FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Using Mobile Edge Computing Technologies for Real-Time Cornering Assistance

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

BSc Matthias Karan
Matrikelnummer 1027663

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Hong-Linh Truong

Wien, 22. Jänner 2018

Matthias Karan   Hong-Linh Truong

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Using Mobile Edge Computing Technologies for Real-Time Cornering Assistance

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

BSc Matthias Karan
Registration Number 1027663

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Hong-Linh Truong

Vienna, 22nd January, 2018

Matthias Karan   Hong-Linh Truong

Erklärung zur Verfassung der Arbeit

BSc Matthias Karan
Fröbelgasse 33, Top 6-9, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. Jänner 2018

Matthias Karan

Acknowledgements

I would first like to thank my thesis advisor Priv.-Doz. Dr. Hong-Linh Truong. When I came towards him with a proposal of writing about this topic, he immediately showed great interest and added key aspects to form the idea into a scientific master thesis. During the time of writing the thesis, his door was always open to ask and I always got valuable feedback.

I would also like to thank the Austrian Road Safety Board (KFV) for introducing me to the very important and interesting topic of traffic safety and for providing useful data for this thesis.

Finally, I want to express my very profound gratitude to my parents, my sisters, my brother, friends and colleagues for supporting me throughout my years of study.

Kurzfassung

Während Autohersteller heutzutage in ihren neuesten Modellen immer mehr Fahrassistenzsysteme und gar vollautonome Software einbauen, sitzt aktuell die Mehrheit aller Autofahrer noch immer in Autos, die ohne diese Hilfstechnologien auskommen müssen. Bis der Zeitpunkt gekommen ist, an dem vollautonomes Fahren die Straßen vollständig erobert hat, werden vermutlich noch einige tausend Verkehrsunfälle durch menschliche Fehler passieren. Statistiken bestätigen: ein Großteil der Autounfälle passiert während dem Kurvenfahren.

Um für mehr Sicherheit im Auto auch bei älteren Modellen beizutragen, stellt diese Arbeit ein neues System vor, welches Fahrern in Echtzeit vor und während dem Kurvenfahren assistiert. Das System ist konzipiert um sowohl in Cloud- als auch in der neuartigen, sogenannten *Edge*-Infrastruktur, ausführbar zu sein. Ziel von Edge-Computing ist es, Recheneinheiten näher zum Anwender beziehungsweise zum Datenerzeuger zu bringen. In der Arbeit präsentieren wir Lösungen zur GPS-Datenverarbeitung, zum Kombinieren der GPS-Daten mit externen Datenquellen sowie zur Berechnung von Kurven und deren Eigenschaften. Als Ergebnis wird ein Prototyp-System entworfen und implementiert welches in simulierten Verkehrssituationen getestet wird. Der Prototyp wird anhand Leistung und Datengenauigkeit evaluiert. Basierend auf dem Prototyp, werden schließlich Kosten geschätzt, um solch ein System für alle Fahrer in Österreich bereitzustellen.

Abstract

Today, modern cars with advanced driver assistance systems (ADAS) and even fully autonomous driving software are on the rise and expectedly will reduce accidents in traffic in the future. But as of now, the vast majority of drivers still drive cars without these technologies. Until cars will be driving fully autonomous, human errors are going to lead to thousands of car accidents. Statistics show that one major cause of accidents is related to cornering.

To contribute to safer driving in any type of car, in this thesis we introduce a novel system that assists drivers in real-time while cornering. The system is designed in a way that it can be deployed to both the cloud and/or the emerging edge-computing infrastructure. The goal of edge-computing is to move computational units closer to where the data is produced. We also contribute solutions to processing location data, using and combining external data sources and calculating curves and properties. As a result, we design and implement a prototype that is tested against simulated traffic scenarios. In the evaluation we show how the prototype behaves in terms of performance and data quality. Based on the prototype, we finally demonstrate how much it would approximately cost to serve all drivers across Austria.

