testing

The purpose of building these three independent environments by developing the same application was to test and compare the simulator's performance while running one, two, and three jobs simultaneously. For a better understanding of the result of this test, the information of these three environments and their names are listed in Table 4-4.

This experiment was performed three times, one hour at a time. Once with a time unit equal to 1 (Test1) and the second time with a time unit equal to 0.1 (Test2). As described earlier, the time unit is used to decrease or increase the simulation speed; in this case, the simulator generates data for about 10 hours in the real world in a period of one hour. The average time given for the random number in Test1 and Test2 is 10. In the third test (Test3), we used the same time unit as in the second test, but with an average time of 30 minutes for the random number generator. This means that the third test generates larger random numbers for the time interval between events.

Amazon CloudWatch metrics were used to compare the performance of three deployed environments in three independent tests. CloudWatch provides a variety of ways to monitor Amazon Web Services resources to analyze system and application performance and usage data. Three metrics were used here: CPU utilization for EC2, CPU utilization for RDS, and write throughput for RDS.

Environment/ Database	Distribution Function	Number of executed jobs	Duration of execution	Number of parking spots
(Test1) *TimeUnit = 1*				
Env1/ MySQL1	exponential (*λ=10*)	1 job	1 Hour	300
Env2/ MySQL2	exponential (*λ=10*)	2 jobs	1 Hour	300
Env3/ MySQL3	exponential (*λ=10*)	3 jobs	1 Hour	300
(Test2) *TimeUnit = 0.1*				
Env1/ MySQL1	exponential (*λ=10*)	1 job	1 Hour	300
Env2/ MySQL2	exponential (*λ=10*)	2 jobs	1 Hour	300
Env3/ MySQL3	exponential (*λ=10*)	3 jobs	1 Hour	300
(Test3) *TimeUnit = 0.1*				
Env1/ MySQL1	exponential (*λ=30*)	1 job	1 Hour	300
Env2/ MySQL2	exponential (*λ=30*)	2 jobs	1 Hour	300
Env3/ MySQL3	exponential (*λ=30*)	3 jobs	1 Hour	300

Table 4-4. Details of the test environments for performance tests

The results of this experiment are shown in the following charts regarding tests 1, 2 and 3. As seen in the figures, the percentage of processor utilization in EC2 is higher for environment 3 (with three concurrent jobs) than in environment 2 (with two simultaneous jobs) and higher in the second environment than in the first environment. Moreover, the average CPU utilization in Test2 is higher than in Test1. The reason for this is the short time intervals between events, as the time unit is smaller and the processor is more utilized.

But when comparing the third test with the second test, it is obvious that the load on the processor is slightly lower. Although the same time unit was used for both. The reason for this is the longer time interval between events (*λ=30*). In fact, the closeness of the time interval in Test2 compared to Test3 is the main reason for this difference.

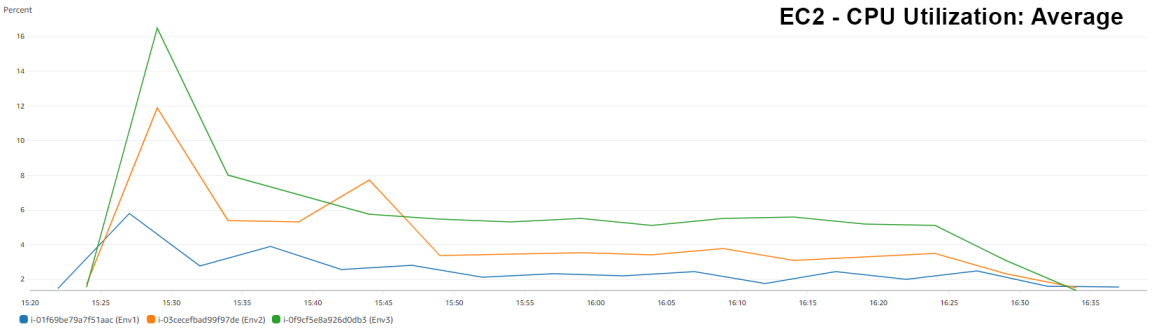


Figure 4.6. Test1 (TimeUnit=1, λ=10) - The percentage of allocated EC2 compute units

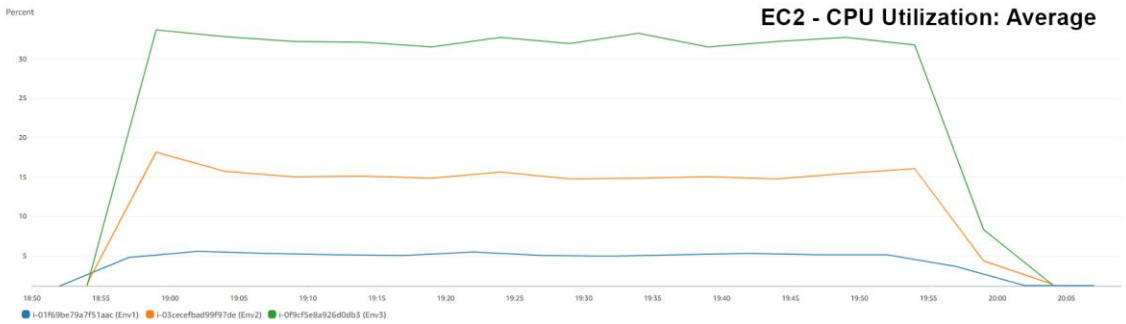


Figure 4.7. Test2 (TimeUnit=0.1, λ=10) - The percentage of allocated EC2 compute units

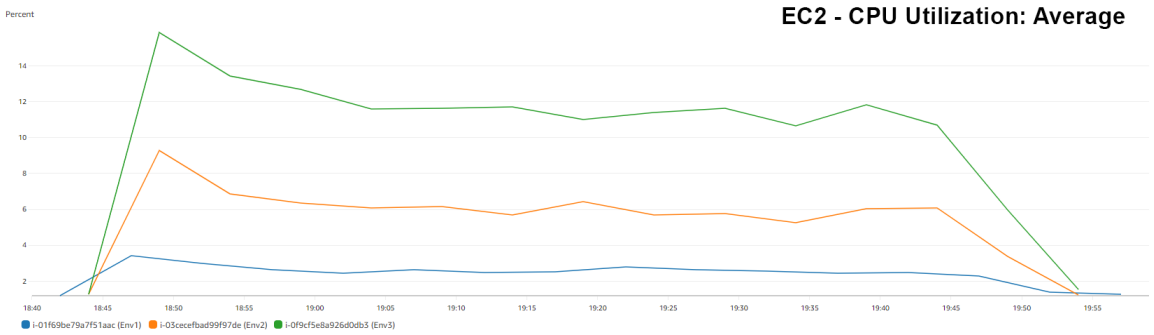


Figure 4.8. Test3 (TimeUnit=0.1, λ=30) - The percentage of allocated EC2 compute units

Comparing the processor utilization in RDS (database) between different environments in all tests in the following figures, it is again evident that the CPU utilization increases as the time unit are shortened, and the number of concurrent jobs is increased. As a result, the growth of CPU utilization (in RDS and EC2) is directly related to the selected time unit and the mean (time interval between events) for the random number generator.

When comparing the RDS processor consumption to the EC2 processor consumption, we observe that the EC2 processors are typically subjected to a greater workload. This is because of Amazon's dedicated hardware (Table 4-3). The RDS uses two CPUs with two threads, while the EC2 uses one CPU with one thread.

