PARKING FINDER MOBILE APPLICATION

BY

THAM CHEE MING

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 2025

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Mr. Tou Jing Yi, who has given me this bright opportunity to engage in a deep learning and mobile application development project. It is my first step in establishing a career in this field. A million thanks to you.

I want to thank all my friends for their patience, unconditional support, and for standing by my side during hard times. Finally, I must say thanks to my parents and family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

With the rise in the number of car owners in fast-growing metropolitan areas, the need for effective parking solutions is becoming more demanding. This project proposes a Parking Finder Mobile Application that will provide real-time information about parking space availability and the parking finding status of vehicles in the parking lot. In this system, computer vision and deep learning models such as YOLOv8 will be utilized for parking spaces and vehicle detection while the DeepSORT algorithm is implemented to track vehicle movement in real-time. The proposed solution tackles the limitations that existing parking systems have including the high cost of implementation and lack of real-time vehicle monitoring. Combining parking space detection with vehicle tracking, the program will shorten the parking search times and improve user experience using a colour-coded status indicator and a simple interface. It is anticipated that such an approach would optimize parking space utilization in cities and promote urban mobility.

Area of Study (Minimum 1 and Maximum 2): Mobile Application Development, Computer Vision

Keywords (Minimum 5 and Maximum 10): Mobile Application, YOLOv8, DeepSORT, Vehicle Parking, Navigation

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

R-CNN	Region-based Convolutional Neural Network
YOLO	You Only Look Once
iOS	Iphone Operating System
RPN	Region Proposal Network
AP	Average Precision
mAP	Mean Average Precision
UAV	Unmanned Aerial Vehicle
CIoU	Complete Intersection over Union
DFL	Distribution Focal Loss
SORT	Simple Online and Realtime Tracking
KCF	Kernelized Correlation Filter
FFT	Fast Fourier Transform
RMSE	Root-Mean-Square Deviation
FairMOT-MCVT	Fairness of Detection and Re-Identification in Multiple Object Tracking – Multi-Camera Vehicle Tracking
MSDA	Multi-scale Dilated Attention
MOTA	Multi-Object Tracking Accuracy
FPS	Frame Per Second
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
JSON	JavaScript Object Notation
CCTV	Closed-Circuit Television

SDK	Software Development Kit
UTAR	Universiti Tunku Abdul Rahman
IOT	Internet of Things

CHAPTER 1

Introduction

As we stepped into the 21st century, the cities grew and car ownership became widespread which makes the demand for parking spaces increase significantly. In an urban setting, looking for a vacant parking space is a tough challenge that every driver encounters. Especially in denser populated areas such as towns and campuses, the shortage of parking spaces causes an increase in frustration and time wasted for every driver in searching for one. This scenario worsens at peak hours or in the case of some special event where the demand for parking spaces gets maximized.

1.1 Problem Statement and Motivation

This problem is crucial as it can contribute to several impacts on life and infrastructure. Traffic congestion is one of the issues as great part of congestion in city traffic due to cars looking for parking and circling the area, this not only contributes to traffic congestion but also affects environmental sustainability since the gas of cars circling for parking adds to air pollution. Moreover, drivers not finding a vacant spot will get frustrated and stressed leading to aggressive driving habits as well like double parking or off-road parking, this in turn poses serious safety issues for other road users.

Without real-time parking information, drivers will have to cruise around the parking spaces adding to traffic congestion and environmental pollution. There are a few existing solutions that solve the parking issues in many places; however, the current existing solutions are costly and time-consuming [1]. The existing solutions mostly use in-ground or surface-mount sensors and surveillance footage. These solutions have high installation and maintenance costs and surveillance footage requires segmenting the video frame manually which is time-consuming.

Nowadays, most of the current existing parking finding applications only display areas that have an open space, as well as give directions to get there. These systems, however, do not consider the current information concerning other vehicles that might be travelling in the same direction. When a group of vehicles arrive at one location and it is already occupied, it may lead to frustration and loss of time due to the lack of coordination. This is due to the absence of real-time vehicle condition

monitoring which would indicate whether a car nearby is actively searching for an available space or is going to leave the parking lot.

The motivation of this research is to address the costs and inefficiencies of the current parking detection systems. As urban growth and car ownership patterns continue to expand, there is a rising demand for more effective and affordable solutions that will provide drivers with real-time parking information. Reducing the time spent searching for available parking space can contribute to decreasing traffic jams, lower pollution levels, as well as improve general road safety. Additionally, the development of a more efficient parking detection system may enhance the use of existing parking facilities thus reducing additional parking space needs while maintaining valuable urban land.

1.2 Project Objectives

The main objectives of developing this parking finder mobile application are focused on enhancing parking management efficiency, improving user experience, and providing real-time smart navigation. The objectives are as follows:

Real-Time Parking Space Vacancy Detection

By using computer vision and deep learning technologies, the system able to detect, identify and show the availability of the parking spaces in a parking lot in real-time accurately. This is to ensure that the user able to receive the real-time update for parking spaces vacancy.

Vehicle Detection and Tracking with Parking Status

This system will detect and track the vehicles in the parking lot and determine the parking finder status of the vehicles such as the vehicle still actively searching or leaving the parking lot based on the movement of the vehicle. With this system, it can provide real-time insight into other vehicles' movement within the parking lot for the user.

Smart Parking Navigation System

By integrating both detection and tracking system into the mobile application, the application able to provide a smart navigation system for the user to the nearest vacant parking space using the result data from the detection and tracking systems. This smart navigation system calculates and navigates user to nearest vacant parking space based on the real-time parking spaces vacancies and other vehicles' movement and status in the parking lot.

1.3 Project Scope and Direction

This project's scope includes designing, creating, and implementing a smart parking system that gives users effective, real-time information on parking space availability. The system's goal is to improve overall parking efficiency and user experience by integrating computer vision techniques, tracking algorithms, cloud-based data management, and mobile application development.

The detection model is the project's first component. In order to detect vehicles and parking spaces, this module process video input retrieve from the parking lot. Custom datasets are used to train a deep learning-based detection framework that can detect vehicles in a variety of scenarios, including partial occlusions, changing lighting, and different vehicle variations. In order to give users precise information about available spaces, the detection model is also made to interpret the occupancy status of preset parking spaces.

The second part is the tracking system, which maintains constant vehicle monitoring over several frames by integrate on the detection results. Each detected vehicles are given a unique identification through the integration of tracking algorithms, which allows the system to identify whether a vehicle is finding or exiting a parking space. This continuous tracking approach enhances reliability compared to detection alone, as it minimizes false updates caused by temporary occlusions or detection errors. The data produced by the tracking system serves as the foundation for accurately determining the real-time occupancy status of each parking space.

The third element is the smart navigation system of the mobile application. This feature transforms the unprocessed tracking and detection data into useful recommendations for users. With the real-time database updates, the application determines navigation routes and displays the availability of parking spaces while

mapping vehicle locations within the parking lot. This enables users to be directed to the closest parking spot efficiently. Users can easily follow parking instructions with the application's user-friendly interface, which makes use of visual cues like colours and routes.

Finally, the system integration ensure that each module works together. The tracking and detection modules are backend-based and communicate with the mobile application continuously as they are connected to a cloud-based database service. This enhances convenience and time efficiency by offering real-time synchronization, allowing users to search for available parking space easily.

1.4 Contributions

This project makes several significant advancements in the field of smart parking management. First, creative and affordable alternatives are presented that reduce reliance on expensive infrastructures. Unlike current systems that often rely on IoT sensors, RFID tags, or physical hardware embedded in each parking lot, this research uses computer vision and deep learning techniques. By utilizing software-based detection algorithms and pre-existing camera infrastructure, the solution lowers installation expenses and continuous maintenance requirements. This approach not only reduces the system's cost but also increases its scalability, allowing it to be implemented across larger parking lots without incurring significant costs.

Another noteworthy addition to this project is the use of real-time data to improve user experience. The system provides incredibly accurate updates on parking space availability by combining tracking and detecting technologies to prevent temporary occlusions or detection errors. The mobile application, which allows users to monitor available spaces, track vehicle positions, and receive routing to the nearest parking space, is directly supported by this real-time feature. The solution reduces traffic congestion caused by prolonged searching, speeds up the process of finding an available space and saves users a significant amount of time through an easy-to-use mobile interface.

1.5 Report Organization

This report includes a few chapters to show the details of the project. In chapter 1, this chapter mainly introduces the introduction of the problem, stating the project scope and objectives, and contribution of this project. Next, technologies, existing systems and application with their strength and weaknesses are reviewed in chapter 2. Moreover, the methodologies and system flows are covered in chapter 3. In chapter 4, this chapter focused on the system design of this application for this project while chapter 5 described how this system and application was implemented. Next, the system evaluation and discussion are presented in chapter 6. Lastly, chapter 7 are the conclusion and recommendation for future enhancements for this project.

CHAPTER 2

Literature Reviews

This chapter gives a summary of previous research, relevant technologies, and existing systems that serve as the project's framework. In addition to highlighting how the suggested system expands upon or enhances existing efforts, it looks at what has previously been created in the field and points out any gaps or limits in existing solutions.

2.1 Review of the Detection Technologies

2.1.1 Previous works on Faster R-CNN and RetinaNet

Faster R-CNN introduced by Ren et al. in the year 2015, is an extremely precise two-stage object detection model and is an enhancement of the R-CNN architecture (Figure 2.1.1) that generates high-quality region proposals rapidly by using RPN. During its second stage, these proposals get further classified and refined, making significant advances in terms of detection accuracy, especially for partially occluded or crowded items.

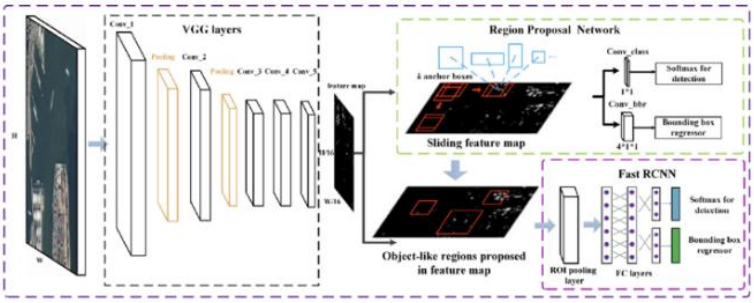

Figure 2.1.1: Faster R-CNN network structure [2]

Alternately, RetinaNet that is introduced by Lin et al. in the year 2017 is a single-stage object detection model that achieves a balance of speed and accuracy. This model included a focus loss function which can solve the class imbalance problem that reduces the performance of the single-stage model [1]. With this development, RetinaNet can assist in faster inference times while attaining accuracy levels comparable to Faster R-CNN. RetinaNet is more useful in real-time applications for object detection. However, Faster R-CNN is more accurate when the scene contains densely packed objects.

To research the effectiveness of RetinaNet and Faster R-CNN in detecting parking space availability, a study was carried out by Padmasiri et al. using surveillance footage. This study found that RetinaNet outperformed Faster R-CNN in identifying unoccupied parking spaces with higher recall and accuracy. Therefore, RetinaNet would be more suitable for situations where the detection of unoccupied parking is important. Nevertheless, it can be shown that the Faster R-CNN and RetinaNet with ResNet-101 backbone have similar performance but are better than RetinaNet with ResNet-50 backbone in detecting occupied parking spaces due to their better object localization and more detailed region proposal suggestions [1].

Table 2.1.1: Class-wise Precision for Object Detection [1]

Model	AP-occupied	AP-unoccupied
RetinaNet (ResNet-50+FPN)	21.03	19.28
RetinaNet (ResNet-101+FPN)	25.48	15.78
Faster RCNN (ResNet-50-C4)	25.46	11.23

2.1.2 Previous works on YOLO (You Only Look Once)

Improved YOLOv3

YOLOv3 uses a feature pyramid network and thus has an advantage over its predecessors as it can recognize at three different scales more competently. Nevertheless, some aspects can still be improved in its performance. Figure 2.1.2 shows the network structure of standard YOLOv3.

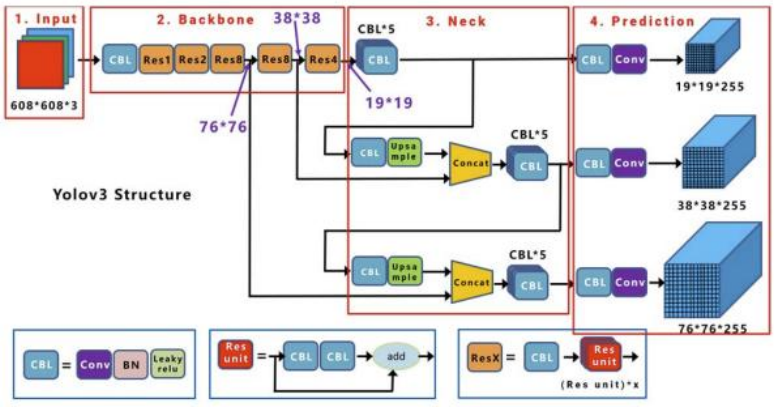

Figure 2.1.2: Standard YOLOv3 network structure [3]

One of the papers proposed a few modifications aimed at enhancing the accuracy of the standard YOLOv3 framework. The attention mechanism is one of the

main improvements where it assists in focusing on relevant picture components resulting in increased detection performance, especially in complex situations when items may overlap with each other or if there are disarranged backgrounds. This technique makes it possible for the model to ignore less relevant background information and focus on the most important sections such as cars and parking spots. This can reduce challenges in recognizing parking spaces which involves being unable to distinguish between the occupied or unoccupied spaces. The result in this paper shows that employing an attention mechanism in the improved YOLOv3 algorithm led to marked improvements in detection accuracy and precision in comparison with the original YOLOv3 [3].

Besides, another paper proposed an integration of YOLOv3 with MobileNetv2 for detecting parking space occupancy. This combination allows the model to process real-time video streams, detect cars and detect parking space occupancy in varying environmental conditions. The authors used a dataset that includes pictures with different backgrounds to test the YOLOv3-MobileNet model and the results show that it can accurately identify vehicle parking status. MobileNet was used to drastically reduce the model size and increase its detection speed without affecting its accuracy. Balancing between precision and efficiency is crucial for such systems' successful deployment into real-life scenarios [4].

Improved YOLOv5

YOLOv5 uses PyTorch for more accurate and quicker deployment while being an enhanced version of the previous YOLO model. Nevertheless, some aspects can still be improved in its performance. Figure 2.1.3 shows the network structure of standard YOLOv5.