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

1.1 Motivation

Every year around 30.000 car accidents happen on Austria's streets [fV15]. In 2015, 479 of these ended fatally, making it 11% more deaths than in the previous year [Aus16]. As in previous years, the main causes of accidents still are inattention and excessive speeding. Though, accidents related to cornering also make up large amounts as studies from the year 2012, conducted by the ADAC in Germany[Ung12], have shown: Accidents related to cornering made up to 39% of all single-vehicle accidents. Especially for young-drivers (aged 18-24), the numbers of accidents related to cornering were very high. Excessive speeding during cornering made more than 10% of accidents.
In order to reduce accidents related to cornering and contribute to safer car driving, the goal of this thesis is to provide a system that assists drivers in real-time during cornering using state-of-the-art mobile edge computing models. While most existing solutions (see Section 2.2) help users to adapt safer driving styles by detecting risky driving maneuvers and giving tips afterwards, none of these solutions yet allow real-time assistance for cornering. In Austria such a system would be especially contributing to reducing accidents since twisty roads with dangerous curves occur frequently on the countryside, rural areas and mountains.

1.2 Problem Statement

Using location data from cars together with curve- and recommendation-algorithms, an application can alert of dangerous curves ahead and warn if a driver is approaching a curve too fast. A possibility to realize a client application in the car is to create a smartphone application that can be installed at the driver's smartphone. Since processing units and memory are limited on smartphones, in order to accomplish the computation tasks of above mentioned services in real-time, computational steps can be offloaded

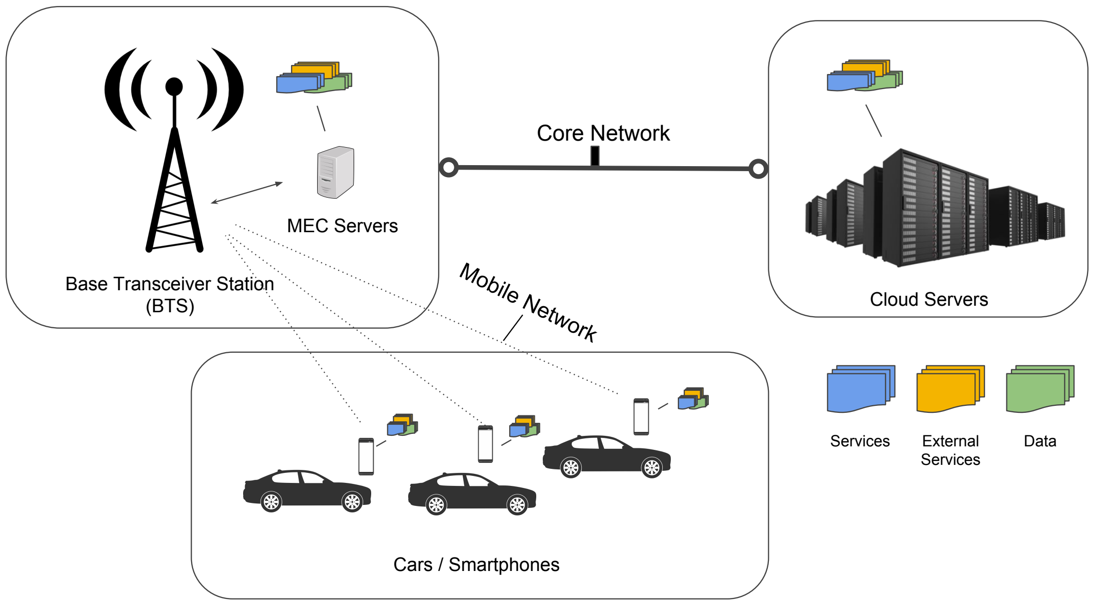


Figure 1.1: Overview of the system's underlying MEC architecture
Icons used from `https://pixabay.com` and `http://clipart-library.com/clipart/1007672.htm`

either to a centralized cloud model or a MEC server model. Compared to the cloud, the former model, referred to as *Mobile Edge Computing* is a relatively new paradigm, allowing to put computation units at the edge of the cellular network, hence directly at cellular base stations (BTS) [Wik]. The functionalities of the system will therefore be mapped onto a distributed system consisting of three components: *Client Applications* that are placed within the car, *MEC Servers* located at BTS and centralized *Cloud Servers*. The architecture is sketched in Figure 1.1.

Although there are existing systems that handle location data from smartphones in real-time, for example [LKA$^+$16] continuously (re)-calculate routes for safer driving, in order to allow curve-detection in real-time, our proposed system requires novel strategies to handle computation and latencies and therefore following questions remain to be answered:

- **Q1 - Data concerns:** What data and at which frequency must data be gathered from driving cars? What additional data needs be combined with the driver's location? What data is produced by the system and who could benefit from the new data?
- **Q2 - Separation of tasks:** The system needs to perform different computational tasks. How can the tasks of the system be distributed and coordinated across the different components of the system? What distribution of tasks enables best response times and allows the system to perform in real-time?