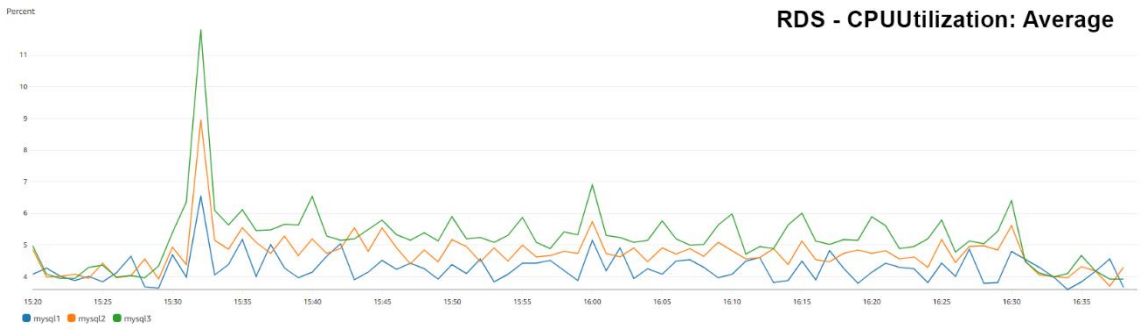


Figure 4.9. Test1 (TimeUnit=1, λ=10) - CPU utilization of RDS MySQL

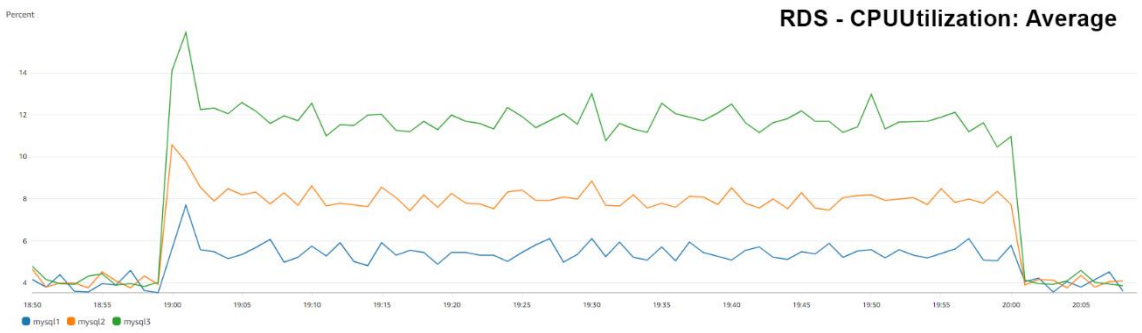


Figure 4.10. Test2 (TimeUnit=0.1, λ=10) - CPU utilization of RDS MySQL

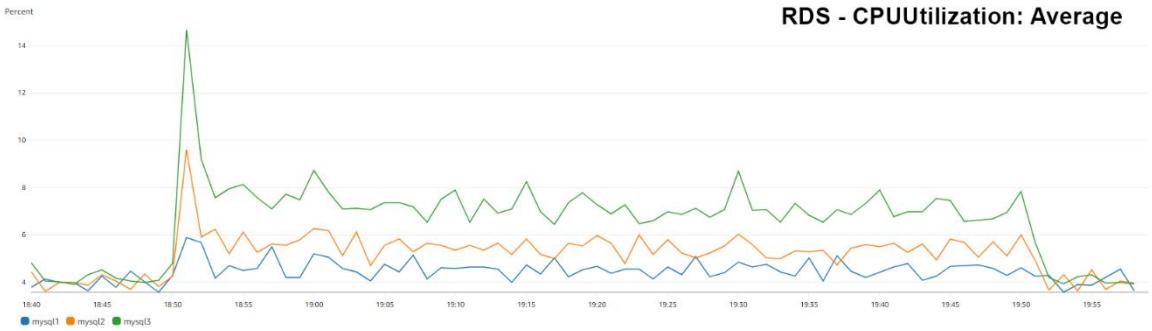


Figure 4.11. Test3 (TimeUnit=0.1, λ=30) - CPU utilization of RDS MySQL

This difference can also be seen in the RDS write throughput, which shows the number of bytes written to the database per second (the following figures). Again, the reason is the operational capacity of Amazon services (AWS Free Tier) for this test and the simultaneous execution of three/two simulation jobs. But the main reason, why the number of bytes written to the disk is extremely higher in the second/third test than in the first test is the high rate of events generated in the second/third test.

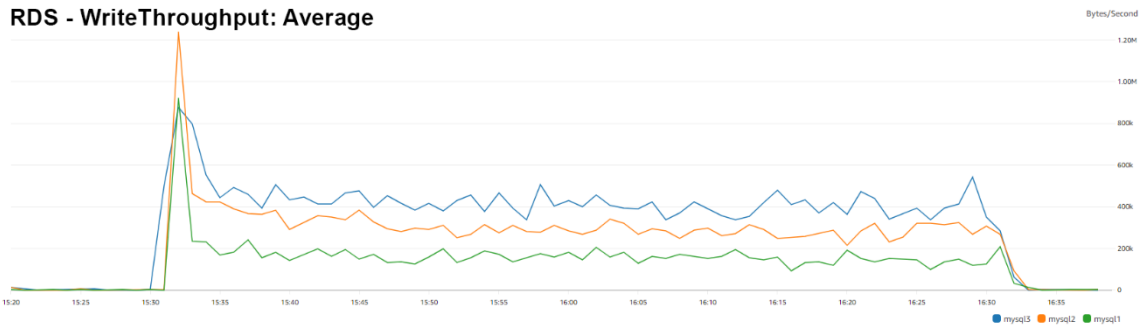


Figure 4.12. Test1 (TimeUnit=1, λ=10) - The number of bytes written to the database per second

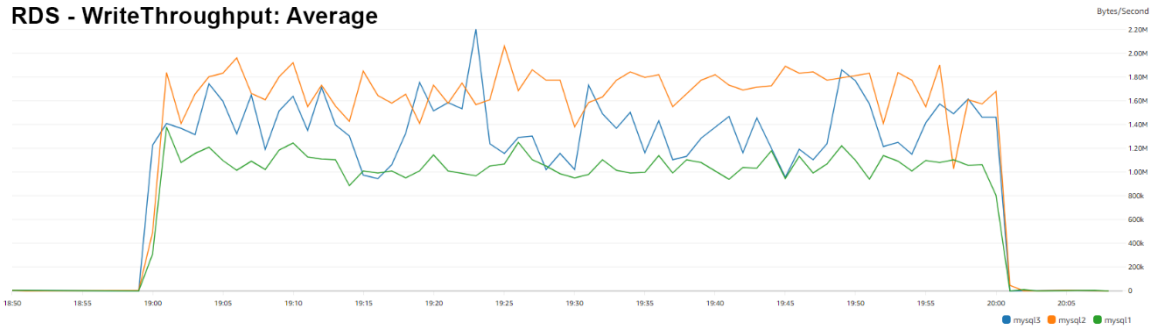


Figure 4.13. Test2 (TimeUnit=0.1, λ=10) - The number of bytes written to the database per second