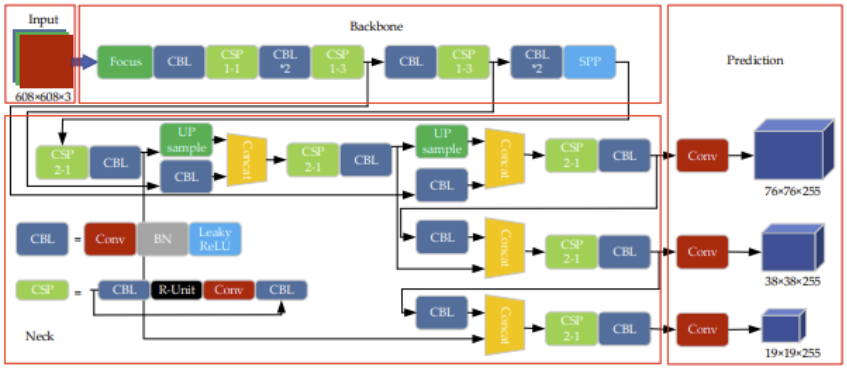

Figure 2.1.3: Standard YOLOv5 network structure [5]

Chang et al. propose several other innovations to improve the YOLOv5 model, but the Flip-Mosaic algorithm is a major improvement in this implementation. The traditional Mosaic algorithm only randomly adds 3 images from the dataset for each image and randomly finds the flattening point in a blank image and the 4 segments of imagery were created by utilizing the intersection point, while the additional parts were discarded (Figure 2.1.4). This Flip-Mosaic algorithm has improved the model's ability to detect smaller car sizes by using the traditional Mosaic algorithm, flipping the 4 images randomly and mosaicing during training (Figure 2.1.5). In addition to this, it reduces occlusion effects and helps the model learn more robust features. The results show that the Flip-Mosaic algorithm significantly increases mAP scores for small cars and those that are partially occluded by other objects, making it better than standard YOLOv5. The improved YOLOv5 obtained greater precision and recall rates with the contribution to a reduction in false positives from the algorithm [5].

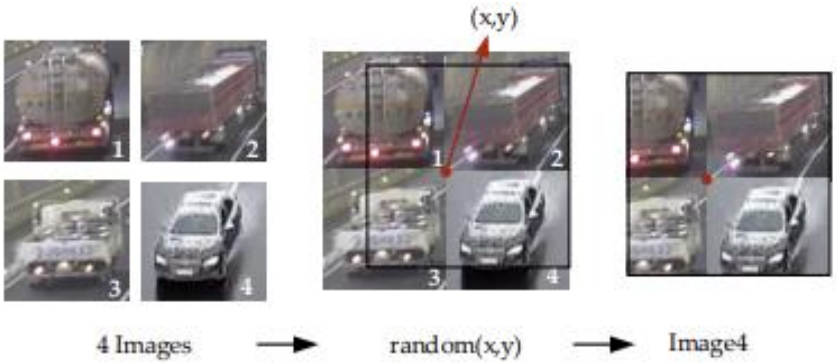

Figure 2.1.4: Traditional Mosaic Algorithm [5]

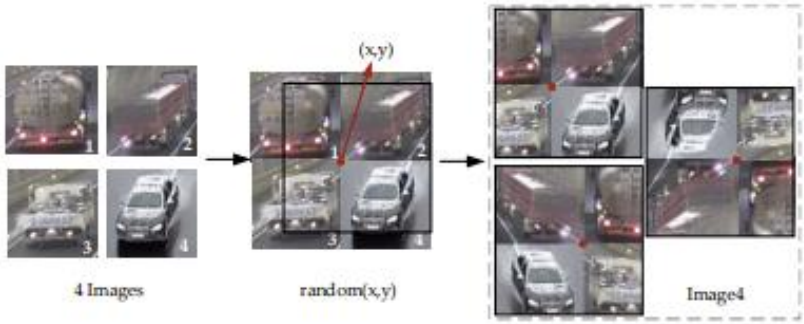

Figure 2.1.5: Flip-Mosaic Algorithms [5]

YOLOv8

In comparison to YOLOv5, YOLOv8 is a stronger tool for object detection tasks, particularly in complex scenarios like car identification from satellite images. One of

the significant improvements is the addition of a Feature Pyramid Network (FPN) and Path Aggregation Network (PAN), which makes it possible for YOLOv8 to detect objects more successfully at various scales and resolutions. Figure 2.1.6 shows the network structure of YOLOv8.

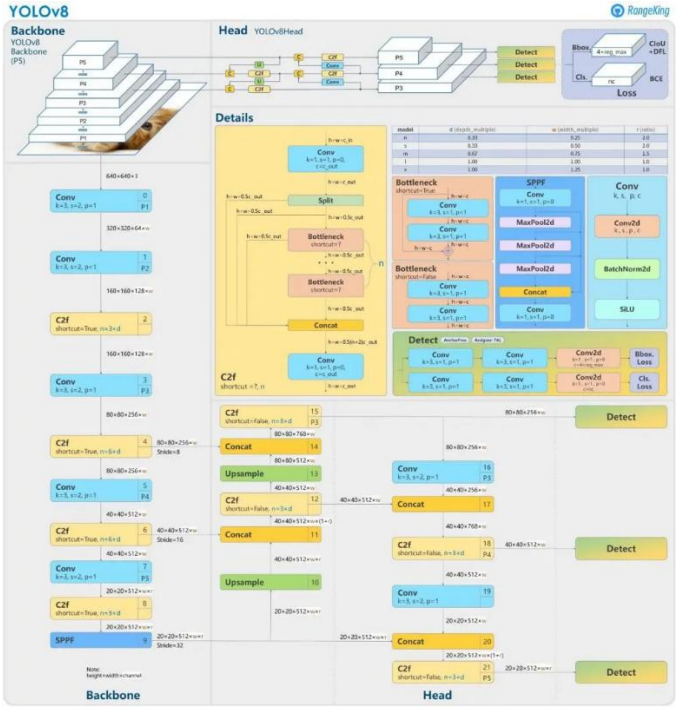

Figure 2.1.6: YOLOv8 network structure [6]

In addition, YOLOv8 comes with an advanced tagging tool that contains auto-tagging, shortcut tagging, and hotkey customisable features that assist in speeding up the labelling process. The model also uses advanced loss functions such as Complete Intersection over Union (CIoU) and Distribution Focal Loss (DFL) which improve the accuracy of bounding boxes, especially for small-sized items. Furthermore, YOLOv8 can be used for a variety of purposes since it supports different tasks like tracking, posture estimation and segmentation. Users can also choose between YOLOv8 nano (YOLOv8n) to YOLOv8 extra-large (YOLOv8x) based on their performance and processing power needs.

However, Table 2.1.2 represents the result from this research and shows that in terms of recall and precision, YOLOv5 always outperforms YOLOv8 resulting in lower numbers of false negatives or false positives. The F1-score comparison is where this is most notably seen as YOLOv5 surpasses YOLOv8 by approximately 2.1%, thus making YOLOv5 the better choice when it comes to situations that necessitate reducing misclassification rates. Moreover, even though the new design of YOLOv8 provides

more capabilities, it also has increased processing demands which may be problematic for real-time applications or devices with lower processing capabilities. With greater precision and a less complex structure, there are still scenarios where YOLOv5 could be a more appropriate option when resource constraints are an issue.

Table 2.1.2: Performance comparison for YOLOv5 and YOLOv8 [6]

Approach	Total	TP	FP	FN	TN	Accuracy	Precision	Recall	F1-Score
YOLOv5	1139	1105	139	61	35	0.928	0.947	0.969	0.958
YOLOv8	1154	1086	129	80	64	0.894	0.931	0.944	0.937
YOLOv5 (not FP)	1139	1105	139	0	35	0.972	1	0.969	0.984
YOLOv8 (not FP)	1154	1086	129	0	64	0.949	1	0.944	0.971

TP = true positive; FP = false positive; FN = false negative; TN = true negative; accuracy = TP + TN/TP + TN + FP + FN; precision = TP/TP + FP; recall = TP/TP + FN; F1 score = 2* precision * recall/precision + recall.

2.1.3 Summary of the Detection Technologies

One of the limitations of the previous works of object detection is the detection challenges in complex environments. Many different methods including RetinaNet, improved versions of YOLOv3, and YOLOv5 are weak in identifying small objects especially when they are far away or partly covered. The next limitation is there are computational and resource limitations. The models such as Faster R-CNN and the improved versions of YOLO that include an attention mechanism and Flip-Mosaic algorithm will require vast computing resources due to their complex structures and additional features. Lastly, the exchange between accuracy and speed is also one of the limitations of object detection in previous works. For example, speed compromises the accuracy in speed-oriented models like any YOLO version especially when detecting tiny and concealed objects. On the other hand, despite having more precision, models such as Faster R-CNN have considerable disadvantages for real-time applications since their inference times are too long.

The best model for precise and effective real-time parking detection in this project is YOLOv8, which offers enhanced multi-scale detection, advanced loss functions, and flexible task support.

2.2 Review of the Tracking Technologies

2.2.1 Previous Works on DeepSORT

DeepSORT is an enhanced version of the SORT algorithm that merges the Hungarian method for data association with the Kalman Filter for motion prediction. Nevertheless, it does not consider appearance information which might lead to identity swapping during tracking. DeepSORT addresses this issue by using a deep learning-based appearance descriptor that allows object identities to be preserved despite having similar-looking objects and occlusions.

Integration of YOLOv3 with DeepSORT allows fast and accurate object detection abilities with reliable tracking capabilities. YOLOv3 is responsible for identifying all objects in every frame of a video by providing bounding boxes as well as class probabilities. Afterwards, these detections with bounding boxes are sent to DeepSORT and use the Kalman filter to predict each object's subsequent place then use the Hungarian algorithm to associate detections with tracks that already existed [7]. In addition, the deep appearance characteristics enable DeepSORT to maintain item IDs across frames.

In this paper, bathija et al. employed SORT for tracing and YOLOv3 (Figure 2.2.1) for detection and was applied to custom datasets, it showed high accuracy and real-time performance with the combination. The experiment shows the advantages of optimizing the system and suggests that more trackers, like DeepSORT, could be integrated into the existing framework to enhance tracking efficiency [8]. On the other hand, Kapania et al. combined DeepSORT and YOLOv3 (Figure 2.2.2) for tracking multiple objects in drone-captured aerial images. Such a combination was required to address issues of occlusions and motion blur, as well as small sizes that come with high altitudes at which drones fly. The authors demonstrated how the tracker's precision was improved using YOLOv3 and DeepSORT to facilitate real-time tracking of many small objects within UAV contexts [7].

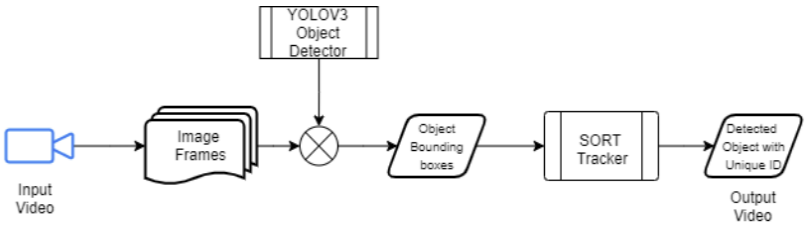

Figure 2.2.1: Flowchart for Visual Object Detection and Tracking [8]

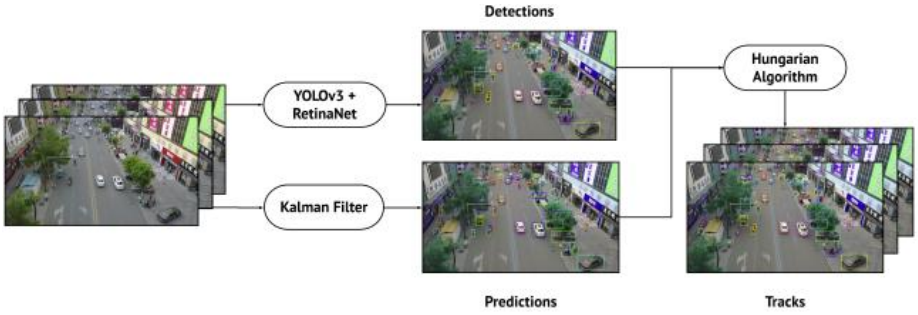

Figure 2.2.2: Kapania et al. Proposed Architecture [7]

There are a few advantages that come with using DeepSORT such as solving the identity shifts issue. Identity shift is a common issue in multi-object tracking especially under cluttered conditions or when objects look alike, by adding appearance features. Another advantage is scalability, in different real-time applications including UAV-based tracking, traffic monitoring and surveillance, DeepSORT can scale up and work efficiently.

2.2.2 Previous Works on Kernelized Correlation Filter (KCF)

The tracking method proposed in this paper is a Kernelized Correlation Filter (KCF) (Figure 2.2.3) [9]. This method was designed to have a fast and efficient manner of tracking cars from UAV recordings. First, a set of vehicle images is used for tracker training for the KCF method to reduce regularization risks through correlation function optimization. Thus, it employs a kernel function that helps map input vehicle images into a feature space, which makes it possible to deal with non-linear patterns in the vehicle data. To achieve effective tracking, this technique can calculate correlations using the Fourier domain based on the Fast Fourier Transform (FFT) along with circulant matrices and kernelized techniques. Such guarantees that each new frame recognizes the position of the respective automobile accurately and rapidly.

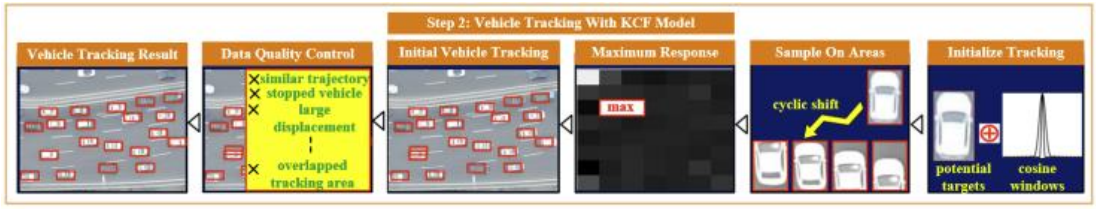

Figure 2.2.3: KCF Model Tracking Steps (from right to left) [9]

The advantages of the KCF are this algorithm for vehicle tracking extraction is very precise with an RMSE of 0.175m and a Pearson correlation of 0.999 and it is an ideal one for detailed traffic studies. By using UAVs, occlusions are minimized and

wide coverage and clear picture are offered. Even when the vehicles are partially occluded, the KCF algorithm ensures proper as well as effective tracking while the wavelet transform effectively smoothens the trajectories providing seamless results. However, the disadvantage of KCF is when visibility is low like at night or during bad weather, this algorithm will encounter difficulties due to it relying on a static camera angle, there is no way to adjust for dynamic UAV movements. In addition to being ineffective, manual curve fitting along lanes may not perform well in complex or high-volume traffic situations.

2.2.3 Previous Works on FairMOT-MCVT

The FairMOT-MCVT method (Figure 2.2.4) proposed in this paper [10] contains different important developments to improve the efficiency and accuracy of vehicle tracking across non-overlapping focus spaces. The FairMOT-MCVT technique's core part is its Block-efficient module, which improves the feature extraction process significantly. This module includes depth-separable convolutions with a multi-branch structure to enhance the detection of small and far-off vehicles. The structure needs to be there so that more detailed image characteristics can be captured by the network which is important for tracking cars over long distances or in low-visibility areas. Furthermore, minimising computational overheads, focusing on important areas within the image as well as concentrating on key locations are some of the ways through which the Multi-scale Dilated Attention (MSDA) module helps in improving the model's feature extraction ability. Thus, these aspects ensure that real-time vehicle tracking is possible without speed loss due to computational complexity.