- **Q3 - Algorithms:** Efficient and reliable detection of curves will be one key enabler of this system. How can curves be detected and upcoming curves be predicted efficiently around the location of the driver? How can information of curves and data from external services be used to deduct meaningful assistance tips for car drivers before/during cornering?
- **Q4 - Evaluation:** How many drivers can the system handle in parallel? How well does the system perform at slow network conditions and inaccurate or missing GPS data? What are the costs to run the system in cloud or edge infrastructures?

The goal of this thesis is to explore possible answers to above research questions with the development of a real-time-cornering-assistance system.

1.3 Approach & Contribution

In order to provide answers to the stated research questions, in our approach we start by analyzing all relevant aspects about input data, processing data and desired output data of the system. We then identify tasks and design a system that is able to distribute them across multiple services that execute in existing cloud or edge infrastructures. To handle all identified tasks we present new algorithms. Finally we demonstrate our approach with implementing a prototype and evaluate it.

This thesis introduces a novel system that assists drivers in real-time while cornering and is designed in a way that it can be deployed to existing cloud and emerging edge-computing infrastructures. The contributions of this thesis are:

- We identified all necessary data attributes from drivers and external data sources to realize a cornering-assistance-application. For each data source, we determined their frequencies of collection and retrieval, propose how to efficiently fuse them together and identified the desired output data of the system. Together with the research of the state-of-the-art, this forms the foundation of the following design of the system and algorithms.
- We designed an architecture of a distributed system in a way that it can be deployed to existing cloud but also emerging edge infrastructures. To that end, we first made clear assumptions about the underlying infrastructure. Based on the assumptions we proposed an architecture using the microservices pattern that supports virtualized MEC infrastructures, separates tasks and concerns and enables scalability. We gave a detailed design of all software components and showed how services are orchestrated.
- In order to accomplish the tasks of the cornering assistance application, we presented and implemented new algorithms. Our curve detection algorithm uses geometric properties of map data and is able to not only identify curves, but also

calculate properties such as length or radius. Using calculated properties of curves in combination with weather data, our recommendation algorithm derives road conditions and based on that calculates safe speeds for entering a curve. To signal warnings when drivers are approaching curves, out of many possible curves in an area, we implemented another algorithm that is able to predict the next upcoming curve along the driver's path from a given location. Since our system is deployed to a heterogenous infrastructure, where many servers with different capabilities execute at different locations, we also presented an approach of how to load balance requests depending on server load and location of clients.

Based on above contributions and to demonstrate a working cornering-assistance-system, we implemented an open-source prototype. The prototype implementation is available at: `https://github.com/rdsea/EdgeCorneringAssistance.git`

1.4 Structure of the Thesis

In Chapter 2 existing driving applications are examined and the state-of-the-art of MEC and curve detection using map data is presented. Chapter 3 presents relevant aspects about input data, processing data and desired output data of the system. The architecture of the cornering-assistance system is designed in Chapter 4. Chapter 5 describes the design of algorithms that are used within services of the system. In Chapter 6 the implemented prototype is described. The prototype is evaluated in Chapter 7. Finally, the findings of this thesis are summarized and future work is outlined in Chapter 8.

CHAPTER 2

State of the Art

2.1 Overview

As a starting base for the thesis, papers about existing driver monitoring and assistance applications, MEC and curve detection algorithms using map data, describe the state-of-the-art and are discussed in this chapter.

2.2 Driver Monitoring and Assistance Applications

In the following, we will present systems and approaches that have been developed to detect patterns in car driving using smartphones with focus on different conditions. While most of the systems that will be presented focus on detecting obvious driving maneuvers, such as accelerations, braking or lane changes, other relevant conditions such as human distraction, drowsiness, road conditions, traffic situation or ecological behavior can be detected. The state-of-the-art of driver behavior applications is split into following categories of detection: *driver condition*, *ecological driving*, *environmental condition* and *driving maneuvers* and for each of the categories one sample approach is picked and compared by the following main three questions:

1. Which driving behaviors are detected?
2. Which types of data are collected and how is the context supported?
3. What is the underlying infrastructure, which algorithms are used for detection and how well do they perform?