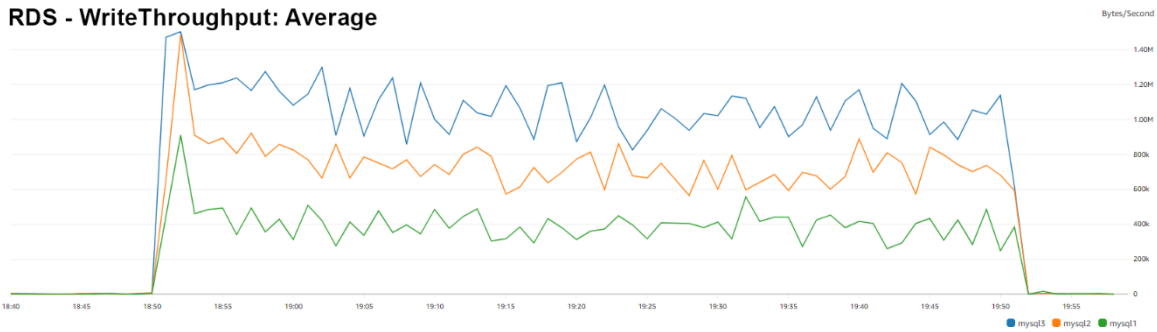


Figure 4.14. Test3 (TimeUnit=0.1, λ=30) - The number of bytes written to the database per second

Another noteworthy point in the above results is the significant increase in the percentage of CPU usage and the number of bytes written to disk (peaks) that occurs at the beginning of all job executions. The reason for this is the initialization phase, which occurs when a job starts and parking spots are created according to the parking lot's capacity and randomly scheduled for the next event. For this reason, the processor and the database are more heavily utilized than at other times.

Another result is the relationship of hardware performance to simulator settings. If we use the simulator to run one or more jobs simultaneously with normal execution speed (real time), we do not need powerful hardware. But of course, for settings like in the Test2, where both the random generator is set with a small mean and the simulation speed has been increased by the time unit, a powerful hardware is required.

A summary of the results obtained in this study has been presented in the following table to provide an overview of the results. This table contains six columns comparing the average values (mean value) for the whole execution time and the highest values (peak values) at the beginning of job execution.

	EC2 CPU Usage - Mean	EC2 CPU Usage - Peak	RDS CPU Usage - Mean	RDS CPU Usage - Peak	RDS Write Throughput - Mean	RDS Write Throughput - Peak
Test1-1	3.74%	5.80%	4.25%	6.54%	99.80 KB	923 KB
Test1-2	5.06%	11.88%	4.68%	8.97%	177.94 KB	1241 KB
Test1-3	6.30%	16.49%	5.04%	11.81%	232.89 KB	877 KB
Test2-1	5.08%	5.56%	5.47%	7.68%	1054 KB	1377 KB
Test2-2	14.30%	18.18%	8.05%	10.55%	1679 KB	2060 KB
Test2-3	30.26%	33.68%	11.86%	15.94%	1376 KB	2202 KB
Test3-1	1.68%	3.41%	4.12%	5.87%	75.57 KB	908 KB
Test3-2	2.85%	9.27%	4.37%	9.60%	137.91 KB	1485 KB
Test3-3	4.46%	15.84%	4.89%	14.66%	191.61 KB	1502 KB

Table 4-5. Performance test overview

4.3 Summary

This chapter presents the accuracy and performance metrics and some numerical results for the simulator. The results in this chapter indicate that some of the goals of this project were achieved. However, the most important objective pursued in this project was the implementation of a high-precision simulation that, in its first version, can simulate smart parking lots.

According to the results obtained in section 4.1 of this chapter, it can be concluded that this simulator can model parking lots based on the given parameters with an accuracy close to that of a real parking lot. Thus, this simulator can be used as a temporary replacement for sensors and IoT devices to test smart parking applications.

In the second test in section 4.2, the software was evaluated in terms of performance and quality. This experiment was conducted on the Amazon Web Service platform (free tier). This platform does not provide advanced hardware facilities but only the most basic facilities for testing Amazon services (Table 4-3). However, by comparing the test results, we concluded that the simulator still has acceptable performance even when running three jobs simultaneously.

5. Summary and Conclusions

This master thesis has taken a step toward developing IoT for smart cities. As explained extensively in chapter 1, the requirement for smart systems is increasing daily due to the population growth in today's cities, increasing energy consumption, and the limitation of various resources. It has made the topic of the Smart City one of the important research topics. Various tools, technologies and applications are currently being deployed for the development of smart cities. Since it is very useful and economical to test these systems in the real environment before the final deployment, a simulation environment can help develop Smart City technologies. This is because changes can be tested in a simulation environment to study the impact before applying these changes to real systems.

Based on this, this work focused on two main objectives: first, to develop an extensible simulator for all types of IoT devices, and second, to develop an algorithm for simulating smart parking lots. In the first step, research was conducted on smart cities and simulation systems used for Smart City projects. The results of this research are presented in chapter 2 (especially section 2.5). This research contributed to the design and development of the simulator. In addition, the study of the existing works and tools, on the one hand, provided a better understanding of the existing requirements in this field and, on the other hand, provided a good opportunity to identify the existing shortcomings and limitations. This work has attempted to address the limitations in previous works or provide opportunities beyond the previous ones.

Additionally, some datasets were studied to identify the features of data related to smart parking. The extraction of these features not only helped develop the simulator software but was also used to develop the desired algorithm and test the simulator. In section 3.2, the used dataset is described and analyzed.

Furthermore, in sections 3.3 and 0, a tool was designed and developed to simulate IoT devices and verify the smart parking model. With this software, the user can define one or more different models for the parking lot and simulate them with the appropriate configuration in the simulation part. The user can use the default developed algorithm or develop this open-source software for different purposes.

Finally, in the evaluation part, the tool was tested using two different case studies. In the first test, the accuracy of the generated data was measured and compared with the original data. In the second case study, the tool was tested for performance. The results of these two tests are presented in chapter 4.

In general, compared to the simulators presented in section 2.5, it can be claimed that they are either used for different purposes than our work or are located in a different layer of the system, such as iFogSim, which is used to model Fog and Edge nodes. While the simulation we developed is located in the sensing layer and is focused on simulating IoT devices. The most similar tools to this work are the IoTIFY and Bevywise, which cannot be fully studied and compared with the presented tool as they are commercial. However, it can be claimed that one of the distinctive features of the presented tool, compared to some previous works, is that there is a standard algorithm that supports users in their work.

Lastly, to verify some of the stated goals, we should look at chapter 3. For example, the images presented in section 3.4.5 illustrate that this tool is configurable. Therefore, the user can easily model a parking lot without programming knowledge by simply configuring the tool. This goal is achieved by implementing the appropriate algorithm (as a default algorithm).

5.1 Limitations

There were no major limitations in this work, but some cases slightly limited the scope of the work. For example, if there were no hardware and time constraints on execution, it would be possible to test the simulator's accuracy and performance over a long period (e.g., create a sample for a year) and send data to the server. The second issue was the limitation of available datasets in the field of the Internet of Things and smart parking in particular. Due to the limitations of these projects or the non-public of their data on the Internet, it was not possible to review different types of data.

5.2 Future Work

The new simulator allows modeling parking lots and IoT devices in smart parking lots. However, developing it for other purposes and adding additional features is possible. As part of this project's continuation, we discuss future works to improve and extend its performance.