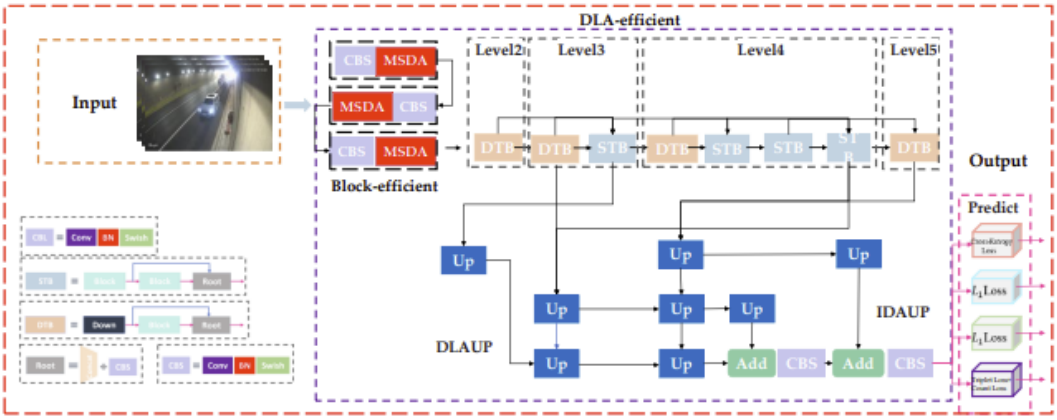

Figure 2.2.4: The Structure of FairMOT-MCVT [10]

One of the main contributions of FairMOT-MCVT is it considers position and velocity continuity in the optimization process using a joint loss function. The loss function helps to maintain a consistent trajectory of the vehicles even when they are obscured from view or have similar appearances by incorporating trajectory smoothing as well as velocity consistency. Additionally, the Re-ID branch enhances this as it enables the algorithm to distinguish between vehicles with similar characteristics. This minimizes the chances of identity switches, which is a common issue in multi-camera tracking.

In this research, the authors used the UA-DETRAC dataset to evaluate this FairMOT-MCVT method and the results in Table 2.2.1 show that it has an improvement in tracking accuracy compared to other tracking algorithms. This algorithm has a MOTA of 79.0, IDF1 score of 84.5, and can process the video at 29.03 FPS which can indicate that it's suitable to use for real-time applications.

Table 2.2.1: Result of Different Tracking Algorithm Experiments [10]

	MOTA	IDF1	MT	ML	IDS	FPS
SORT	70.3	79.6	145	4	65	25.10
DeepSORT	74.5	82.4	153	3	55	26.88
FairMOT	77.7	84.2	160	4	47	28.35
CenterTrack	77.0	84.6	155	4	50	27.55
RobMOT	76.0	83.0	150	5	53	27.00
MTracker	75.5	82.7	152	4	54	26.75
FairMOT-MCVT	79.0	84.5	159	4	45	29.03

2.2.4 Previous Works on Bounding-Box-Based Tracking Algorithm

The Bbox-based vehicle tracking algorithm [11] proposed by the authors in this study is a robust tool for tracking and re-identifying cars across video sequences. Initially, the algorithm extracts the bounding box data of the detected vehicles, which includes their width (W), height (H) and centre coordinates (X, Y). To link vehicles within frames it computes the Euclidean distance (Eq.1) between bounding boxes from two consecutive frames. This way, Euclidean distance helps to identify good matches of vehicle pairs thus ensuring that the tracker can follow every car's movement in time. This process involves creating a distance matrix that is sorted to determine which pairs of identified vehicles in subsequent frames are closest to each other.

$$B_i : \left[\left[X_0^{center}, Y_0^{center}, W_0, H_0\right], \ldots, \left[X_n^{center}, Y_n^{center}, W_n, H_n\right]\right]$$
$$B_j : \left[\left[X_0^{center}, Y_0^{center}, W_0, H_0\right], \ldots, \left[X_m^{center}, Y_m^{center}, W_m, H_m\right]\right]$$
$$D_{ij} = \sqrt{\sum_i \sum_j \left(B_i - B_j\right)^2}, \; i \in \left[0; n\right] \; and \; j \in \left[0; m\right] \tag{1}$$

Bbox-based vehicle tracking algorithm has a feature of tracking the appearance and disappearance timing which is intended to deal with situations where vehicles temporarily disappear due to obstructions, external elements or even shaken cameras. If the target leaves for some time, the program predicts its next location using linear equations (Eq.2) based on the previous trajectory of the car. Hence, the prediction step makes the tracking algorithm more robust against temporary disturbances in vehicle visibility as it allows for the system's knowledge of probable places where the vehicle can come into view again. According to this paper if a vehicle is out of sight for a long time (beyond 100 frames), then it is considered lost by the algorithm and removed from active surveillance.

$$d_x = \frac{\sum_{i=0}^{n} x_i}{n}, \; d_y = \frac{\sum_{i=0}^{n} y_i}{n}, \; d_w = \frac{\sum_{i=0}^{n} w_i}{n}, \; d_h = \frac{\sum_{i=0}^{n} h_i}{n},$$
$$x_{n+1} = x_n + d_x, \; y_{n+1} = y_n + d_y, \; w_{n+1} = w_n + d_w,$$
$$h_{n+1} = h_n + d_h, \; B_{predict} = \left[x_{n+1}, y_{n+1}, w_{n+1}, h_{n+1}\right] \tag{2}$$

This novel proposed Bbox-based vehicle tracking algorithm has several advantages. It is computationally efficient since it only utilizes geometric information rather than pixel, shape or colour data which are more sophisticated and slow methods of tracking. In addition, it resists common issues faced in real-time videos such as camera shaking and occlusion due to its predictive nature. However, the tracker does have some limitations. In crowded or fast-moving environments where bounding boxes overlap, accurate tracking may be difficult. Furthermore, when vehicles suddenly change direction, the linear motion prediction may not apply anymore thus causing tracking errors under challenging conditions.

2.2.5 Summary of the Tracking Technologies

There are a few limitations of object tracking in the previous works. The first limitation is it greatly dependent on the detection system. If the detection system is unable to detect and locate the vehicles, there will be no bounding box for the tracking system to track the object. Therefore, the tracking system required a very accurate detection system to track the vehicles. The next limitation is low performance in poor visibility. For example, during low brightness situations, fog, and rain the tracking system will face difficulties in tracking vehicles due to tracking depending on clear visual. Lastly, difficulty in tracking in a packed area is also one of the limitations. This is because when two or more vehicles are near each other, it will cause overlapping and tracking will more likely have inconsistent tracking.

In the end, DeepSORT is the recommended option for this project because it reduces identity shifts and ensures dependable real-time tracking by fusing motion prediction with deep appearance descriptors.

2.3 Review of the Existing Systems/Applications

2.3.1 Parking Finder Application for Intelligent Parking System

The parking finder application proposed by the author [12] uses a system that is integrated with hardware and software. The author chose ultrasonic sensors with Arduino as a detection method for reliability and real-time accuracy and Raspberry Pi as a server. The flow of the proposed method is the sensors first detect the parking occupancy and the information is transmitted wirelessly through the Raspberry Pi and saved information in cloud storage. The system uses Firebase for cloud storage and the information is saved in the format shown in Figure 2.3.1.

Figure 2.3.1: Field device data from the firebase [12]

In the Firebase the vacancy status of the parking spaces is saved as “V” for vacant and “O” for occupied which later be used in the mobile application for parking status. The author developed the parking finder application in Android Studio using Java programming language. The parking finder application consists of features such as an authentication page and interface for parking space status display. The parking space's status is colour-coded where red indicators are for occupied and green indicators for vacant for user friendly interface.

In this parking finder application, the limitation was that the application only displayed parking lot information to the user. In the parking finder application developed by the author [12], as it only shows the availability of the parking spaces in the parking lot to the user, the user might not be able to fully utilize the vacant parking space information as some other vehicle might get to the vacant space before the user. This will waste user time and make users frustrated.

2.3.2 Parking Finder Application for Intelligent Parking System

Yindeesuk et al. [13] introduced the CarPark mobile application, a smart parking solution designed for Nakhon Ratchasima Rajabhat University. The system integrates IoT-based sensors installed at individual parking spaces to detect whether a space is occupied or available. Users able to check the availability of spaces in real time before they arrive at the parking lot because of the sensor data that is sent to a central server and instantly reflected in the CarPark mobile application. Another advantage of the system is that, in the event that a vehicle's owner is blocked, users can use the application to search the license plate and contact them directly. This function improves coordination and lessens driver disagreements. The mobile application improves efficiency and user experience by reducing the needless traffic flow inside the lot and the amount of time users spend searching for parking space. In terms of technology, this project shows how to effectively integrate IOT devices in conjunction with a mobile interface to provide precise and current parking information.

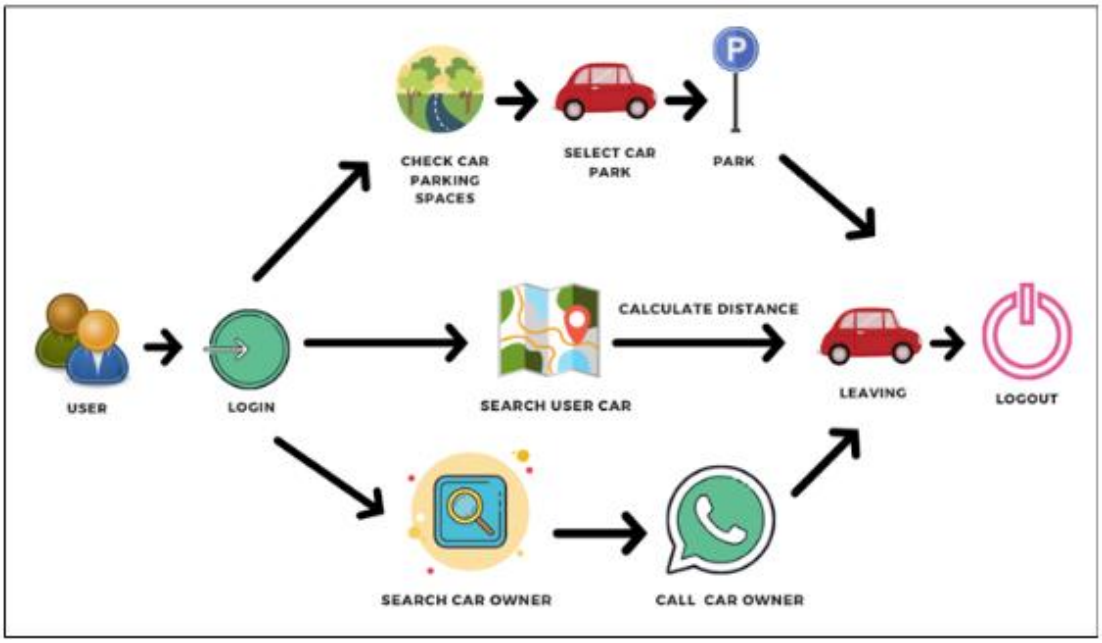

Figure 2.3.2: The flow of the CarPark Mobile Application.

Despite these advantages, this review also identifies a number of limitations. Due to the system's reliance on physical sensors, it is less feasible for large-scale installations like shopping centres, airports, or citywide parking systems because it is more expensive to build and requires constant maintenance. An additional drawback is that the app just offers parking availability status updates where it lacks of navigation and route assistance that can assist users in finding the closest available space quickly.

CHAPTER 3

System Methodology/Approach

This chapter outlines the system's general architecture and design. The system methodologies are presented using a variety of models and diagrams that illustrate the structure, user interactions, and operational workflows of the system before the actual implementation.

3.1 System Architecture

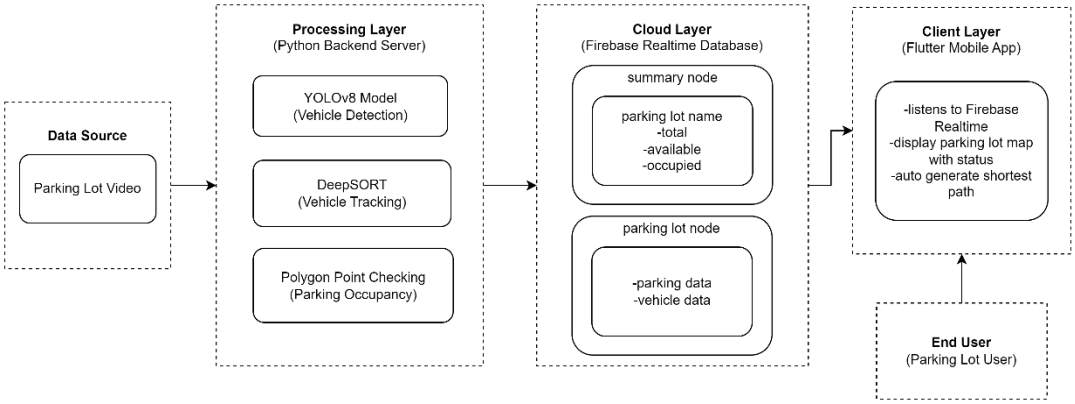

Figure 3.1: High-level system architecture diagram

In figure 3.1 shows that the smart navigation system and parking finder mobile application are modular, with each component being built, tested, and improved upon before being fully integrated. Model training, system integration, mobile application development, dataset preparation, and performance evaluation are some of the phases that make up the project. Datasets with a variety of parking lot and vehicle photos taken in a range of lighting, weather, and layout scenarios were first gathered from internet sources. These datasets were used to improve detection accuracy by fine-tuning and training a pre-trained YOLOv8 model with data augmentation approaches. Following model validation, the vehicle detection module was merged with the DeepSORT tracking algorithm. Parking spaces are manually mapped in this configuration, and the bottom-centre point of a detected vehicle's bounding box is used to calculate the occupancy of each space. This guarantees accurate and real-time updates on parking availability. A Firebase Realtime Database, which is organized into two primary nodes

where one of every parking lot and one for summary of all parking lot receives the processed results of detection and tracking.

Concurrently, the Flutter framework and the Dart programming language were used to develop a cross-platform mobile application. The app's user-friendly interface shows the parking layout, the locations of the vehicles, and the current parking space status (red for occupied, green for available). Using a graph-based Dijkstra's algorithm, its smart navigation system calculates and displays the shortest route from the user's location to the closest available space while taking into consideration vehicles ahead of them.