Environmental Conditions: Ruta et al. [RSSB10] introduced a knowledge-based system to distinguish between aggressive and regular driving style, even or uneven roads and

high vs. low-dense traffic. Based on detected behaviors and conditions a prototype app suggests tips for safe driving. In case of heavy rain, wind or high speed driving, the system for instance suggests to reduce speed or activate the Anti-lock Braking System (ABS) and Electronic Stability Control (ESP). Data from the smartphone's Global Position System (GPS) and the acceleration sensor is used to detect road conditions, whereas engine rotation signals from an installed On Board Diagnostics Device (OBD II)[obd] device is used to classify driving styles. The data is collected in fixed intervals (60 seconds) and then evaluated by a WebService that performs ontology-based semantic matchmaking to classify driving behavior and road conditions. The detection is supported by further data from external web-services that include weather data, road information and location information. The detection was tested in three different environment settings, with two test drives each and in all cases the system was able to provide meaningful tips according to the setting.
A very different detection method has been implemented by Bhoraskar et al. [BVRK12], where GPS, accelerometer and gyroscope sensor data is used to detect bumps or potholes in roads and braking as the only driver behavior. This method initially performs calibration. The detection is based on a machine learning approach. In a first step, a k-Means clustering algorithm classifies sample points of test data as either "smooth" or "bumpy". After this classification step a Support Vector Machine is trained over time to distinguish between smooth and bumpy. To also support driver behavior detection, the same approach is applied to detect braking maneuvers. During an evaluation 29 of 37 braking events and 18 out of 20 bump events were correctly identified.
In their cloud-based solution, Zhaojian et al. [LKA+16], propose an application with the goal of determining routes with low risk of accidents as well as fast travel time. GPS locations and the desired destination are used to calculate the optimal route and their planning algorithm avoids risky roads. Risky roads are detected using a risk prediction model that uses road information, traffic information and weather data. In case the driver leaves the planned route, the GPS location of the driver is continuously sent to their cloud server in order to recalculate safety-based routes.

Driver Condition: With the aim of reducing incidents caused by drunk driving, Dai et al. [DTB+10] proposed a smartphone application that detects driving maneuvers related to drunk driving by using only acceleration and gyroscope sensor. The focus in this approach is to detect patterns like hard accelerations, abrupt breaking or left/right turn movements like drifting or swerving on straight roads and check if they correlate with typical drunk driving patterns. Compared to [RSSB10], the application does not restrict the phone's position within the vehicle which is why a calibration algorithm is performed first. After calibration, data is captured and drunk driving patterns are recognized through windowing and variation thresholding. The false positive rate of the detection in the testing environment was very low when the phone was in fixed position. In this case the performance of the detection was very good, although the test data set for drunk driving was imitated and might therefore not strictly be differentiable from regular patterns like lane changing.
With the use of the front camera of smartphones and complex image processing algorithms

You et al. developed *CarSafe* [YMdOB$^+$12], an application that can detect driver distraction or drowsiness. The back camera is also used to alert risky distances to other cars. Like [DTB$^+$10], additionally sensor data from GPS, accelerometer and gyroscope is used to detect risky accelerations, breaks or lane changes. Evaluation of the algorithm showed that CarSafe is able to detect dangerous driver conditions at an acceptable 75% detection rate. Though the performance of the detection is strongly influenced by lighting conditions (shadows, over-exposure, darkness).

Ecological Driving: With focus on ecological aspects of driving, Araujo et al. [AIdCA12] designed the Android application *DrivingCoach* that uses a combination of smartphone data (GPS and acceleration sensor) and data from an OBD II device (throttle signal, fuel consumption, engine rotation) to detect unecological driving behavior such as high fuel consumption and early gear shifting. Additionally, the application can distinguish between the two road conditions "urban" or "highway". Raw data from sensors is summarized using average and min/max operators and then encoded into fuzzy logic values. During classification, firstly the driving condition as well as fuel consumption are mapped to fuzzy values ("urban/highway" and "very poor/good consumption") and in the second stage the most suitable hint is derived by a fuzzy evaluation algorithm and delivered to the user via a graphical interface.