- **Custom algorithm**: One of the features that multiply the functionality of this software is the possibility to add custom algorithms in a high-level language like Python by experts and developers. This way, the user can write his algorithm instead of using the default one. Furthermore, the required parameters for the algorithm can also be provided in the form of hard code or by developing the user interface.
- **Determine periods by user**: Currently, there are two time periods for the division of daily hours (night and day), where the user has to enter the starting hours of darkness and daylight. Perhaps this option will be much more useful if the user can make this division according to his needs.
- **Limitation of the request number**: As explained in chapter 4, the number of requests from the simulator is much higher than the number of real data (the desired request number), probably affecting the overall average. If it is possible to limit this somehow, better results may be obtained. This possibility can be optionally created in the user interface.
- **Holidays**: One of the software's features, shown in section 3.4.5 and tested in section 4.1, is the possibility to enter average time information in different forms (Figure 3.10 and Figure 3.11). In the WorkRestTime variant, this capability is currently limited only to workdays and weekends. The reason for the inactive holiday field is the lack of information about a country's official holidays. The possibility of adding this information and using it in the simulation can allow the user to simulate the request rate on holidays in addition to working days and weekends.

- **Temperature and humidity conditions**: Another piece of information that leads to the production of more accurate and unique data is the temperature and humidity conditions in the parking area. Naturally, different weather conditions affect traffic volumes and parking times.
- **Development of templates**: The idea to develop this simulator for different urban services existed from the beginning; on this basis, the corresponding application was designed and implemented. Therefore, this tool can be developed for other urban systems such as street lighting, traffic, transportation, energy, etc.
- **Error generator**: One of the features that can make this tool more efficient for testing smart applications is the ability to generate errors. An error generator brings the data transfer process closer to real environments. The term “error generator” is a general topic with different aspects. Everything from simulating a defective or inactive device to generating a wrong message to simulating a DoS attack can be part of this topic.
- **Custom message**: currently, the message structure sent to the server is fixed and depends on the fields of the record table. However, the possibility of customizing the sent messages can also be a practical option. This means that the user can specify the parameters and their types.
- **Server-Sent Events**: SSE is an asynchronous and un-directional communication technology that allows the client to automatically receive information from the server. This technology enables IoT devices to receive commands from the server for some decisions, for example, reserving a parking spot or reducing or increasing the amount of light from a streetlight depending on the humidity. Moreover, these are the decisions made by processing Big Data on the server. By implementing SSE in the simulator, we give Smart City applications that support push technology the ability to test it.

Bibliography

[1] Department of Economic and Social Affairs, Population Division, "World Urbanization Prospects 2018: Highlights (ST/ESA/SER.A/421)," United Nations, 2019.

[2] H. Ritchie and M. Roser, "Urbanization," 9 2018. [Online]. Available: https://ourworldindata.org/urbanization.

[3] G. Gagliardi, M. Lupia, G. Cario, F. Tedesco, C. G. Francesco, F. Lo Scudo and A. Casavola, "Advanced Adaptive Street Lighting Systems for Smart Cities," *Smart Cities,* vol. 3, no. 4, pp. 1495-1512, 7 12 2020.

[4] F. M. Awan, Y. Saleem, R. Minerva and N. Crespi, "A Comparative Analysis of Machine/Deep Learning Models for Parking Space Availability Prediction," *Sensors,* vol. 20, no. 1, p. 322, 2020.

[5] R. Faria, L. Brito, K. Baras and J. Silva, "Smart mobility: A survey," in *International Conference on Internet of Things for the Global Community (IoTGC)*, 2017.

[6] C. Thangavel and P. Sudhaman, "Security Challenges in the IoT Paradigm for Enterprise Information Systems," *Connected Environments for the Internet of Things,* pp. 3-17, 2017.

[7] "Smart City Simulation," IoTify - Developer Docs, [Online]. Available: https://iotify.help/network/smart-city/simulation.html.

[8] A. R. Portabales and M. L. Nores, "Dockemu: An IoT Simulation Framework Based on Linux Containers and the ns-3 Network Simulator — Application to CoAP IoT Scenarios," *Advances in Intelligent Systems and Computing,* pp. 54-82, 2019.

[9] T. Pflanzner, M. Fidrich and A. Kertesz, "Simulating Sensor Devices for Experimenting with IoT Cloud Systems," *Connected Environments for the Internet of Things,* pp. 105-126, 2017.

[10] H. Liu, "Key Issues of Smart City," in *Smart Cities: Big Data Prediction Methods and Applications*, Singapore, Springer Singapore, 2020, pp. 3-23.

[11] L. Neckermann, Smart Cities, Smart Mobility: Transforming the Way We Live and Work, Troubador Publishing Ltd, 2017.

[12] A. E. Oke, S. S. Stephen, C. O. Aigbavboa, D. R. Ogunsemi and I. O. Aje, Smart Cities: A Panacea for Sustainable Development, Emerald Group Publishing, 2022.

[13] L. G. Anthopoulos, Understanding Smart Cities: A Tool for Smart Government or an Industrial Trick?, Springer International Publishing, 2017.

[14] L. Anthopoulos, M. Janssen and V. Weerakkody, "A Unified Smart City Model (USCM) for Smart City Conceptualization and Benchmarking," *International Journal of Electronic Government Research,* vol. 12, no. 2, pp. 77-93, 2016.

[15] M. Elhoseny and A. E. Hassanien, Emerging Technologies for Connected Internet of Vehicles and Intelligent Transportation System Networks, Cham: Springer International Publishing, 2020.

[16] A. Shufian, M. A. Afroj, M. Hasibuzzaman, M. Al Miraz, A. Sarker, M. J. I. Rashed and M. A. Miah, "Smart Car Parking Technique for Metropolitan City," *Lecture Notes in Electrical Engineering,* vol. 686, pp. 143-154, 2020.

[17] K. Ashton, "That “Internet of Things” Thing," *RFiD Journal,* vol. 22, pp. 97-114, 2009.

[18] G. Surtani, S. Gupta, S. Singh, M. Yadav and M. S.S.Kulkarni, "Smart Parking System and It‟s Simulation," *International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE),* vol. 7, no. 3, pp. 327-332, 2018.

[19] A. Khanna and R. Anand, "IoT based smart parking system," *International Conference on Internet of Things and Applications (IOTA),* 2016.

[20] "What is Big Data?," Oracle, 2021. [Online]. Available: https://www.oracle.com/big-data/what-is-big-data/. [Accessed 7 2022].

[21] "Big Data: was es ist und was man darüber wissen sollte," SAS, [Online]. Available: https://www.sas.com/de_at/insights/big-data/what-is-big-data.html. [Accessed 7 2022].

[22] I. A. T. Hashem, V. Chang, N. B. Anuar, K. Adewole, I. Yaqoob, A. Gani, E. Ahmed and H. Chiroma, "The role of big data in smart city," *International Journal of Information Management,* vol. 36, no. 5, pp. 748-758, 2016.

[23] G. D. Maayan, "How Big Data Impacts Smart Cities," DATAVERSITY, 28 4 202. [Online]. Available: https://www.dataversity.net/how-big-data-impacts-smart-cities/. [Accessed 7 2022].

[24] C. Baun, M. Kunze, J. Nimis and S. Tai, Cloud Computing: Web-basierte dynamische IT-Services, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[25] Z. Mahmood, Fog Computing: Concepts, Frameworks and Technologies, Cham: Springer International Publishing, 2018.

[26] F. Al-Turjman, Edge Computing: From Hype to Reality, Cham: Springer International Publishing, 2019.

[27] S. Tanwar, Fog Data Analytics for IoT Applications: Next Generation Process Model with State of the Art Technologies, vol. 76, Singapore: Springer, 2020, pp. 175-198.

[28] N. D. Patel, B. M. Mehtre and R. Wankar, "Simulators, Emulators, and Test-beds for Internet of Things: A Comparison," *Third International conference on I-SMAC,* 2019.