3.2 Use Case Diagram

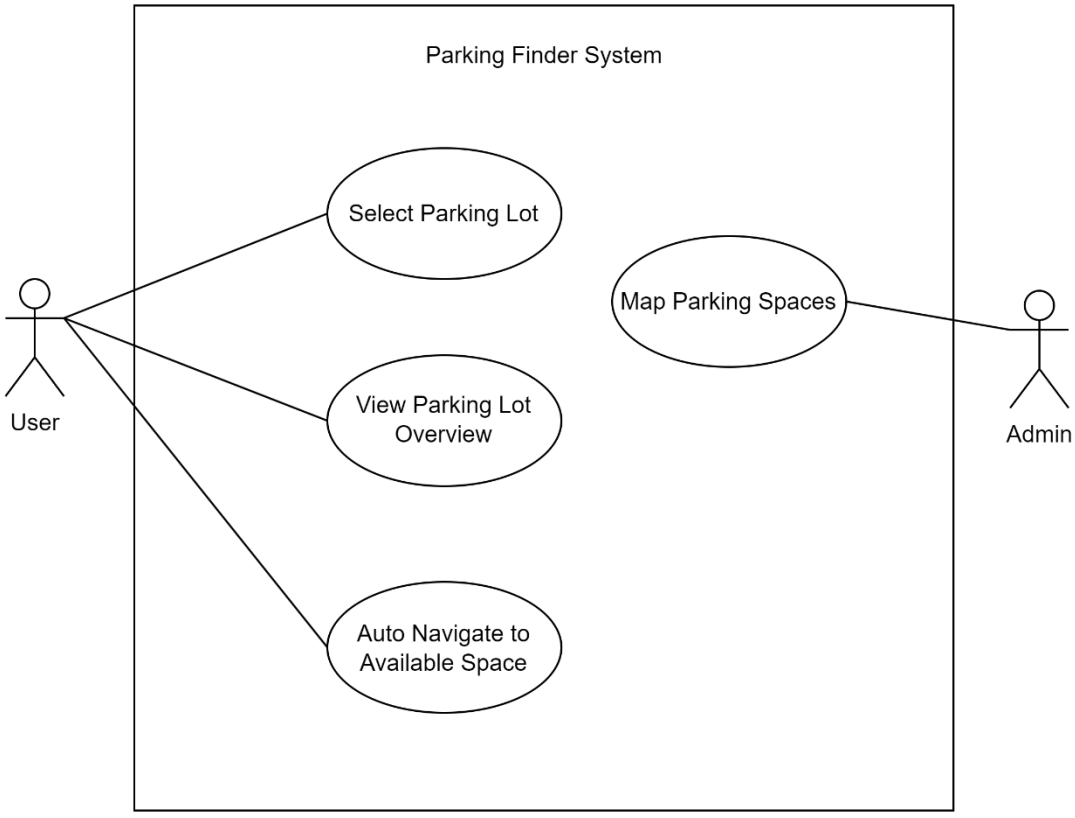

Figure 3.2: Use case diagram of the parking finder system

Figure 3.2 shows the use case diagram that illustrates the interactions between two main actors, the user and the admin, with the system. The user represents the driver who makes use of the mobile application to find and navigate to an available parking space. Through the system, the user can select a parking lot of their choice, view the parking

lot overview, and automatically navigate to an available space. When selecting a parking lot, the user chooses from a list of available lots, and upon selection, the system loads the chosen lot for further use. Once selected, the user can view the parking lot overview, which displays the layout together with the real-time status of spaces, showing which are occupied and which are available. The system also supports automatic navigation, where the user is guided to the nearest available space through a calculated path that take consideration of other vehicles status along the path.

On the other hand, the admin is responsible for mapping parking spaces within the backend system and configuring the layout inside the code. This task includes defining the structure of roads and parking spaces so that the system can accurately represent the real parking environment. Once the mapping is completed, the layout is linked with the application logic, ensuring that users can access it for navigation and real-time monitoring. By handling this responsibility, the admin ensures that both the backend data and the application code remain consistent, which is essential for accurate pathfinding and reliable parking space updates.

3.3 Use Case Description

3.3.1 Select Parking Lot

Table 3.3.1: Use Case Description for Select Parking Lot

Use Case ID	UC001	
Use Case	Select Parking Lot	
Purpose	To allow user select the parking lot user wanted to view the overview and use navigation feature.	
Actor	User	
Trigger	Launch the parking finder app and in homepage.	
Precondition	User is in the homepage of the Parking Finder App.	
Scenario	**Step**	**Action**
Main Flow	1	The user launches the Parking Finder Mobile Application.
	2	App load and connect to Firebase.

	3	System displays list of parking lot available in homepage.
	4	User allowed to select any parking lot.

3.3.2 View Parking Lot Overview

Table 3.3.2: Use Case Description for View Parking Lot Overview

Use Case ID	UC002	
Use Case	View Parking Lot Overview	
Purpose	To allow user to view the overview of the parking lot selected such as parking spaces status and vehicle location.	
Actor	User	
Trigger	User selects a parking lot layout from the homepage.	
Precondition	The Firebase Realtime Database must be running and synchronized with backend system.	
Scenario	**Step**	**Action**
Main Flow	1	User select any parking lot from homepage.
	2	System displays parking lot layout.
	3	System retrieves both parking and vehicle data from Firebase.
	4	System updates the map with the data retrieved periodically.
	5	User view the parking lot layout with parking spaces statuses and vehicle movement within the parking lot.

3.3.3 Auto Navigate to Available Space

Table 3.3.3: Use Case Description for Auto Navigate to Available Space

Use Case ID	UC003
Use Case	Auto Navigate to Available Space

Purpose	To guide user to the nearest available parking space with shortest path take consideration of other vehicle's status.	
Actor	User	
Trigger	User choose any parking lot from the homepage.	
Precondition	The parking lot layout is available in the system.	
Scenario	**Step**	**Action**
Main Flow	1	User chooses a parking lot from the homepage.
	2	App retrieves both parking space data and vehicle data from Firebase.
	3	App display parking lot layout with parking space status and vehicle movement status.
	4	System calculates the shortest path using algorithm.
	5	System highlighted shortest path with green colour.

3.3.4 Map Parking Space

Table 3.3.4: Use Case Description for Map Parking Space

Use Case ID	UC004	
Use Case	Map Parking Space	
Purpose	To allow admin to define and update the structure of the parking lot	
Actor	Admin	
Trigger	The admin configures or modifies a parking lot layout.	
Precondition	Have access to backend system and codebase.	
Scenario	**Step**	**Action**
Main Flow	1	The admin defines nodes representing roads and parking spaces in the system code.
	2	Admin add edges to connect the nodes, forming a directed graph of the parking lot.
	3	The updated layout is made available for user navigation and monitoring.

3.4 Activity Diagram

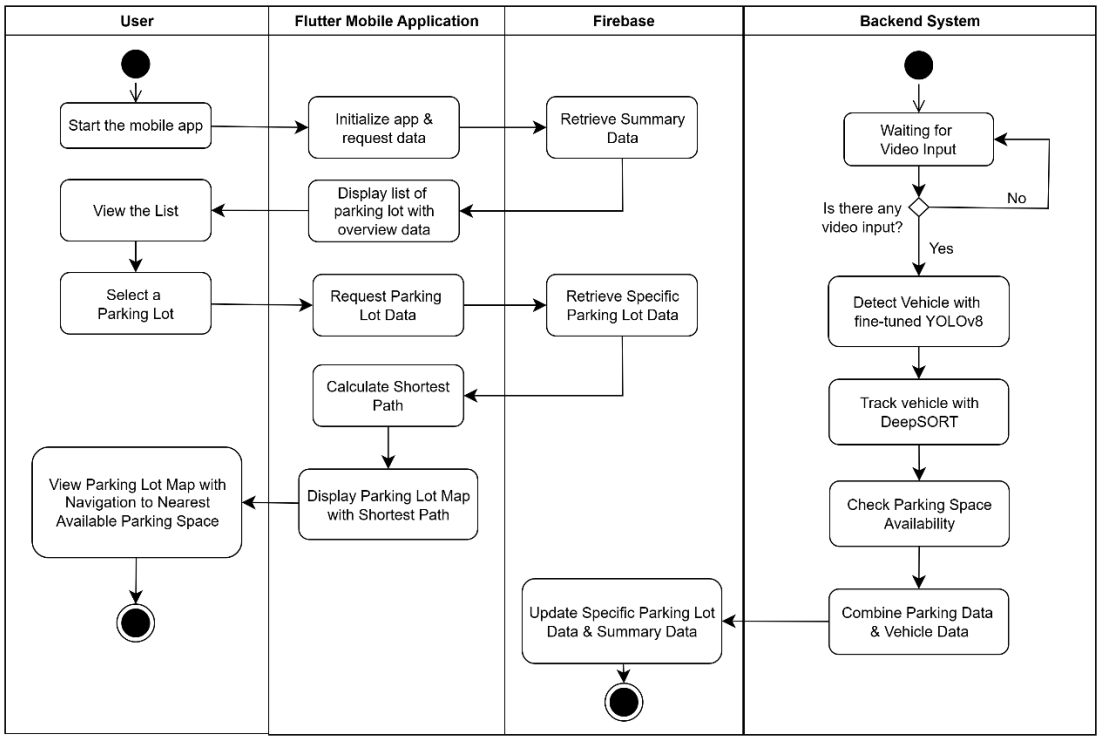

Figure 3.3: Activity Diagram for Parking Finder Mobile Application.

Figure 3.3 shows the interaction between the user, Flutter mobile application, Firebase, and backend system. When the user launches the mobile application, it initializes and request summary data from Firebase. The application then displays a list of parking lots with overview information after fetching the required summary data from Firebase. When a user chose a particular parking lot from the list, the app request Firebase again for parking lot information user selected. In response, Firebase retrieves and returns to the app the particular parking lot data. In the meantime, the backend system continuously analyses video input, utilizing DeepSORT to monitor and a fine-tuned YOLOv8 model to detect vehicles. It also checks the parking space availability within the parking lot and combines parking and vehicle data before updating Firebase with the latest summary and specific parking lot information. After receiving this information, the application calculates the shortest route to the closest parking space and shows the user a map of the parking lot with directions, which allow them view parking lot information visually and follow the path for easier parking finding.

CHAPTER 4

System Design

4.1 System Block Diagram

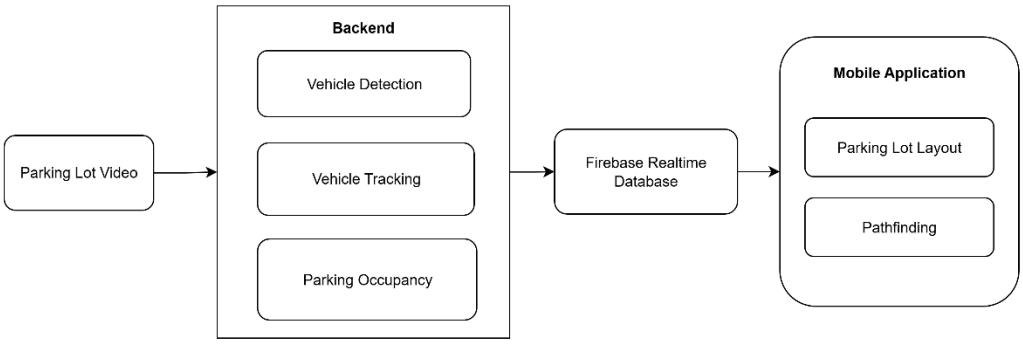

Figure 4.1: High-Level System Block Diagram

The system block diagram shown in figure 4.1 is the overall work flow of the Parking Finder Mobile Application. The video feed from the parking lot cameras is first processed using the YOLOv8 detection model to detect vehicles. The detections are then passed into the DeepSORT tracker, where IDs are assigned and vehicle identities are maintained between frames. Bottom-centre coordinates of bounding boxes are utilized within the occupancy checking module, where space availability is determined using predefined parking space coordinates as to whether a space is available or occupied. This processed data is formatted into space status, vehicle information, and summary statistics, which are uploaded continuously to Firebase Realtime Database. The Flutter mobile application then fetches this information in real time, providing users with a graphical parking map, vehicle location, and smart navigation instructions to the nearest available space.

4.2 System Components Specifications

The Parking Finder Mobile Application is built from several interdependent components that together enable real-time vehicle detection, status tracking, and mobile navigation assistance. Each component has its own specifications, covering the algorithms, technologies, and configurations that ensure its functionality.

4.2.1 Vehicle Detection and Tracking Module

The vehicle detection and tracking module is responsible for processing live video streams to detect and track vehicles within the parking lot. This module is built around the YOLOv8 object detection model, which has been fine-tuned using both public and custom datasets to improve detection accuracy in real parking environments. Each video frame is standardized through resizing to a fixed resolution and conversion into RGB colour format before being passed into the model for inference. Detection parameters, such as the confidence threshold and Intersection-over-Union (IoU) threshold, are configured to minimize false positives and handle overlapping bounding boxes effectively. The detection outputs, including bounding box coordinates and confidence scores, are then processed by the DeepSORT tracker, which assigns unique IDs to each vehicle and maintains consistent identities across frames. By combining motion prediction with appearance-based features, the tracker is able to follow vehicles reliably, even under temporary occlusions. To determine occupancy, the bottom-centre point of each bounding box is tested against pre-defined parking space polygons that are manually plotted during system setup. Alongside generating structured metadata such as vehicle ID, coordinates, status, and timestamp, the module also provides visual overlays of bounding boxes, IDs, and status labels during testing to enable verification of detection accuracy.

4.2.2 Firebase Realtime Database

The Firebase Realtime Database serves as the central communication component that connects backend detection processes with the mobile application interface. It operates as a NoSQL, JSON-based database designed to maintain live synchronization between multiple system components. Within each parking lot node, the database maintains two primary structures which are a parking_data node that stores parking space information such as unique identifiers, current occupancy status, and timestamps of the latest updates, and a vehicle_data node that stores details of all tracked vehicles, including assigned IDs, current coordinates, statuses, and update timestamps. In addition, a summary node provides an overview of parking lot conditions by storing aggregated data such as the number of available and occupied spaces. Through Firebase's real-time synchronization, any updates made by the detection and tracking module are instantly

reflected on mobile application, ensuring that vehicle movements and parking space changes are available within milliseconds.

4.2.3 Mobile Application

The mobile application serves as the user-interface component of the system, acting as the main interface through which users interact with the parking finder and navigation features. It is developed using the Flutter framework, allowing deployment on both Android and iOS platforms while ensuring a consistent and responsive user experience. The main specification of the mobile application is its integration with the Firebase Realtime Database, which allows it to receive instant updates on parking space availability and vehicle movements. As soon as the detection and tracking module updates the database, the changes are reflected in the app without delay, ensuring users always have access to the latest parking information.

From a functional perspective, the mobile application provides an overview of parking lots and their current status, including the number of available and occupied spaces. When a user selects a particular parking lot, the application displays a detailed layout of the lot, highlighting each parking spaces in real time with clear visual indicators where green for available and red for occupied. Vehicle icons are also shown to reflect active vehicle movements within the parking area.

Next, another main specification of the application is its pathfinding features. The parking layout is represented as a graph of nodes and connecting roads, and Dijkstra's shortest path algorithm is applied to determine the shortest path from the user's location to the nearest available space according to other vehicles status. This path is dynamically updated whenever parking space statuses change, ensuring that the user is always directed to the nearest and most possible parking space.

4.3 Components Design

4.3.1 Vehicle Detection and Tracking Design

The detection and tracking module are designed around a pipeline structure that begins with video input and ends with structured output ready for database updates. The video stream is continuously read frame by frame. Each frame undergoes pre-processing steps,

including resizing to a consistent resolution and conversion to RGB format. These steps standardize input and reduce computational overhead during inference.

The YOLOv8 object detection model is the core of this pipeline. The model was trained and fine-tuned to detect vehicles in parking lot, and its design supports high-speed inference while maintaining accuracy. During processing, YOLOv8 produces bounding boxes, class labels, and confidence scores for detected objects. To ensure reliability, the design includes thresholds for confidence and IoU, filtering out low-quality detections and resolving overlapping boxes.

The DeepSORT tracking module receives the results from the YOLOv8 model's vehicle detections. The DeepSORT combines appearance-based feature embeddings with motion prediction via a Kalman filter, allowing the system to consistently assign and maintain unique IDs for every vehicle, even when there is temporary occlusion or overlapping movement. Besides, each bounding box's bottom-centre point is computed and compared to predefined parking space polygons in order to connect detections with the actual parking lot layout. These polygons are defined during the system setup where they are representing the boundaries of individual parking spaces. By checking whether a vehicle’s bottom-centre point lies within these polygons, the system can determine if the vehicle is entering, currently parked, or exiting a space.