Driving Maneuvers: MIROAD (Mobile Sensor Platform for Intelligent Recognition of Aggressive Driving) is a system designed by Johnson et al. [JT11] that is able to detect speeding, acceleration, breaking, (u)-turns and lane changing and classify them as aggressive or very aggressive. Data is fused from GPS, acceleration sensor, gyroscope and magnetometer and in a first step a driving event's start and end is detected using a simple moving average filter. As soon as an event is triggered, the system additionally starts video capturing using the rear camera. The actual detection of different driving patterns is done using a Dynamic Time Warping (DTW) algorithm. With this method it is possible to compare signals of different lengths and therefore it is possible to compare new captured driving patterns with some pre-defined patterns to classify driving patterns accordingly. During evaluation, the system performed very well and 97% of aggressive driving events were detected. Since regular driving maneuvers do not trigger the algorithm, this approach is only suitable for detecting aggressive events.
Other approaches that also used DTW algorithms for detecting driving maneuvers have been implemented for instance by Eren et al. [EMAY12] or Saiprasert et al. [SPT15]. Like [AIdCA12], in their approach Paefgen et al. [PKZM12] used a combination of smartphone sensor data and OBD II data to detect accelerations, breaks and turns. Similar to [DTB$^+$10], a calibration algorithm is included and thus the smartphone can initially be in any arbitrary position. The detection makes use of simple thresholding to detect driving behaviors. The main focus of this paper was to find out if smartphone measurements are reliable enough to detect driving behaviors by comparing smartphone results to OBD II results. During evaluation it was revealed that significant correlations between mobile and OBD II event counts exist and therefore shows that smartphone applications for driving behavior detection can keep pace with on-board solutions. Though,

good results highly depended on the position of the smartphone and some positions resulted in higher error rates.
Facing the problem of heavy noise in sensor data of smartphones, caused by free placement of the device within the car, in their paper, Li et al. [LLL+12] present algorithms to correct noise or drift in sensor data. Their system also is one of the first driving monitoring systems for smartphones that dynamically compensates disorientation of the device by using wavelet-based analysis. The paper presents a personalized driving behavior monitoring and analysis system for hybrid vehicles. Compared to other approaches the authors focus mainly on improving sensing technologies and removing noise. Sensor data from GPS, accelerometer, gyroscope and magnetometer is collected, pre-processed using a low-pass-filter and sent to a server for further analysis as soon as the WIFI-signal is available. Similar to [PKZM12], the authors used data from an OBD II device (speed, acceleration) as ground truth and compare results with processed signals of the smartphone. Evaluation of the proposed system resulted in 0.88 – 0.996 correlation values between processed smartphone signals and OBD II signals.
Considering data collection, the Android application *DrivingStyles* developed by Meseguer et al. [MCCM13], similarly to [RSSB10], [AIdCA12] and [PKZM12] uses smartphone data, in this case only GPS and additionally data from an OBD II device (throttle signal, engine rotation, speed, acceleration) to analyze if a driver has a "quiet", "normal" or "aggressive" driving style. In this approach, after a trip, the recorded data is sent to a WebService where a Neural Network algorithm classifies driving styles. The Neural Network is also trained to distinguish road conditions from urban, suburban or highway roads. Evaluation resulted in 98% degree of accuracy in classifying road conditions and 77% accuracy on driving styles.
Similar to [AIdCA12], the developers of the application *SenseFleet* by Castignani et al. [CDFE15] used fuzzy set evaluation to successfully detect risky accelerations, breaks and lane changing from sensor data (GPS, accelerometer, gyroscope and magnetometer). Additionally, they consider weather data for event detection and the device's position can be calibrated. Users of the app receive scores for their driving (e.g. if risky events occurred, the score decreases) and scores are aggregated on a web server for further analysis. Evaluation resulted in an event-detection rate of over 90%, though achieving these results required long calibration times.
Compared to all other approaches, Wahlstroem et al. [WSH15] focus on detecting sliding or rollovers during cornering using only location data from a global navigation satellite system (GNSS) on a smartphone. In order to increase accuracy of location data they pre-process GNSS data with a kalman filter. Dangerous events during cornering are detected by estimating vehicle dynamics using physical laws. The paper only presents a framework how dangerous vehicle cornering events could be detected using smartphones. An actual detection system has not been implemented so far.
As stated before, [JT11] and most other approaches so far only detect dangerous or aggressive maneuvers. Daptardar et al. [DLR+15] designed a novel algorithm to also detect normal maneuvers using acceleration and gyroscope data from smartphone sensors. After low pass filtering sensor data to reduce noise and applying a 4-state kalman filter

to correct velocity and acceleration data, their application is able to detect accelerations, breaks, turns and lane changes using two different algorithmic approaches. While longitudinal events, e.g. accelerations and breaks, are detected using a jerk energy based technique, lateral events, e.g. turns and lane changes, are detected using a novel hidden-markov-model based technique. During evaluation of collected datasets both approaches were able to detect driving maneuvers with an accuracy of 95%.