[29] L. ADLER, "SimCities: Designing Smart Cities through Data-Driven Simulation," Data-Smart City Solutions, 29 8 2016. [Online]. Available: https://datasmart.ash.harvard.edu/news/article/simcities-designing-smart-cities-through-data-driven-simulation-893.

[30] H.-J. Bungartz, S. Zimmer, M. Buchholz and D. Pflüger, Modellbildung und Simulation: Eine anwendungsorientierte Einführung, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[31] B. R. HAVERKORT, Performance of Computer Communication Systems: A Model-Based Approach, Aachen, Germany: Wiley, 2001.

[32] C. Pilch, *Development of an event-based simulator for model checking hybrid Petri nets with random variables, M.S. Thesis,* Münster: Faculty of Mathematics and Computer Science, 2016.

[33] T. Manivannan and D. P. Radhakrishnan, "A Comprehensive Analysis of Simulation Tools for Internet of Things," *Solid State Technology,* vol. 63, no. 5, pp. 461-471, 2020.

[34] B. F. d. S. A. Belchior, *Testing in IoT systems: From simulation to visual-based testing, M.S. Thesis,* Porto, Portugal: Faculty of Engineering of the University of Porto, 2019.

[35] "IoTIFY Network Simulator," IoTIFY - Docs, [Online]. Available: https://docs.iotify.io/.

[36] "Bevywise IoT Simulator," Bevywise - Docs, [Online]. Available: https://www.bevywise.com/iot-simulator/help-document.html.

[37] "OMNeT++ simulation environment," Cogitative Software FZE, [Online]. Available: https://omnetpp.org/intro/.

[38] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments," *Software: Practice and Experience,* vol. 47, no. 9, pp. 1275-1296, 2017.

[39] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *Journal of Systems Architecture,* vol. 72, pp. 93-107, 2017.

[40] S. P. Lau, G. V. Merrett, A. S. Weddell and N. M. White, "StreetlightSim: A simulation environment to evaluate networked and adaptive street lighting," *IEEE Asia Pacific Conference on Wireless and Mobile,* 2014.

[41] E. Dizon and B. Pranggono, "Smart streetlights in Smart City: a case study of Sheffield," *Journal of Ambient Intelligence and Humanized Computing,* vol. 13, no. 63, p. 2045–2060, 2021.

[42] M. Pahr, *Analysis and Processing of Smart City Data M.S. Thesis,* Vienna: University of Applied Sciences FH Technikum Wien, 2020.

[43] "MoSCoW Prioritisation," Agile Business Consortium, [Online]. Available: https://www.agilebusiness.org/page/ProjectFramework_10_ MoSCoWPrioritisation. [Accessed 06 2022].

[44] "Vaadin Licensing," Vaadin Ltd, [Online]. Available: https://vaadin.com/license.

[45] "On-street Car Parking Sensor Data - 2019," City of Melbourne, 14 4 2020. [Online]. Available: https://data.melbourne.vic.gov.au/Transport/On-street-Car-Parking-Sensor-Data-2019/7pgd-bdf2.

[46] Mainziel-Straßenverkehrsamt, "Parkdaten dynamisch," Offene Daten Portal der Stadt Frankfurt/Main, 14 01 2022. [Online]. Available: https://www.offenedaten.frankfurt.de/dataset/parkdaten-dynamisch. [Accessed 7 2022].

[47] T. Erdogan, "distfit's documentation," 2020. [Online]. Available: https://erdogant.github.io/distfit/pages/html/index.html. [Accessed 2022].

[48] D. P. Voorhees, "Introduction to Software Design," in *Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models*, Cham, Springer International Publishing, 2020, pp. 1-15.

[49] "Commons Math: 8 Probability Distributions," Apache Commons TM, 21 5 2021. [Online]. Available: https://commons.apache.org/proper/commons-math/userguide/distribution.html. [Accessed 7 2022].

[50] "AWS Free Tier FAQs," Amazon Web Service, [Online]. Available: https://aws.amazon.com/free/free-tier-faqs/. [Accessed 7 2022].

[51] M. Paul, "Boxplots leicht gemacht!," Statistik Gym, 8 10 2019. [Online]. Available: https://www.statistikpsychologie.de/boxplot/. [Accessed 7 2022].

[52] "Error Metrics: How to Evaluate Your Forecasting Models," Jedox, [Online]. Available: https://www.jedox.com/en/blog/error-metrics-how-to-evaluate-forecasts/. [Accessed 7 2022].

List of Figures

List of Tables

Appendices

A. Functional Requirements

MUST

1. The simulator must be able to perform a discrete event simulation for a specified period on the input model. In this case, the input model refers to parking on the ground of Queen Street in Melbourne [45].
2. The simulator must be configurable to enable the user to appropriately adjust the parameters being considered to simulate a parking lot. Some param
    a. Simulation of a parking lot by determining a general average parking time.
    b. Simulation of a parking lot by determining the average parking time for each day of the week.
    c. Simulation of a parking lot by determining the average parking time for weekends and weekdays.
    d. Possibility to set the three above parameters separately for day (daylight) and night (darkness)
    e. Possibility to set the start time of daylight/sunrise and darkness/sunset
    f. Possibility to set the capacity of a parking lot
    g. Possibility to set the default number of occupied spots for a parking lot
3. The simulator must be able to run several simulation jobs at the same time. This means that the user can simulate some parking lots by running different jobs simultaneously.
4. The time unit of the simulator must be adjustable. For example, if the user sets the simulation time unit to 0.1, then every minute in the simulation environment corresponds to 10 minutes in the real system. This means that the simulator will actually run faster and can simulate a period of 30 days in three days.
5. The parameters of the model and the configurations of the simulator must be stored in an internal database like MySQL.
6. The simulator must be able to store the generated/simulated data in its own database.
7. In order to test the algorithm of the simulator, a dataset needs to be generated for a specific parking lot (such as Queen Street in Melbourne) and in a specific period and compared to the original dataset from Melbourne [45].
8. The simulator must be able to generate random numbers using at least one or more of the following distributions:
    a. Exponential distribution
    b. Normal distribution
    c. Gamma distribution
    d. Weibull distribution

SHOULD

1. The simulator should display some reports on the simulation results (e.g., start and end time of simulation, request numbers per day/week/month, and parking duration time per day/week/month)
2. The simulated data should be exported in CSV format for evaluation purposes.
3. User should be able to export and import configurations in a format like JSON, Properties or YAML.
4. The simulator should have error handling and handle any type of exception that interrupts execution.

COULD

1. The simulator is desired to store the generated data in an external database (e.g., SQLite or MongoDB)
2. The simulator is desired to transfer the simulated data to a configured server (e.g., a cloud application)
3. The simulator is desired to send the data to a server via a configured protocol such as CoAP, MQTT or HTTP
4. The simulator is desired to transmit the data to a server in a configured format such as JSON, CSV or XML
5. The simulator should have a suitable logging system that stores the main steps of the simulation process and the results in a log file or in the database (in different log levels).

WON'T

1. The simulator will not yet provide Javadoc documentation in the form of HTML files, but it is desirable to implement this in the future.
2. Currently, there are no tooltips and user guides in the GUI. It is better to implement this in the future to facilitate the use of the simulator.
3. The simulator does not yet run a simulation job for two or more parking lots at the same time (or even, for example, a parking lot template and a streetlight template).
4. The simulator will not yet allow to add a user-defined algorithm in the Python language instead of the default algorithm
5. It would be nice to implement the frontend with another framework like ReactJS instead of Vaadin, or to combine these two, since some components of Vaadin like charts are not free to use.
6. And all the "nice to have" features explained in section 5.2, such as information about holidays, temperature and weather, error generator, SSE and custom messages.