The design of the system also incorporates a state-labelling scheme that assigns status indicators to vehicles according to their position and movements. Vehicles that are first entered into parking lot are assigned with the status "finding”. If the bottom-centre point is stably for a period of time within a parking space polygon, then “parked” status is assigned. If a parked vehicle exits from a parking space polygon, then its status is changed to "exiting". These assignments of status are not only important for maintaining database consistency, they are also a useful context for the mobile app, such that mobile app system able to calculate the shortest path based on these statuses for users.

For debugging and verification, the system overlaying IDs, bounding boxes, and status indicators onto the live feed. The visual response allows easier verification on vehicles are being properly tracked and correctly identified, thus guaranteeing that the whole tracking and detection pipeline is working correctly.

4.3.2 Database Design

Firebase Realtime Database is constructed to be the backend and frontend's data synchronization backbone. It is stored in structured JSON, and two top-level nodes are maintained which include each parking lot node and summary node. In each parking lot node, they include parking_data and vehicle_data, parking_data is the node that contains details of all parking spaces. Each space is act as a child node with id, status, and timestamp attributes. The status attribute is binary, either the space is "occupied" or "available" and the timestamp allows for consistency in case of updating conflict.

The vehicle_data node holds real-time information for all vehicle being tracked within the parking lot. The record holds the ID that is unique, vehicle coordinates at the moment, and its status (finding, parked, exiting). The vehicle_data node enables the system to not only maintain vehicle locations but to also deliver context data that could be utilized in the navigation logic of the mobile application.

To enable smooth operation, real-time event listeners are utilized in database design. These notify the mobile application in real time whenever modifications are being performed on the data. Push-based communication therefore eliminates continuous polling at every point, ensuring that users are always served the current parking updates. Security rules are also defined to allow read and write access during development times to enable the backend to change data while it is being read by the application without interference.

4.3.3 Mobile Application Design

The mobile application is designed as the user interaction layer, where it is modular and layered architecture type for maintainability and clarity. The main.dart file served as an entry point of the application and it sets up Firebase according to the configuration that is defined in firebase_options.dart. This ensures that the application is synchronized with the database prior to displaying any of the application's UI screens. Initially, it first shows a splash screen to provide users with some feedback during initialization. Once the app is opened, it takes the user to the home page screen that list down every parking lot availability overview. The home page draws aggregated data from Firebase and shows the number of parking space that are available for every lot. Users able to choose

their parking lot of preference, and it will redirect user to the lot layout screen that matches.

Next, layout screens for each parking lot aim to combine real-time parking information with pre-defined graph-based parking maps. The layouts comprise nodes and edges that represent roads and parking space. The structure of the nodes is defined in the file node.dart and includes connections to adjacent nodes. The structure is a directed graph type, in which edges represent allowable driving routes between nodes. Pathfinding is done with Dijkstra-based algorithm. Once the user enters the parking lot, the app requests Firebase for the most up to date available parking spaces. The algorithm will find the shortest path from the user's point to the closest available space take consideration on other vehicle's status. The output is marked directly onto the map to display the suggested path. If circumstances are changed such as earlier available space being taken, the app works out the path dynamically again to ensure that the user is at all times pointed towards the nearest available space.

To graphically represent this information, the app implements custom widgets for rendering. Parking spaces are rendered as rectangles, green to indicate availability and red to indicate occupied. Roads and intersections are represented as connected sections, and vehicles as moving markers. The UI is self-updated at all times whenever Firebase is updated with fresh data, making it real-time and responsive.

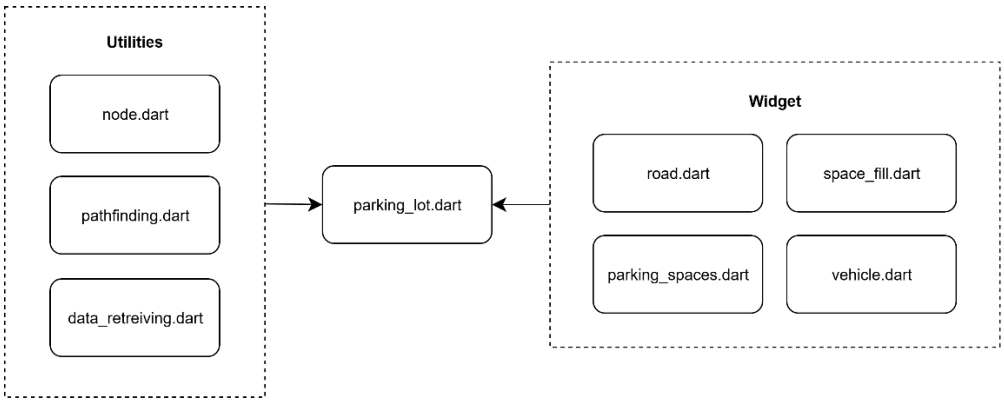

Figure 4.2: Component Design for Mobile Application.

4.4 System Components Interaction Operations

The operation of this project is not only based upon the individual operation of its components, but it is also based upon the interaction of components and data exchange.

Although all of the modules are individually assigned tasks, interactions form repeated and continuous loops extending from the beginning of video input to tracking, from synchronization of data to the Firebase Realtime Database, to the mobile application for use by users.

The process begins with the detection and tracking module, which processes incoming video streams from the parking lot cameras. Each frame is analysed by the YOLOv8 detection model to detect vehicles, and these detections are then passed to the DeepSORT tracker to assign consistent IDs across frames. The tracker also determines vehicle statuses such as “finding,” “parked,” or “exiting,” based on their movement and position relative to defined parking space polygons. At this stage, visual overlays such as bounding boxes and ID labels are added for monitoring purposes, but more importantly, structured data is generated for further use. This structured data includes vehicle identifiers, coordinates, and space occupancy, all of which are essential for synchronization with the backend.

Once results of tracking and detection for a frame are settled, the system sets up for updating of the database. The parking space status is kept under the parking_data node while vehicle information like ID, coordinates, and status are kept under the vehicle_data node inside a parking lot node in Firebase, and. These updates are sent to the database asynchronously to ensure that latest results overwrite old data without interrupting running processes. The database itself serves as the communication link between backend and frontend, provide a single repository where the latest status of the vehicle and spaces can be obtained. The real-time aspects of Firebase allow the mobile application to obtain these updates almost in real-time, without manual refreshing or constant polling.

On the mobile side, the Flutter application is designed to respond immediately to database changes. As soon as new data arrives, Firebase listeners trigger state updates within the app, prompting the user interface to refresh the map view. Parking spaces are updated to reflect to the most recent availability, with green indicating available and red showing occupied. At the same time, vehicle positions and movement statuses are rendered on the map, giving users a live representation of the parking lot. This creates an interactive, real-time link between the backend detection pipeline and the frontend user interface, allowing users to monitor parking space availability and vehicle flow seamlessly.

Implementation of the pathfinding system into this cycle is important to the system's effectiveness. The application employs the use of the defined node and edge structures from the graph-based representation of the map to determine navigation routes. Once a new user enters the lot, the application detects available parking spaces from the parking_data node and chooses the nearest available space based on graph distance. Dijkstra's Algorithm is employed to work out the shortest path from the user's point to the destination parking space. The output of this operation is a highlighted path on the map, directing the user step by step through the parking lot. In case a formerly available parking space is taken by another vehicle before the user reaches it, Firebase instantly pushes out the change to the app, prompting the pathfinding system to recalculate and reroute the user to the next nearest available space. This continuous recalculation ensures that navigation is accurate and reliable even for very dynamic parking lots.

Overall component interaction is designed to be continuous and recurring. Cameras continuously feed video input, the tracking and detection module generate structured data, the database assigns this data, and the mobile application utilize it for visualization and navigation. Therefore, the vehicle movements within the parking lot will determine the state of the system because occupied or available spaces trigger the new detections that roll again. This tightly coupled loop creates a closed feedback system in which all activities in the actual parking lot are resemble electronically within the application.

By structuring interactions in this way, this project system can ensure that its components never operate in isolation but rather operate to build towards a logical workflow. The design guarantees that vehicle detections feed seamlessly into updates of parking availability, that updates are pushed to the mobile application in quick succession, and that users always receive accurate navigation recommendations. The interaction operations thus become the foundation of the system, thus transforming raw video inputs to useful, real-time assistance for drivers at parking locations.

CHAPTER 5

System Implementation

This chapter explains how the system was implemented, including how the hardware and software were set up, configured, and integrated. Additionally, it provides screenshots of the system in use, discussion about implementation difficulties, and concludes with the overall results.

5.1 Hardware Setup

The processor unit and the mobile device used for application testing make up the two primary components of the hardware setup for this project. The system's processing unit handled video processing, communication with Firebase, and the implementation of the detection and tracking algorithms. In the meantime, the mobile device mainly served as the end-user platform, enabling real-time parking space availability monitoring with pathfinding for user and confirming that the Flutter application accurately presented data from the backend.

5.1.1 Processing Unit (PC)

Table 5.1.1: Specifications of desktop computer

Description	Specifications
Mobo	MSI B650M Gaming Plus WIFI
Processor	AMD Ryzen 5 7500F
Operating System	Windows 11 Pro
Graphic	NVIDIA GeForce RTX 4060 Ti 8GB GDDR6
Memory	16GB DDR5 RAM
Storage	1TB SSD

A desktop computer is served as the primary processing platform for the development and operation of the parking space and vehicle detection and tracking modules. While the CPU and memory resources enabled video processing, Firebase connectivity, and result logging, the GPU played a crucial role in speeding up the YOLOv8 model for model training, model fine-tuning and enabling real-time detection and tracking. By

sending detection outputs to Firebase and synchronizing data with the mobile application, the desktop not only ran the models but also manage the integration of many components. During testing, this hardware configuration made sure that everything worked easily and dependably, from video input to real-time monitoring.

5.1.2 Android Mobile Device

Table 5.1.2: Specifications of android mobile device

Description	Specifications
Model	Vivo V25 5G
Processor	Mediatek Dimensity 900 (6nm)
Operating System	Android 14
Graphic	Mali-G68 MC4
Memory	16GB RAM (8GB + 8GB Extended)
Storage	256GB

During the deployment phase, an android mobile device was widely utilized to facilitate the mobile application's testing and validation. It served as the main operating base for the Smart Parking Finder Flutter application, which offered the user interface for real-time pathfinding, vehicle tracking results and parking space status monitoring. The Firebase database updates were continuously synchronized and seen via the mobile device, enabling confirmation of the correct transmission and presentation of detection outputs from the backend.

During the testing stage, the smartphone was also utilized to record videos and capture images for parking lot data. By giving the detection and tracking modules accurate input, these video recordings and images made it possible to test the system in real-world parking lot scenarios. Vehicle detecting and tracking, data exchange, and end-user real-time visualization were all part of the end-to-end workflow that was validated by using a mobile device for testing and data gathering.

5.2 Software Setup

The detection and tracking framework, database service, and mobile application environment must be installed and configured as part of the software setup process. The way these parts were put together enables them to function as a single, cohesive system. The backend environment was configured to track and detect vehicles, handle video data, and determine the availability of parking spaces. The mobile application environment was ready to retrieve the data and provide it to users via an easy-to-use interface, while the database service was set up to store and update this information in real time.

5.2.1 YOLOv8 for Vehicle Detection

Ultralytics' YOLOv8 (You Only Look Once, version 8) detection framework was chosen for this project due to its ability to effectively balance processing speed and accuracy. To ensure that the detection was adjusted for real-world circumstances, a small-sized model, YOLOv8s was trained using a custom dataset that included annotated photos of vehicles gathered from online datasets and the target parking location. The model was deployed using the Ultralytics Python module after training and fine-tuning, enabling real-time inference and acting as the system's basis for vehicle detection.

5.2.2 DeepSORT for Vehicle Tracking

DeepSORT was combined with YOLOv8 to guarantee that vehicles were given consistent IDs across frames and overlapping camera views. Through a pre-trained ReID (re-identification) model, the tracker integrated appearance-based matching with motion prediction using a Kalman filter. This made it possible for the system to consistently preserve vehicle IDs over time, even in situations when cars were momentarily obscured or in close proximity.

5.2.3 Python Environment and Libraries

The pipeline for tracking and detection was using Python 3.13. Ultralytics for YOLOv8-based car detection, deep-sort-realtime for tracking multiple objects, opencv-

python for processing video frames and detecting parking spaces, and firebase-admin for interacting with the Firebase Realtime Database were among the essential libraries. The implementation of real-time detection, tracking, and data logging was made possible by these fundamental elements. The complete list of dependencies and their version details is documented in the requirements.txt file, which is included in Appendix A for reference.

5.2.4 Firebase Realtime Database

The Firebase Realtime Database was set up to handle vehicles activity logs and parking spaces availability. Vehicles and spaces were the two main branches of the database structure. While the space branch kept track of each parking space's occupancy status, the vehicle branch stored information including IDs, status, coordinates xy and timestamps. The configuration file google-services.json was incorporated into the Flutter application to facilitate smooth synchronization with the backend, and database administration and monitoring were done via the Firebase Console.

5.2.5 Flutter Mobile Application

The Flutter SDK (version 3.29) in Android Studio was used to create the mobile application, and Firebase was implemented using firebase_core and firebase_database. Real-time parking spaces availability, vehicles positions, and navigation routes are all displayed on the application's user-friendly interface. By integrating these functions, the software improves monitoring, direction, and general efficiency in the parking environment in addition to helping users find available spots.

5.2.6 Development Tools

Visual Studio Code, which offered an environment for writing and debugging Python scripts, was used to construct the system's backend. The Flutter mobile application, including emulator simulations, was developed and tested using Android Studio. Throughout the project, effective code management and collaboration were made possible by the usage of GitHub for version control.

5.3 Setting and Configuration

5.3.1 Backend Configuration

In order to develop the Smart Parking Finder Application, the backend configuration was created to incorporate real-time database synchronization, vehicle detection, and tracking. Starting with the environment setup, Python 3.13 was utilized together with necessary libraries including DeepSORT, OpenCV, NumPy, and Ultralytics YOLO. Additionally, the Firebase Admin SDK was set up to facilitate connection with cloud databases.

The detection component made use of a specially trained YOLOv8 model that was fine-tuned for vehicle detection in order to increase accuracy in the intended setting. The inference parameters were set at an IoU threshold of 0.3 and a confidence threshold of 0.6, and the model weights were fed straight into the system. When detecting vehicles, these criteria made sure that recall and precision were balanced.

The DeepSORT tracking was included to keep track of the vehicle's identity consistency between frames. With a maximum age limit of 30, this tracker combined motion and appearance elements to allow for brief occlusions without erasing track IDs. Every vehicle that was detected was given a unique identification number and continuously monitored throughout the video frames.

A technique for manually generating coordinates was used to predefine parking spaces. Each space was represented by four points that admin user could indicate on an image frame using the program. The coordinates were then saved in a text file called parking_coords.txt. In order to determine if a slot was occupied or available in real time, the backend loaded these coordinates during runtime and used a polygon test to match vehicle positions against them.

A local video file was used as the system's primary input during testing, with each frame resized to a fixed resolution for consistent processing. The YOLOv8 detector first identified vehicles, which were then passed to the DeepSORT tracker. The tracker output included bounding boxes, unique IDs, and the bottom-centre point of each vehicle, which was cross-referenced with the parking slot coordinates.

The processed data was then structured into three categories which are individual space availability, vehicle details, and summary statistics. To optimize performance, Firebase database updates were performed asynchronously using Python's threading library. This ensured that detection and tracking continued uninterrupted while maintaining real-time synchronization of data, including the number of totals, occupied, and available spaces.

Finally, a visualization layer was implemented for verification. Bounding boxes, vehicle IDs, and space boundaries were overlaid on the video feed, with colour coding (green for available and red for occupied) to provide immediate feedback during operation. This configuration allowed the backend to operate efficiently, ensuring accurate detection, robust tracking, and reliable data management.

5.3.2 Database Configuration

The backend of the system was integrated with Firebase Realtime Database to manage parking slot status, vehicle logs, and overall lot summaries in real time. The connection was established using the Firebase Admin SDK, which required the export of a service account JSON key from the Firebase Console. This file was attached to the backend for authentication and secure access, allowing the Python program to perform read and write operations during execution. Similarly, a google-services.json file was exported and integrated into the Flutter project to enable synchronization between the mobile application and the database.

The database was organized under the root node of each parking lot and summary node, where each parking lot node consisted of two primary branches which is parking_data and vehicle_data. The first branch parking_data, stored information for each individual parking space, including its unique identifier, availability status, and a timestamp of the last update. The second branch vehicle_data, logged the tracking results from the DeepSORT algorithm, with each entry indexed by a unique vehicle ID. These records contained the bounding box coordinates of vehicles, their current status (finding, parked, or exiting), and the corresponding timestamp. Another root node which is summary, provided collection statistics for each parking lot such as the total number of spaces, the numbers of occupied slots, available slots, and the last update

time. The data format in the Firebase for each parking lot node and summary node is shown in Figure 5.1 and Figure 5.2.

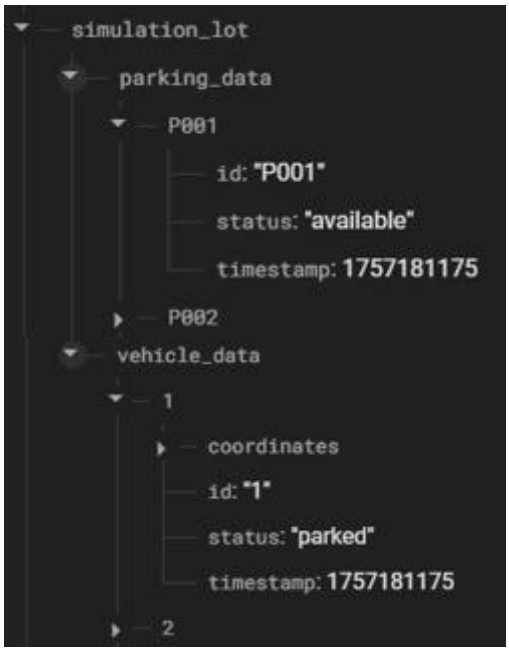

Figure 5.1: Firebase data format for each parking lot

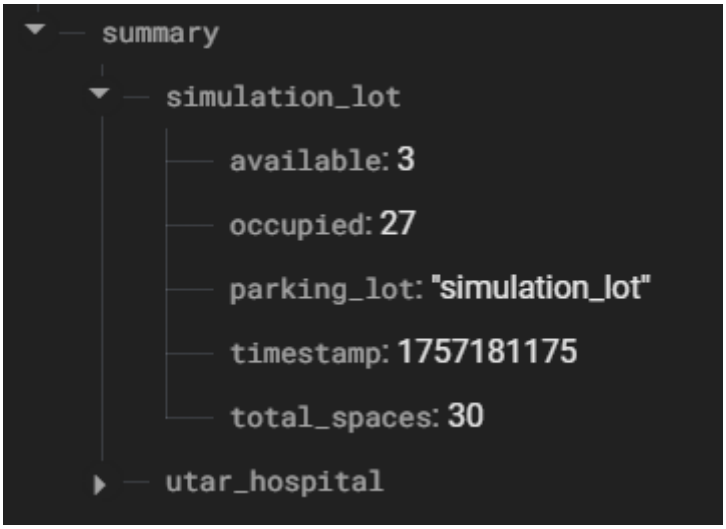

Figure 5.2: Firebase data format for summary node

This hierarchical structure was chosen to ensure a clear separation between space-level details, vehicle-level logs, and high-level summaries, enabling efficient retrieval and updates from both backend and frontend. To maintain real-time responsiveness, updates to Firebase were pushed every one second on a background thread, ensuring continuous synchronization without disrupting detection and tracking. The database rules were configured such that both read and write permissions were set

to true, allowing the backend system to update data and the Flutter mobile application to retrieve information without restrictions.

5.3.3 Flutter App Configuration

The Flutter mobile application was created using a modular architecture to achieve scalability, maintainability, and a clear separation of responsibilities. Firstly, the main.dart file serves as the entry point of the Flutter application. It begins by initializing Firebase using the credentials specified in the firebase_options.dart file, ensuring seamless synchronization with the Firebase Realtime Database. Once initialization is complete, the application launches into the splash_screen.dart, which provides a brief loading interface before redirecting the user to the homepage.

The screens directory is the central location where all application interfaces are organized. The homepage.dart acts as the entry point for users, displaying a list of available parking lots together with the corresponding parking availability data retrieved from Firebase. From this page, users can select their desired parking lot. Each subsequent screen, such as hospital_layout.dart, parking_layout.dart, and simulation_layout.dart, is designed to render the parking lot map specific to its environment. These layouts integrate components from other directories, such as utilities and widgets, to build the interactive map, plot parking spaces, and provide real-time navigation features. Additionally, the splash_screen.dart is used to display the initialization interface during app startup.

The utilities directory contains a collection of utility classes that the application uses to support computational and data processing operations. Real-time listeners are managed via the data_retrieval.dart file, which retrieves vehicles and parking spaces changes straight from Firebase. The node.dart file defines the structure of nodes representing roads and parking spaces. It connects these nodes into a directed graph, enabling pathfinding features within the parking map. By implementing Dijkstra's Algorithm, the pathfinding.dart file enables the system to determine the quickest route between the user's present location and a parking space that is available.

Custom widgets from the widget directory are used to create the parking lot's graphical representation. Each parking space is represented graphically via the

parking_spaces.dart widget, which is updated dynamically based on the database's availability status. Road connections within the lot are rendered by the road.dart file, while gaps between spaces and roads are filled by space_fill.dart to guarantee correct alignment of unoccupied regions. Using coordinates retrieved from the backend, the vehicle.dart widget is in charge of showing and animating vehicles on the layout.

For these to work as one, first the parking map is defined and connected as a graph structure, where nodes that represent parking spaces, road, and space fills, while edges define the valid paths between them such as one way road. Once the graph is established, real-time data retrieved from Firebase such as parking space statuses and vehicle locations are integrated into the map. This data ensures the system reflects the current parking lot conditions. The pathfinding algorithm, based on this graph, calculates the shortest available route from the user's position to the nearest available spaces, taking into account of vehicles status in front of user. The resulting path is then highlighted on the user interface, visually guiding the user along the road network. Every time new updates are received from Firebase, the pathfinding process is recalculated, ensuring that users are always directed to the most optimal available parking space with accurate, real-time navigation.

5.4 System Operation

5.4.1 Detection and tracking

When the system runs, video frames are read continuously and pre-processed before inference. Frames are first resized for consistent processing and converted to the required colour space for the model. The fine-tuned pre-trained YOLOv8 model performs vehicle detection on each frame using the configured inference thresholds, and detected vehicle bounding boxes are returned together with confidence scores. These bounding boxes are then passed to the DeepSORT tracker, which assigns and maintains unique tracking IDs across subsequent frames. During tracking, each vehicle is also assigned a real-time status such as "finding" when it is first detected and still moving through the lot, "parked" when it occupies a predefined parking space, and "exiting" when leaving a previously occupied parking space. For visual verification, bounding boxes are drawn onto the frame along with the tracker ID, status label, and a

small marker at the bottom-centre of each box. These overlays make it straightforward to observe whether a vehicle is being consistently tracked over time and to confirm its current parking status. Figure 5.3 shows a sample detection frame with bounding boxes, assigned IDs, statuses, and bottom-centre markers.

Figure 5.3: Detection and tracking output

5.4.2 Parking space plotting and occupancy determination

Once detections are available and vehicles are tagged with track IDs, the system determines parking space occupancy by testing each vehicle's bottom-centre point against predefined parking slot polygons. Parking slots are manually mapped (four-point polygons per slot) and stored in a coordinate file which the backend loads at runtime. After the plotting, each parking space is drawn as a polygon using the coordinate loads from the file on the video frame. The spaces are color-coded where green if available and red if occupied based on the results of the polygon test that checks whether a small marker at the bottom-centre of each vehicle box falls inside the space, shown in Figure 5.6. Parking space IDs are also displayed next to each polygon for easier verification. This visual overlay clearly shows which vehicle belongs to which space and is done right before updating the database with the occupancy results.

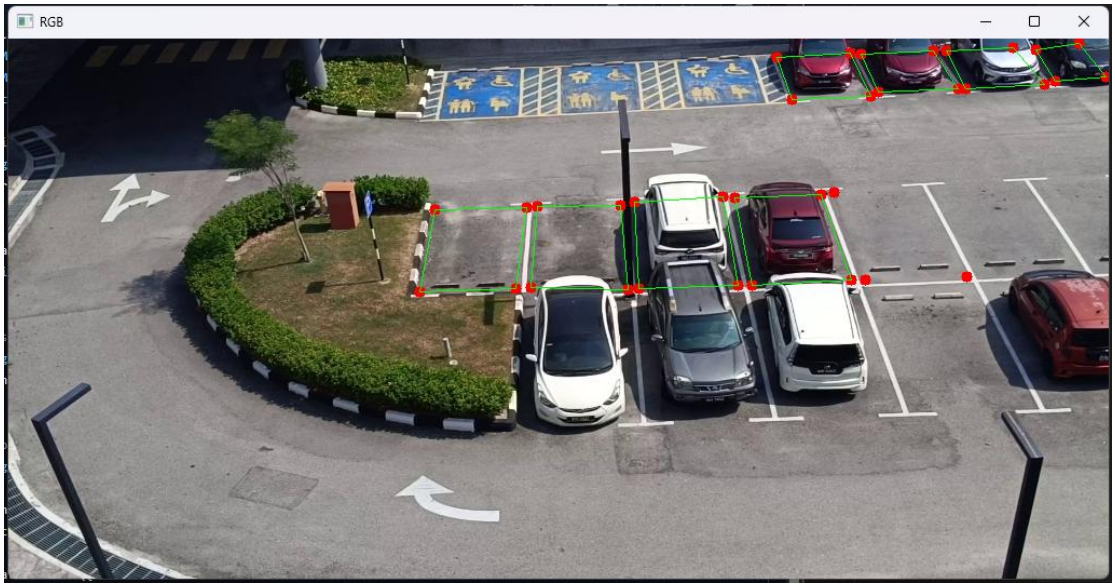

Figure 5.4: Parking space plotting

YOLO > python > parking_coords.txt

```
1   [(713,19), (727,56), (800,52), (779,15)]
2   [(792,15), (807,50), (877,46), (856,14)]
3   [(873,14), (889,45), (954,40), (935,10)]
4   [(952,9), (969,39), (1016,35), (1000,5)]
5   [(398,163), (384,229), (466,228), (474,159)]
6   [(495,160), (492,226), (570,224), (566,158)]
7   [(586,156), (588,222), (673,223), (659,151)]
8   [(675,149), (687,225), (779,221), (755,146)]
9   [(765,145), (794,218), (886,217), (845,141)]
10  [(861,140), (901,218), (991,214), (942,136)]
11  [(959,137), (1009,211), (1018,211), (1016,133)]
12  [(484,249), (488,355), (580,352), (572,246)]
13  [(588,245), (596,347), (701,346), (680,241)]
14  [(690,240), (716,343), (819,342), (788,241)]
15  [(801,240), (839,339), (947,338), (894,230)]
16  [(909,230), (965,335), (1016,334), (1003,229)]
17
```

Figure 5.5: Parking space coordinate file

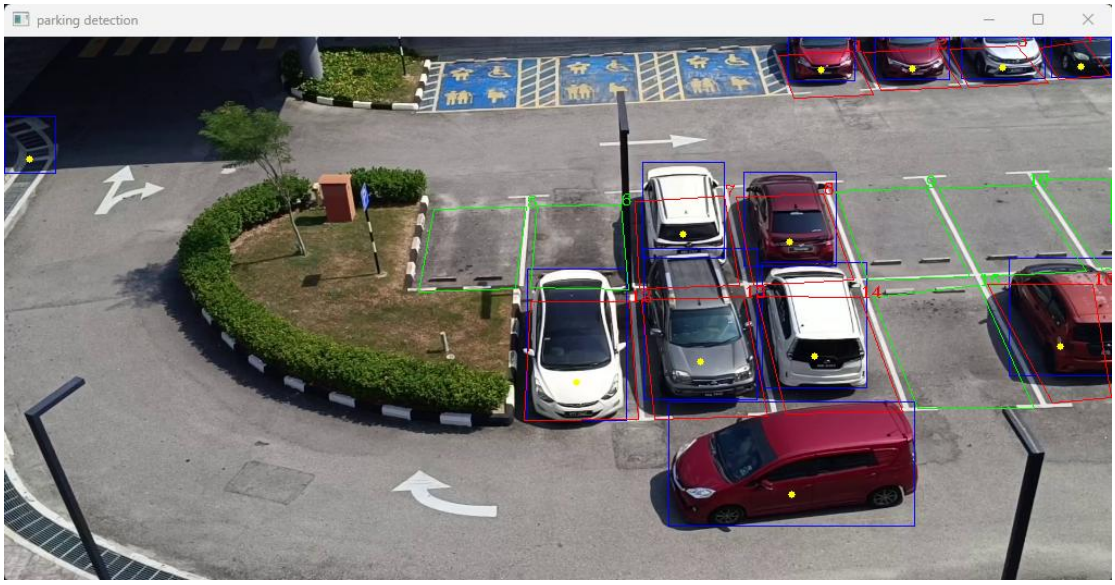

Figure 5.6: Parking space occupancy determination

5.4.3 Firebase update and data packaging

After occupancy and tracking state are determined per frame, the backend combined the information into structured records for cloud synchronization. For each parking space the system composes a parking record (id, status, timestamp), and for each confirmed track it produces a vehicle record (id, coordinates, status, timestamp). A small summary object containing total spaces, occupied count and available count is also constructed. To avoid impacting real-time inference, these updates are dispatched on a background thread at a regular interval (which is every one second in this project), so detection and tracking continue without blocking. The database schema is written to the Realtime Database under the configured root, populating parking_data, vehicle_data and summary branches so the frontend can efficiently retrieve slot and vehicle information.