B. Random Variable Generators

The following table shows the implementation of discrete and continuous distributions in the Apache library. All distributions include a sample() method used to generate random numbers. For more details, see Source [49].

Distribution	**Generator**	**Parameters**
Beta	BetaDistribution	$\alpha, \beta \in \mathbb{R}$ – (Alpha and Beta)
Binomial	BinomialDistribution	trials $\in \mathbb{N}$, p $\in \mathbb{R}$ – (Probability of success)
Cauchy	CauchyDistribution	$median, scale \in \mathbb{R}$
Chi Squared	ChiSquaredDistribution	$n \in \mathbb{R}$ – (Degrees of freedom)
Exponential	ExponentialDistribution	$\lambda \in \mathbb{R}$ – (mean)
F	FDistribution	$n, d \in \mathbb{R}$ – (Numerator and Denominator)
Gamma	GammaDistribution	$shape, scale \in \mathbb{R}$
Geometric	GeometricDistribution	$p \in \mathbb{R}$ – (Probability of success)
Hypergeometric	HypergeometricDistribution	$p, n, s \in \mathbb{R}$ – (Population, number, sample size)
Levy	LevyDistribution	$mu, c \in \mathbb{R}$ – (Location, scale)
Log-Normal	LogNormalDistribution	$scale, shape \in \mathbb{R}$
Normal	NormalDistribution	$\lambda, \sigma \in \mathbb{R}$ – (Mean, standard deviation)
Pareto	ParetoDistribution	$scale, shape \in \mathbb{R}$
Pascal	PascalDistribution	$r, p \in \mathbb{R}$ – (Number and Probability of success)
Poisson	PoissonDistribution	$p \in \mathbb{R}$ – (Poisson mean)
T	TDistribution	$d \in \mathbb{R}$ – (Degrees of freedom)
Triangular	TriangularDistribution	$a, b, c \in \mathbb{R}$ – (Lower limit, Upper limit, mode)
Uniform Integer	UniformIntegerDistribution	$lower, upper \in \mathbb{R}$
Uniform Real	UniformRealDistribution	$lower, upper \in \mathbb{R}$
Weibull	WeibullDistribution	$\alpha, \beta \in \mathbb{R}$ – (Alpha and Beta)
Zipf	ZipfDistribution	$n, d \in \mathbb{R}$ – (Number of elements and Exponent)

C. Goodness of fit with Distfit and K-S test

Here again, the *distfit* function and K-S test were used to determine the goodness of fit of the empirical data. The results show that the exponential distribution ranks third, and the Dweibull and Gamma distributions are first and second, respectively. However, further investigation found that both the Gamma and Dweibull distributions are similar to the exponential distribution in some special cases. Therefore, we also attempted to generate random numbers in python for the first three distributions (Exponential, Gamma, and Dweibull) using the parameters listed in the table below and repeated the experiment with the random numbers. Three Python methods are used to generate random numbers to determine if they really fit into one of these distributions:

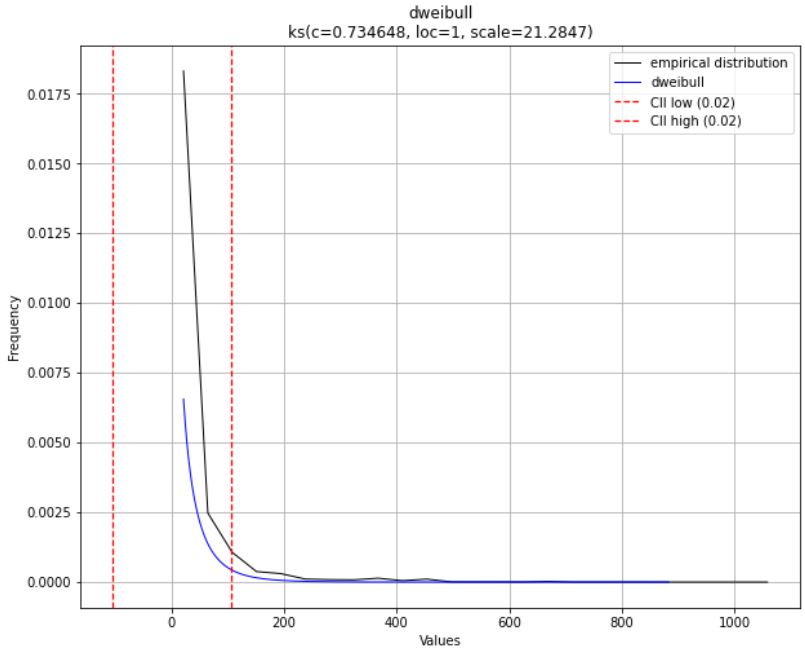
$$np.random.exponential(scale, size)$$

$$np.random.gamma(shape, scale, size)$$

$$np.random.weibull(a, size) \text{ or}$$
$$scipy.stats.weibull\_min.rvs(a, loc, scale, size)$$

The result was different for the first two cases, and only the mean of the random exponential data was very close to the experimental data. Therefore, based on this experiment, the exponential distribution model was selected for our empirical data.

	distr	score	LLE	loc	scale	arg
0	dweibull	1.108472	NaN	1.0	21.28465	(0.7346479499791447,)
1	gamma	1.108472	NaN	-0.0	175.210115	(0.34240852592845084,)
2	expon	2.716536	NaN	0.0	31.985456	()
3	pareto	2.716536	NaN	-0.808938	0.808938	(0.4343496282554167,)
4	t	2.716536	NaN	1.012107	1.287194	(0.4009850082790162,)
5	loggamma	2.716536	NaN	-10320.819252	1681.200159	(472.99410793361085,)
6	norm	3.228419	NaN	31.985456	68.426721	()
7	genextreme	3.228419	NaN	2.090289	14.247144	(-6.815865139618332,)
8	lognorm	3.228419	NaN	-0.0	2.323414	(8.532457678850543,)
9	beta	5.070877	NaN	-0.0	6070.292102	(0.34398564031160805, 300.00706088173865)
10	uniform	9.508497	NaN	0.0	1080.0	()


dweibull
ks(c=0.734648, loc=1, scale=21.2847)
empirical distribution
dweibull
CII low (0.02)
CII high (0.02)
Frequency
Values
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
200
400
600
800
1000

D. The Distfit results for simulated data

Both figures refer to the execution of the *distfit* method on simulated data in section 4.1.