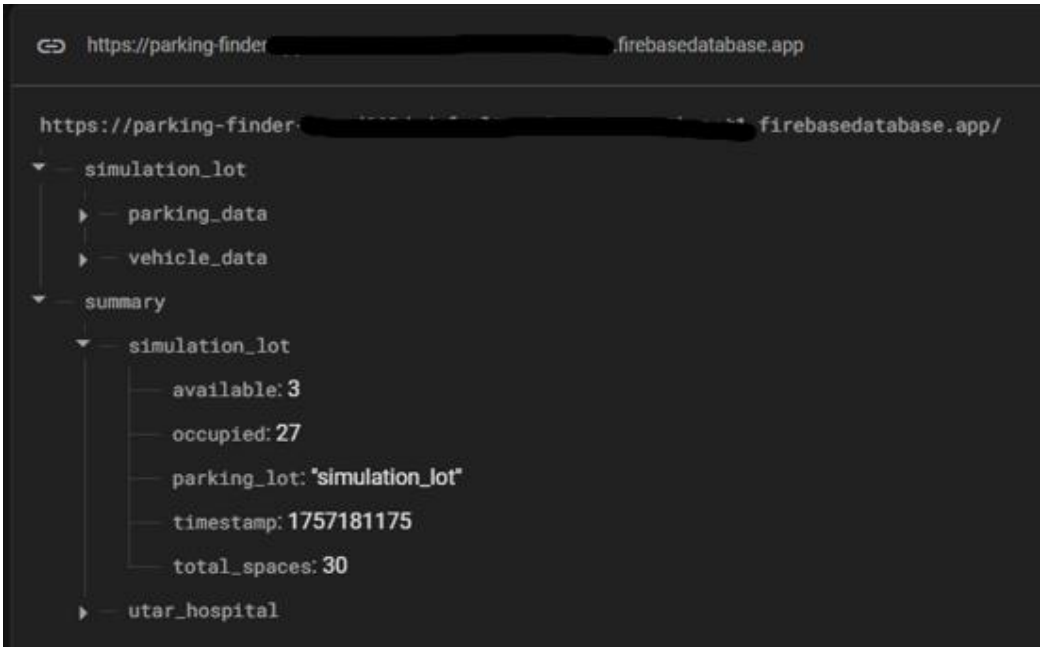

Figure 5.7: Firebase Realtime Database entries produced by the backend

5.4.4 Mobile app visualization and navigation

The mobile application listens for changes on the corresponding Firebase nodes and reacts in real time. When parking_data or vehicle_data is updated, the data_retrieval utility receives the change and updates the app state. The app renders the parking lot as a graph-based map using custom widgets where parking spaces are drawn and colour coded, roads and empty areas are rendered, and vehicles are animated at their reported coordinates. The pathfinding utility recalculates routes using the current parking graph and Dijkstra's algorithm whenever the user's position or parking spaces availability changes, edge weights reflect travel cost and can be adjusted to account for other vehicles status in front of user. The app interface also provides live animation of the recommended route so users see the path to the target parking space and any vehicles ahead.

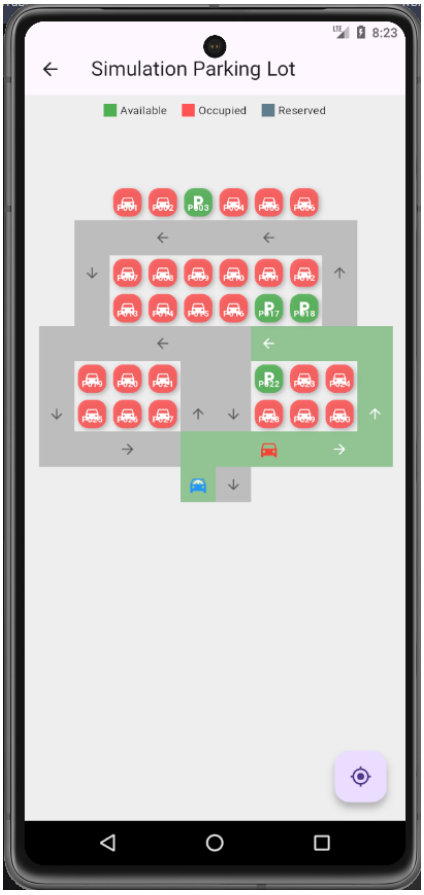

Figure 5.8: Mobile app visualizing space availability, vehicle positions and calculated navigation route

5.5 Implementation Issues and Challenges

Throughout the development of this project, several implementation challenges were encountered that directly impacted both the accuracy and efficiency of the solution. One of the most significant issues was related to GPU memory usage during the training and fine-tuning of the YOLOv8 detection model. While higher-capacity variants of the model, such as YOLOv8l or YOLOv8x, could potentially provide superior accuracy in detecting vehicles under diverse conditions, the limited GPU resources available restricted the use of these larger models. Attempting to train or fine-tune them often resulted in out-of-memory errors, forcing the system to rely on a smaller version of YOLOv8. This choice provided a practical trade-off between performance and hardware limitations, though it introduced a compromise in detection precision that may affect scalability in larger deployments.

Another challenge is from the problem of occlusion and overlapping vehicles. In real-world scenarios, vehicles often park closely side by side or temporarily obstruct one another when entering or leaving parking space. While the integration of the DeepSORT tracking algorithm helped address short-term occlusions by combining appearance and motion cues, prolonged or severe occlusions still caused occasional ID switches. These ID mismatches sometimes led to false occupancy updates, particularly when two vehicles overlapped near the same parking space. Additional measures, such as more advanced ReID models, may be necessary in future iterations to further enhance tracking stability.

Moreover, due to restrictions in accessing actual parking lots for data collection, real-world datasets of vehicles entering and exiting parking lots could not be obtained for this project. Instead, simulated data was generated and uploaded to Firebase to recreate parking lot scenarios, allowing the mobile application to function as though it were connected to a live environment. The simulated data included both parking_data and vehicle_data, which were used by the mobile application to perform pathfinding and navigation functions. This setup enabled controlled testing of detection, tracking, and navigation features within the app. However, it may not fully capture the unpredictability and diversity of real-world conditions such as varying lighting, weather, or heavy traffic flow. As such, the results obtained serve primarily as a proof-of-concept rather than a definitive evaluation of the system's performance in live deployment.

5.6 Concluding Remarks

The implementation of this project successfully demonstrated the integration of computer vision, real-time database synchronization, and mobile application visualization into a cohesive end-to-end solution. Despite challenges such as GPU resource limitations, vehicle occlusion, and reliance on simulated datasets, the system proved capable of detecting and tracking vehicles, determining parking space occupancy, and guiding users to available parking space through a responsive Flutter application. The project validates the feasibility of using YOLOv8 and DeepSORT in a smart parking context and highlights the potential of cloud-backed mobile applications for real-time user assistance.

CHAPTER 6

System Evaluation and Discussion

6.1 System Testing and Performance Metrics

To evaluate the performance of this project, a series of tests were conducted focusing on four primary metrics which includes detection accuracy, tracking stability, database update time, and mobile application latency. These metrics were chosen to measure the accuracy, reliability, and responsiveness of the system across its major components, ensuring both backend and frontend performance could be objectively assessed.

6.1.1 Detection Accuracy (mAP)

The accuracy of vehicle detection was measured using the mean Average Precision (mAP), which evaluates the precision-recall performance of the YOLOv8 model at various confidence thresholds. Two different training setups were tested. First, the YOLOv8s model was trained simply on an online vehicle dataset, providing a baseline for general detection capability across varied environments on detecting vehicles. Next, this model was further fine-tuned using custom images collected from the target parking lot, allowing adaptation to environment-specific characteristics such as camera angle, lighting, and parking space geometry.

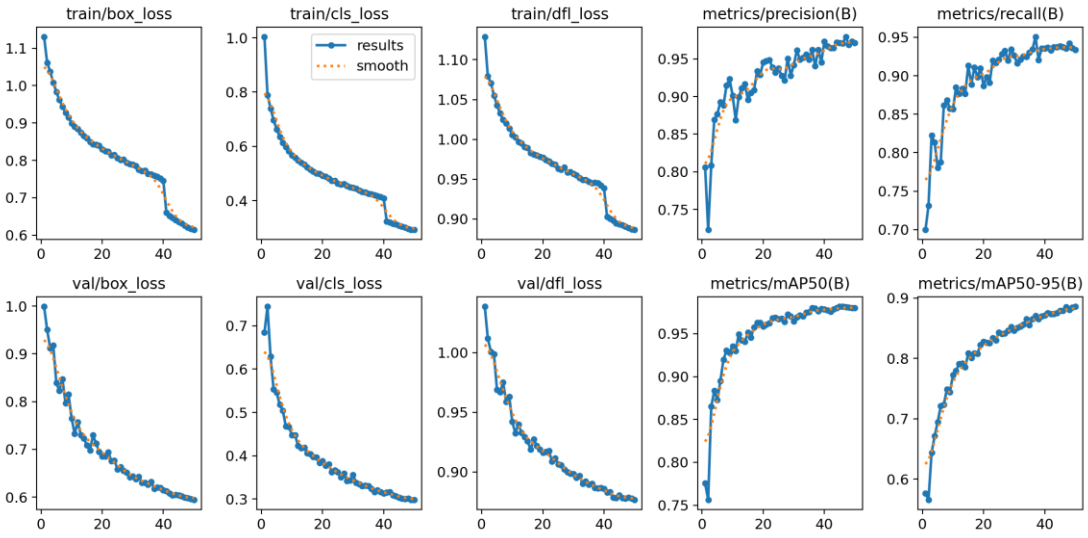

Figure 6.1: Model training result using online dataset

6.1.2 Tracking Stability (ID Switches)

The consistency of the DeepSORT tracker was evaluated by monitoring the number of ID switches that occurred during vehicle movements. An ID switch is recorded when the same vehicle is mistakenly assigned a new identifier during continuous tracking. While short-term occlusions and side-by-side vehicles were generally handled well, longer occlusions or overlapping vehicles sometimes caused ID mismatches. These switches could occasionally propagate to false parking space status updates.

6.1.3 Database Update Time

The responsiveness of the backend-to-database communication was assessed by measuring the time taken to package detection and tracking results and push them to Firebase Realtime Database. On average, updates were performed asynchronously at an interval of one second, which balanced real-time responsiveness with stable system performance. This ensured that parking space availability and vehicle data remained up to date without introducing significant computational overhead.

6.1.4 Mobile Application Pathfinding

The performance of the mobile application was examined by testing the accuracy and reliability of its pathfinding feature. The system was compared to its ability to calculate and display the shortest path from user's point to the nearest available parking space under different simulated parking lot conditions. Using the graph-based representation of the lot, Dijkstra's algorithm always produced optimum paths that occupied spaces. One of the main observations during testing is that the algorithm continuously updated navigation to reflect the shortest available route even as space availability changed with vehicles entering or exiting. With each update of the database, the application recalculated routes in real time and marked the new route on the parking map. The findings reconfirmed that the navigation module not only preserved route validity but also guaranteed that users were always routed through the most optimal path to the destination, making the system effective for real-time parking directions.

6.2 Testing Setup and Result

6.2.1 Test Environment

The system was tested with real videos shot using the mobile phone's camera maintained at the entrance and within the UTAR Hospital parking lot. The videos were shot at 1920×1080-pixel resolution and 30 frames per second, which represents the typical field of view of a fixed surveillance camera. The captured videos were then provided as input to the backend tracking and detection system.

The backend was executed on a desktop computer with the specification from chapter 5 (Table 5.1.1). YOLOv8 fine-tuned using parking lot images specific to the custom environment was used for vehicle detection, while object tracking was carried out using DeepSORT. Detection and tracking output were pushed to Firebase Realtime Database in real time for synchronization with the mobile app.

The mobile app was then tested using an Android phone running on specification from chapter 5 (Table 5.1.2). The app connected to real-time updates of Firebase and presented both the status of parking space and navigation directions. Pathfinding effectiveness was then tested through testing different parking lot situations using the simulated data to ensure that the app always calculated and displayed the shortest path accordingly to available spaces.

6.2.2 Test Procedure

For testing the system in a simulated environment, a 30-slot simulated parking lot facility was created. Before running each simulation, 25 slots were randomly pre-assigned as occupied and 5 slots were kept available for incoming cars. The test process was designed in a way to test entry and exit parking space scenarios. For vehicle entry, five vehicles were created at the given entry node and automatically directed to the nearest available space using Dijkstra's shortest path algorithm. Their positions were updated continuously until they arrived and took up their given parking positions. In the meantime, vehicle exit was simulated by randomly selecting three cars from the parked group and directing them out to given exit nodes. These vehicles experienced a "parking" to an "exit" state and were removed from simulation as they hit the boundary of their path. Simulation loop executed in real time with one-second tick interval, in which vehicle displacements were updated, parking space statuses recalculated, and all

updates synchronized to Firebase Realtime Database. This ensured the mobile application is always showing the latest parking availability and movement of vehicles. The process was performed numerous times to ensure detection stability, correct state transition correctness, correctness of available parking space updates, and pathfinding result reliability. Tests confirmed whether users were always routed to the nearest available space and whether exiting cars correctly left their designated spaces in the database.

6.2.3 Test Results

The system was subjected to independent test scenarios to evaluate its detection, tracking, database synchronization, and mobile app performance. Overall, the findings justified that the designed approach achieved high accuracy and robust performance under different parking conditions.

Detection was done using the fine-tuned YOLOv8 model with an overall mean Average Precision of 61.2%, though fairly modest by other standards, it was adequate for good vehicle detection in real-world conditions with fluctuating light and occlusion. Tracking performance using the DeepSORT algorithm was consistent with minimal switches per run. Though longer occlusions at times led to mismatches, overall vehicle ID continuity between frames was good enough to provide correct parking space status transitions.

```
Ultralytics 8.3.75 🚀 Python-3.13.1 torch-2.6.0+cu118 CUDA:0 (NVIDIA GeForce RTX 4060 Ti, 8188MiB)
Model summary (fused): 168 layers, 11,127,132 parameters, 0 gradients, 28.4 GFLOPs
val: Scanning D:\UTAR\Degree\FYP\YOLO\car_dataset\valid\labels.cache... 500 images, 0 backgro
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100
                   all        500       4845      0.254      0.764      0.403      0.316
                   bus         94         96    0.00201      0.969    0.00643    0.00413
                   car        500       4103      0.587      0.635      0.612      0.445
                 truck        135        188     0.0587      0.553      0.175       0.14
                   van        212        458      0.369        0.9      0.817      0.676
Speed: 0.5ms preprocess, 5.1ms inference, 0.0ms loss, 1.1ms postprocess per image
Results saved to runs\detect\val8
```

Figure 6.2: Car detection results mean Average Precision

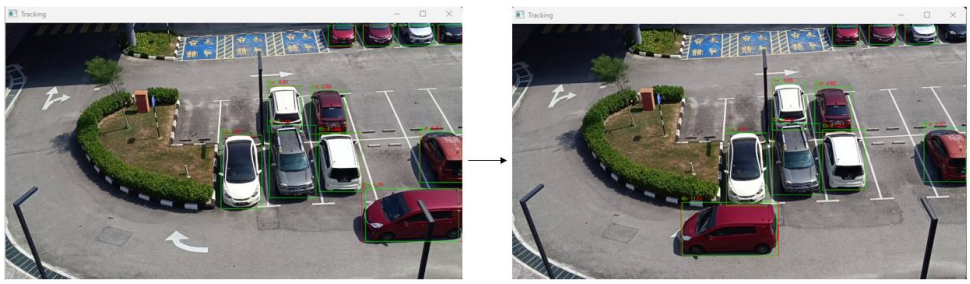

Figure 6.3: Stable ID assigned between few frames

Communication between the database also worked as required with updates arriving at a mean rate of one second. This was sufficient to provide changes in the availability of spaces and vehicles locations continuously on Firebase without being too laggy. The mobile application could effectively utilize these updates to dynamically recalculate pathfinding directions.

Pathfinding outputs also showed the robustness level of the system. In all test cases, the application achieved a 100% success rate of the shortest path from the user's point to a nearest available parking space according to other vehicle's status. Ideally, when vehicles departed and left newly opened spaces, the app dynamically recalculated and updated changed routes, always directing users through the best route.

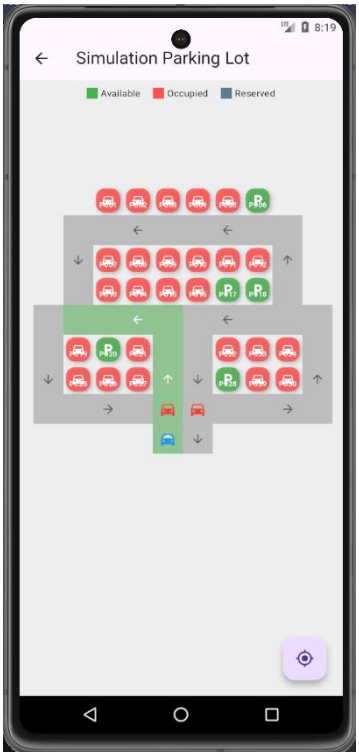

Figure 6.4: Shortest Path Result with Random Simulated Data Set

Figure 6.5: Shortest Path Result with other Random Simulated Data Set

The combined outcomes indicate that not only does the system identify and pinpoint vehicles with excellent precision but also learns synergistically with the backend and mobile application to provide real-time parking instructions.

6.3 Projects Challenges

First challenge faced during the project was related to real-time synchronization overhead. Since the system relies on multiple components which are detection, tracking, database updates, and mobile app visualization to operate simultaneously, maintaining smooth synchronization across all modules occasionally introduced latency. When detection and tracking workloads were heavy, for example in frames with multiple vehicles, the asynchronous updates to Firebase sometimes lagged behind, which slightly affected the freshness of the data displayed on the mobile application.

In addition, the next project challenge is unexpected user behaviour. Although the system is designed to guide users to the nearest vacant parking space using the shortest path, users may not always strictly follow the recommended route. Besides, some other vehicle may also not take the nearest empty parking space which will make the navigation route constantly changing for user as it brings user to nearest vacant

parking space. Such behaviours introduce uncertainties that can affect the accuracy of real-time navigation and occasionally lead to inconsistencies between the system's guidance and the actual user actions.

Finally, the mobile application itself presented performance constraints, particularly when rendering large layouts with many parking spaces and active vehicles. The Flutter framework, while flexible, required optimization to handle frequent updates and animations in real time. When multiple widgets such as vehicles, spaces, and navigation paths were updated simultaneously, occasional performance drops occurred in a larger lot, which reduced the smoothness of the user experience. These issues highlighted the need for further optimization and possibly more efficient rendering techniques in future versions of the application.

6.4 Objectives Evaluation

The first objective of this project was to develop a real-time parking space vacancy detection system using computer vision and deep learning. This objective was successfully achieved by integrating the YOLOv8 detection model with predefined parking space coordinates. The system was able to determine whether each space was occupied or available and update this information continuously in real time. By synchronizing the processed results with Firebase Realtime Database, users could access the latest parking availability through the mobile application, ensuring that accurate and reliable information was always presented.

The second objective was to implement a vehicle detection and tracking system with parking finding status. This goal was also accomplished by combining YOLOv8 with the DeepSORT tracking algorithm, which maintained vehicle identities across frames and provided insights into their status, such as whether a vehicle was searching for a space, parked, or exiting the space. Despite occasional challenges caused by occlusion or overlapping vehicles, the system demonstrated stable tracking performance and ensured that vehicle behaviour within the parking lot could be monitored effectively. This provided valuable real-time information on the movements of other vehicles for the end user.

The final objective was to design and develop a mobile application that delivers a smart navigation system to guide users to the nearest vacant parking space according to vehicle status in front. This was achieved using Flutter to create a cross-platform

application that integrates seamlessly with Firebase. The application not only displayed parking space availability and vehicle movements in real time but also implemented a graph-based navigation system using Dijkstra's Algorithm to calculate the shortest path to an available slot taking consideration of other vehicle's status along the path. The interface provided users with clear visualization, live updates, and dynamic route adjustments, ensuring efficient and user-friendly navigation throughout the parking lot.

6.5 Concluding Remarks

The evaluation and discussion of this project highlighted both the strengths and limitations of the proposed solution. Testing results showed that the system was able to achieve accurate vehicle detection, stable tracking performance, and timely synchronization with Firebase, ensuring that users could rely on real-time updates for parking availability. The mobile application also performed effectively in delivering navigation guidance and visualizing parking spaces statuses with minimal latency, demonstrating the feasibility of integrating backend intelligence with a user-friendly frontend interface.

At the same time, several challenges were observed during evaluation, such as GPU memory constraints that limited the use of higher-capacity detection models, occlusions that occasionally caused ID switches in tracking, and performance overheads in rendering complex layouts within the mobile app. Additionally, the unpredictability of user behaviour posed practical challenges for navigation accuracy. Despite these issues, the system met its core objectives and provided a strong proof of concept for a smart parking finder solution.

CHAPTER 7

Conclusion and Recommendation

7.1 Conclusion

This project pursues to develop a Smart Parking Finder Mobile Application using computer vision, real-time databases, and mobile navigation to address the daily problem of finding available parking in urban cities. The overall objectives were to develop a parking space detector system, create a vehicle tracking system with parking status, and offer a user-friendly mobile application with smart navigation features. All of these objectives were achieved, providing a proof of concept for how AI and mobile technologies can be used to improve parking efficiency.

On the backend, the system used YOLOv8 for real-time vehicle detection and DeepSORT for continuous tracking to detect and track vehicles across frames. Status tags such as finding, parked, and exiting were incorporated to provide reasonable context for database records and mobile application updates. The data was synchronized in real time with Firebase Realtime Database, which acted as the communication gap between the mobile application and backend. On the frontend, this information was retrieved in real time by the mobile app built with Flutter, presenting a dynamic parking map and providing smart navigation. Using a graph-based model of the parking lot and Dijkstra's algorithm, the app was constantly highlighting the shortest available path to an available parking space, adapting instantly to such events as vehicle entering, parked, or exiting.

Although the system achieved its major goals, there were some issues faced during the implementation process. The finite GPU memory limited fine-tuning of larger detection models, and occlusions and overlapping vehicles occasionally caused ID switches in tracking. Moreover, mobile rendering performance had to be optimized when handling frequent updates and a high number of vehicles and slots. Despite these issues, the system worked effectively as a functional prototype, proving that the integration of detection, tracking, and navigation has the potential to reduce parking search times, minimize congestion, and enhance the user experience.

7.2 Recommendation

One of the key recommendations for enhancing the system is the integration of a License Plate Recognition (LPR) system. Many parking facilities, particularly in shopping malls and office complexes, already employ LPR to monitor vehicle entry and exit. By combining LPR with the YOLO-based detection and DeepSORT tracking, the system would gain a secondary method of vehicle identification. This would significantly reduce cases of ID switches or tracking errors, as each vehicle could be consistently tied to its license plate rather than relying solely on appearance-based tracking. In turn, this integration would improve tracking accuracy, provide a more reliable user record, and open opportunities for added features such as personalized navigation or secure access control.

Another recommendation involves expanding the mobile application features to make the system more useful and user-friendly. While the current version focuses on real-time vacancy detection and navigation, future iterations could include a parking history log for users to review their past activity. A reservation feature could also allow users to book parking spaces in advance, reducing uncertainty during peak hours. Additionally, integration with payment systems would enable drivers to not only locate and reserve a space but also pay parking fees directly within the application. These expansions would elevate the system from a basic navigation tool into a comprehensive smart parking platform capable of supporting both operational efficiency and customer convenience.

Finally, it is essential to take user behaviour into account when refining the system's navigation logic. Real-world drivers may not always behave predictably—some may stop midway along a suggested route, double park in unauthorized areas, or even enter the parking lot from non-monitored entrances. Such behaviour can lead to mismatches between the system's suggested paths and actual driver movements. By designing navigation logic that adapts to these situations, such as recalculating paths dynamically or flagging irregular actions, the system can remain robust and reliable even under unpredictable conditions. This would ensure that the smart parking solution continues to provide realistic and practical guidance, thereby improving its readiness for real-world deployment.

REFERENCES

[1] H. Padmasiri, R. Madurawe, C. Abeysinghe and D. Meedeniya, "Automated Vehicle Parking Occupancy Detection in Real-Time," 2020 Moratuwa Engineering Research Conference (MERCon), Moratuwa, Sri Lanka. 2020, pp. 1-6, doi: 10.1109/MERCon50084.2020.9185199.

[2] Deng, Zhipeng & Sun, Hao & Zhou, Shilin & Zhao, Juanping & Lei, Lin & Zou, Huanxin. "Multi-scale object detection in remote sensing imagery with convolutional neural networks." ISPRS Journal of Photogrammetry and Remote Sensing. 2018, 145, doi: 10.1016/j.isprsjprs.2018.04.003.

[3] Z. Xie and X. Wei, "Automatic parking space detection system based on improved YOLO algorithm," 2nd International Conference on Computer Science and Management Technology (ICCSMT), Shanghai, China, 2021, pp. 279-285. doi: 10.1109/ICCSMT54525.2021.00060

[4] L.-C. Chen, R.-K. Sheu, W.-Y. Peng, J.-H. Wu, and C.-H. Tseng, "Video-Based Parking Occupancy Detection for Smart Control System," *Applied Sciences*, vol. 10, no. 3, p. 1079, Feb. 2020, doi: 10.3390/app10031079.

[5] Y. Zhang, Z. Guo, J. Wu, Y. Tian, H. Tang, and X. Guo, "Real-Time Vehicle Detection Based on Improved YOLO v5," *Sustainability*, Sep. 2022, vol. 14, no. 19, p. 12274, doi: 10.3390/su141912274.

[6] Kılıçkaya, Fatma Nur & Taşyürek, Murat & Öztürk, Celal. "Performance evaluation of YOLOv5 and YOLOv8 models in car detection". Imaging and Radiation Research, 2024 6, 5757, doi: 10.24294/irr.v6i2.5757.

[7] Kapania, Shivani & Saini, Dharmender & Goyal, Sachin & Thakur, Dr. Narina & Jain, Rachna & Nagrath, Preeti. "Multi Object Tracking with UAVs using Deep SORT and YOLOv3 RetinaNet Detection Framework". 2020, 1-6, doi: 10.1145/3377283.3377284.

[8] Bathija, A. and Sharma, G. "Visual Object Detection and Tracking Using Yolo and Sort." International Journal of Engineering Research Technology, 2019, 8, 705-708, doi: 10.32628/CSEIT206256.

[9] X. Chen, Z. Li, Y. Yang, L. Qi and R. Ke, "High-Resolution Vehicle Trajectory Extraction and Denoising From Aerial Videos," in *IEEE Transactions on Intelligent*

Transportation Systems, vol. 22, no. 5, pp. 3190-3202, May 2021, doi: 10.1109/TITS.2020.3003782.

[10] M. Li, M. Liu, W. Zhang, W. Guo, E. Chen, and C. Zhang, "A Robust Multi-Camera Vehicle Tracking Algorithm in Highway Scenarios Using Deep Learning," *Applied Sciences*, vol. 14, no. 16, pp. 7071–7071, Aug. 2024, doi: https://doi.org/10.3390/app14167071.

[11] J. Azimjonov and A. Özmen, "A real-time vehicle detection and a novel vehicle tracking systems for estimating and monitoring traffic flow on highways," *Advanced Engineering Informatics*, vol. 50, p. 101393, Oct. 2021, doi: https://doi.org/10.1016/j.aei.2021.101393

[12] Benny, L & Soori, PK 2017, 'Prototype of parking finder application for intelligent parking system', International Journal on Advanced Science, Engineering and Information Technology, vol. 7, no. 4, pp. 1185-1190. https://doi.org/10.18517/ijaseit.7.4.2326

[13] W. Yawai, "Smart Application for Car Parking System at Nakhon Ratchasima Rajabhat University", *IJC*, vol. 42, no. 1, pp. 41–58, Apr. 2022, Accessed: Sep. 17, 2025. [Online]. Available: https://ijcjournal.org/InternationalJournalOfComputer/article/view/1922

Appendix A: requirement.txt

anyio==4.9.0
CacheControl==0.14.3
cachetools==5.5.2
certifi==2025.1.31
cffi==1.17.1
charset-normalizer==3.4.1
colorama==0.4.6
contourpy==1.3.1
cryptography==45.0.5
cycler==0.12.1
deep-sort-realtime==1.3.2
filelock==3.17.0
filetype==1.2.0
firebase-admin==6.9.0
fonttools==4.56.0
fsspec==2025.2.0
google-api-core==2.25.1
google-api-python-client==2.176.0
google-auth==2.40.3
google-auth-httplib2==0.2.0
google-cloud-core==2.4.3
google-cloud-firestore==2.21.0
google-cloud-storage==3.2.0
google-crc32c==1.7.1
google-resumable-media==2.7.2
googleapis-common-protos==1.70.0
grpcio==1.73.1
grpcio-status==1.73.1
h11==0.16.0
h2==4.2.0
hpack==4.1.0
httpcore==1.0.9
httplib2==0.22.0
httpx==0.28.1
hyperframe==6.1.0
idna==3.7
Jinja2==3.1.5
joblib==1.5.1
kiwisolver==1.4.8
MarkupSafe==3.0.2
matplotlib==3.10.0
mpmath==1.3.0
msgpack==1.1.1
networkx==3.4.2
numpy==2.1.1
opencv-python==4.11.0.86

packaging==24.2
pandas==2.2.3
pillow==11.1.0
proto-plus==1.26.1
protobuf==6.31.1
psutil==6.1.1
py-cpuinfo==9.0.0
pyasn1==0.6.1
pyasn1_modules==0.4.2
pycparser==2.22
PyJWT==2.10.1
pyparsing==3.2.1
python-dateutil==2.9.0.post0
python-dotenv==1.0.1
pytz==2025.1
PyYAML==6.0.2
requests==2.32.3
requests-toolbelt==1.0.0
rsa==4.9.1
scikit-learn==1.7.1
scipy==1.15.1
seaborn==0.13.2
setuptools==75.8.0
six==1.17.0
sniffio==1.3.1
sympy==1.13.1
threadpoolctl==3.6.0
torch==2.6.0+cu118
torchaudio==2.6.0+cu118
torchvision==0.21.0+cu118
tqdm==4.67.1
typing_extensions==4.12.2
tzdata==2025.1
ultralytics==8.3.75
ultralytics-thop==2.0.14
uritemplate==4.2.0
urllib3==2.3.0

Appendix B: Poster