Simulation1:

	distr	score	LLE	loc	scale	arg
0	expon	0.000008	NaN	0.1	34.151084	()
1	t	0.000027	NaN	10.021011	9.985629	(1.0055278239335337,)
2	beta	0.000037	NaN	0.1	5722.153968	(0.6101874180576812, 101.0687458566969)
3	lognorm	0.000044	NaN	0.099358	12.566549	(1.5114513425671794,)
4	genextreme	0.000056	NaN	7.258562	9.467323	(-1.10521677067167,)
5	dweibull	0.000116	NaN	5.466667	20.705467	(0.6542260804042288,)
6	pareto	0.00013	NaN	-2.454376	2.554376	(0.5234529781884905,)
7	norm	0.000146	NaN	34.251084	61.682658	()
8	loggamma	0.000151	NaN	-21607.957803	2849.247114	(1990.7296740612023,)
9	uniform	0.000307	NaN	0.1	1098.45	()
10	gamma	0.000327	NaN	0.1	3.894015	(0.026860647208902595,)

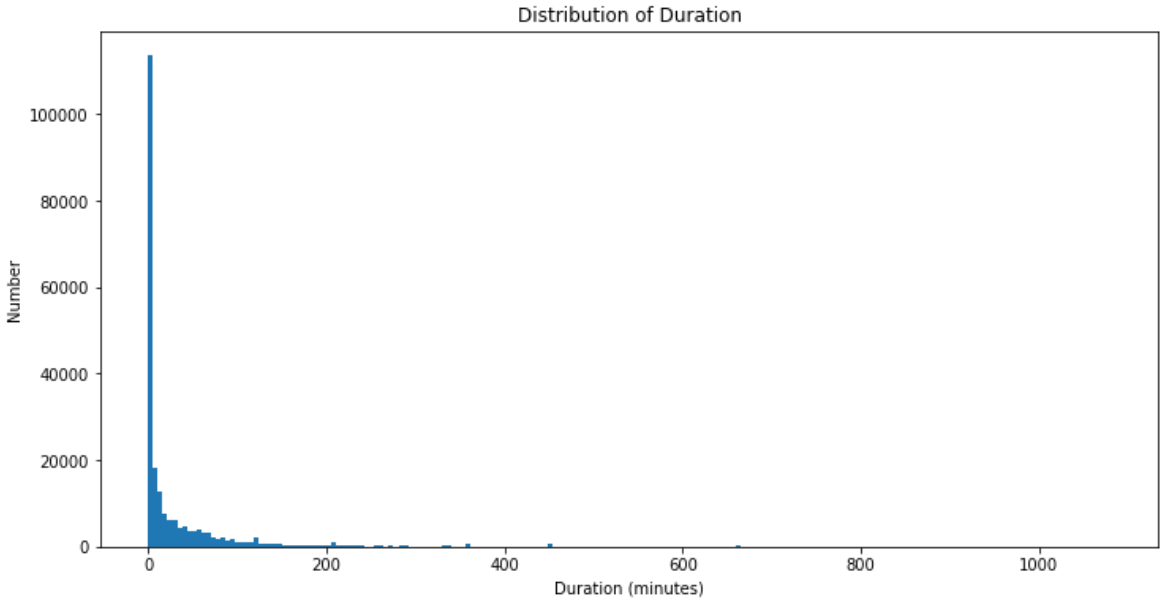
Simulation2:

	distr	score	LLE	loc	scale	arg
0	expon	0.000009	NaN	0.116667	34.437338	()
1	t	0.000024	NaN	10.158386	10.186233	(1.01085388001759,)
2	lognorm	0.000044	NaN	0.115278	12.628146	(1.5208417456889967,)
3	beta	0.000055	NaN	-127.023866	634191865002527.375	(19.640417844346903, 78255999784494.42)
4	genextreme	0.000055	NaN	7.311156	9.56581	(-1.1095294433437823,)
5	dweibull	0.00012	NaN	5.1	21.289979	(0.6480235917021253,)
6	pareto	0.000136	NaN	-2.018293	2.134959	(0.46184210117504276,)
7	norm	0.000146	NaN	34.554005	61.631254	()
8	loggamma	0.000156	NaN	-21259.185074	2838.296851	(1812.2227909936064,)
9	uniform	0.000307	NaN	0.116667	1091.983333	()
10	gamma	0.000328	NaN	0.116667	3.869994	(0.0297075468894838,)

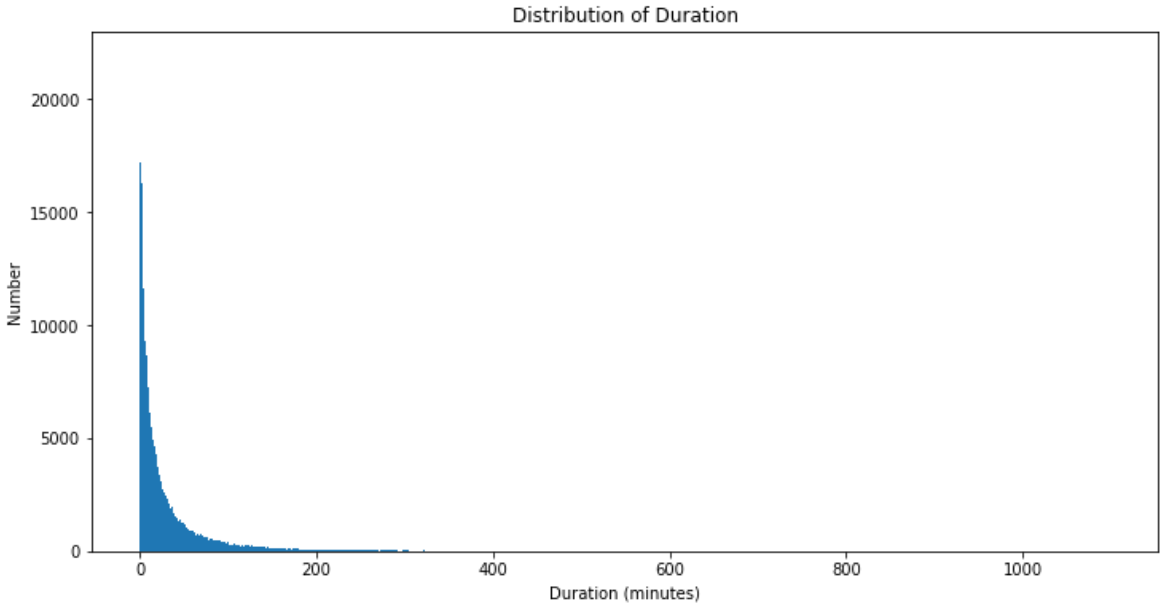
E. Histogram for the duration

Here are histograms plotted for three data sets (original and result of simulation 1 and simulation 2). The x-axis shows the time interval between events (duration time), and the y-axis displays the number of events with such duration time. For example, as shown in the diagram below, the number of events with a duration time of 1 minute is more than 100,000, while this value for a duration time of 2 minutes is about 20,000. This difference between the individual columns in the simulator data is much more reasonable than the real data.

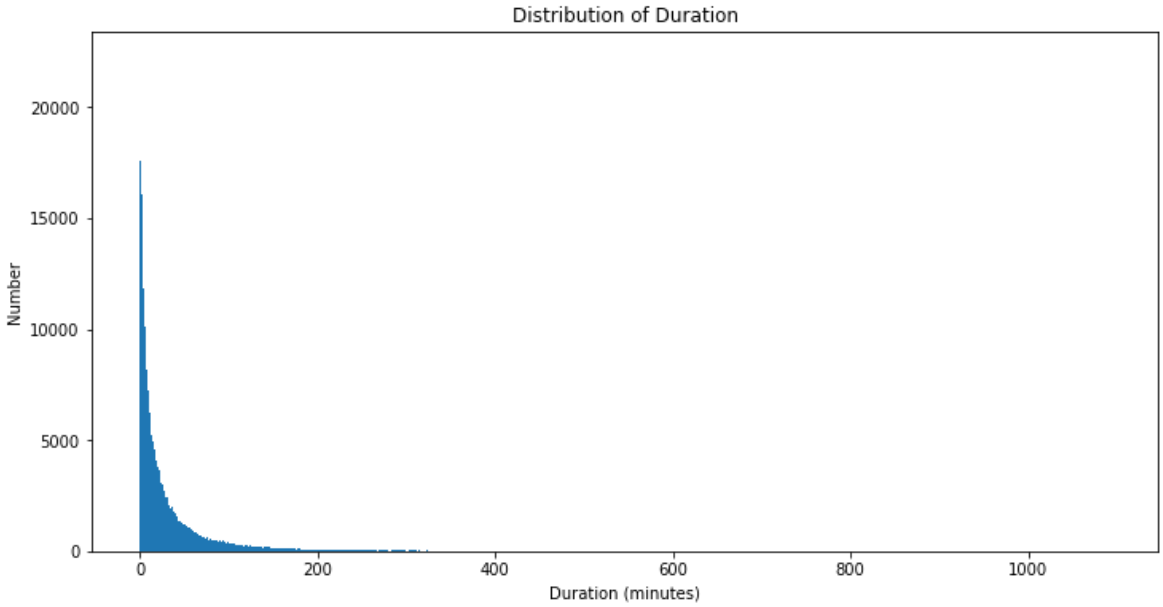
Original Data:



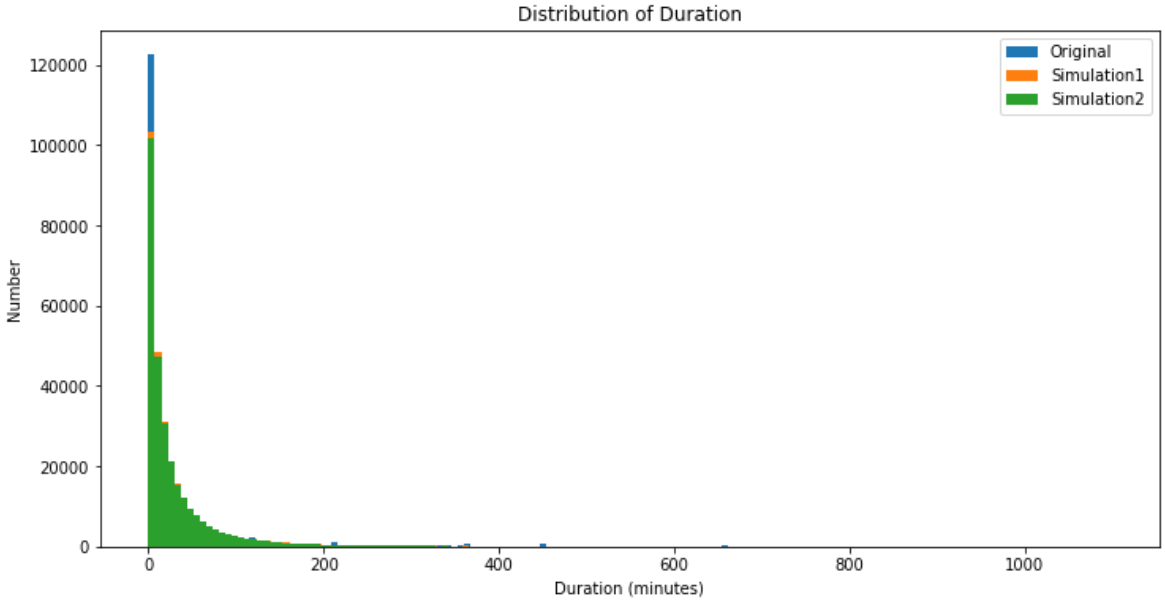
Simulation 1:



Simulation 2:



Display of the comparison of all three in one diagram:



F. Statistical tools and Python packages

Boxplot:

One of the most famous diagrams that shows many indicators of descriptive statistics in the context of data is the “boxplot”. Boxplot is a standard method of showing the distribution of data using the statistical indicators of smallest amount (minimum), first quartile (Q1), median, third quartile (Q3), and largest amount (maximum). This chart can also give you information about the presence of outliers and determine their value. Also, showing the symmetry of the data is one of the functions of this chart. It is worth mentioning that the concentration level and even the skewness of the data can also be seen in this diagram [51].

Distfit:

The *distfit* function is a Python package for fitting a probability distribution to a data set. This function performs a goodness of fit test among 89 probability distributions using the Residual Sum of Squares (RSS), and after comparing it to each of the 89 different distributions, scores them and returns the best one. The *distfit* function has multiple use cases. Besides determining the best theoretical distribution for a data set, it is also used to detect outlier points, null distributions, and new data points that deviate significantly [47].

A sample Python code for using this function:

```
from distfit import distfit
dist = distfit()
dist.fit_transform(empirical_data)
dist.plot()             # Draw the most suitable distribution
dist.summary            # Report summary of the found distributions
```

Residual Sum of Squares (RSS):

In statistics, the RSS is the sum of the squares of the observed values and the values predicted by a model. Such a variance is called a residual or simply an error. The RSS measures the variance of the error term. Therefore, it is a quality measure for a linear model and describes the imprecision of the model. The RSS is also known by other names such as SSR (Sum of Squared Residuals) and SSE (Sum of Squared Errors). RSS is computed by:

$$RSS = \sum_{i=1}^{i=n} (y_i - f(x_i))^2$$

Mean Absolute Error (MAE):

The MAE is the average difference between the real value and the predicted value over all training samples, where y is the actual value and y´ is the predicted value. n is the total number of values in the test set. The MAE indicates the average error we can expect in the prediction. The error values are given in the original units of the predicted values, and MAE

0 means that there is no error in the predicted values. In fact, the lower the MAE value is, the model or estimation is better [52].

$$MAE = \frac{1}{n}\sum_{i=1}^{i=n}|y' - y|$$

Root Mean Squared Error (RMSE):

The RMSE is the square root of the MSE. MSE, or mean squared error in estimation theory, indicates how much the point estimator scatters around the value being estimated. The MSE is always positive, and if the result goes to zero, it means that there are fewer errors. RMSE is the abbreviation for the root mean squared error. It is used to measure the root mean square difference between the predicted and the actual observation. In the following formula, y′ is the predicted value, y is the actual value, and n is the total number of values in the test set. By comparing the RMSE and MAE, it is possible to determine whether the prediction has large but rare errors [52].

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{i=n}(y' - y)^2}$$

Normalized Root Mean Squared Error (NRMSE):

The NRMSE, like the RMSE, is used to evaluate the error of a model, except that it standardizes this work in terms of the range of values used in the model. It is also useful when scatter is important and extreme values need to be normalized. To calculate the NRMSE, the RMSE value must be divided by the mean or range (the difference between the maximum and minimum values) [52].

$$NRMSE = \frac{RMSE}{mean(y)} \quad \boldsymbol{or} \quad NRMSE = \frac{RMSE}{y_{max} - y_{min}}$$

R-Squared (R2):

R-Squared is the percent of variation in the response described by a linear model. It is a statistical metric used to measure the fit of the data to the regression line. It is also called the coefficient of determination or, in the case of multiple regression, the coefficient of multiple determination. The R-Squared always takes values between 0 and 100 percent. 0% means that the model does not explain the deviation of the response from the mean at all. 100% means that the model fully explains the variation of the response relative to the mean. Generally, the higher the R-Squared, the better the model fits the data. In the following formula, $\hat{y}$ is the predicted value of y and $\bar{y}$ is the mean value of y.

$$R2 = 1 - \frac{\sum(y_i - \hat{y})^2}{\sum(y_i - \bar{y})^2}$$