2.2.1 Summary of existing Driver Monitoring Systems

Figure 2.1 gives an overview of above described driver monitoring and assistance systems and compares them in terms of features and system design. As it has been pointed out, the compared solutions are able to detect lots of driving related events in different categories. Except for [LKA+16], that continuously re-calculates routes, all other above presented approaches provide assistance only after a certain driving event occurred. In our approach though, we need to provide assistance before the driver actually enters a curve. Hence, our approach can not be compared in all terms as it has been done in this section. Still, similar to almost all presented approaches, also in this thesis the GPS of the smartphone will be used to track the driver. While in [RSSB10] and [BVRK12] for instance, every GPS coordinate is transferred to some distributed server, in our thesis we will actually only send very few locations and only at a certain time. While most other approaches use many other sensors of the smartphone, for instance [JT11], [BVRK12] and [AIdCA12] additionally use the acceleration, gyroscope and magnetometer sensors, our approach needs no other sensors. Using fewer sensors can save a lot of battery usage of the driver's smartphones. [RSSB10], [AIdCA12] and [LKA+16] make use of contextual support for their application in the form of weather, road or even traffic information. Also in our approach we retrieve weather information to derive road conditions. For future works though, it might also be useful to incorporate road or traffic information for our cornering application. Regarding detection of driving maneuvers, only one of the presented approaches deals with cornering. The presented approach of Wahlstroem et al.[WSH15] is the only one that considers detecting dangerous cornering movements such as sliding or rollover. Compared to this thesis though, an actual detection system has not been implemented so far.

2.3 Edge Computing

Often called the successor of Cloud Computing, Edge Computing, also often referred to as "Fog Computing", is an emerging paradigm that provides cloud and IT services within the close proximity of end-users, therefore moving computing applications, data and services away from some central nodes (the *core*) to the other logical extreme (the *edge*) of the Internet [GLME+15]. While in cloud computing data is pushed to centralized computing infrastructures and analyzed there, in edge-centric computing, analytics and computations are done in close proximity or even directly where data is generated. Facing

	ruta	dai	miroad	drivingcoach	paefgen	wolverine	li	eren	carsafe	drivingstyles	wahlstroem	sensefleet	saiprasert	daptardar	zhaojian
Published	2010	2010	2011	2012	2012	2012	2012	2012	2012	2013	2014	2015	2015	2015	2016
Detection															
Driving Behavior															
acc		x	x	x	x							x	x	x	
brk		x	x	x	x	x						x	x	x	
turns		x	x		x				x					x	
u-turns			x												
lane change / swerving			x						x			x	x	x	
speeding	x		x						x			x			
"aggresive vs. regular" style	x									x					
"safe vs. risky" style								x							
distance to other cars									x						
risky cornering											x				
Ecological Aspects															
(high) fuel consumption				x											
(early) gear shifting				x											
Environmental Conditions															
road classification	x			x						x					
bump detection						x									
traffic situation	x														
weather conditions	x														
wind conditions	x														
risky roads															x
Driver Condition															
drunk driving		x													
distraction/drowsiness									x						
Data Collection															
Smartphone															
GPS	x		x	x	x	x	x		x	x	x	x	x		x
Acc	x	x	x	x	x	x	x	x	x			x	x	x	
Gyro		x	x		x		x	x	x			x		x	
Magnetometer			x			x	x	x				x			
Camera (rear)			x						x						
Camera (front)									x						
OBD II															
Throttle Signal				x						x					
Instant Engine Fuel consumption				x											
Engine rotation	x			x						x					
Speed	x														
Acceleration							x			x					
Context Support															
Weather Data	x											x			x
Road Information	x														x
Time of Day												x			x
Traffic information															x

Figure 2.1: Comparison of state-of-the-art approaches

challenges that arose with cloud computing, the main improvements of edge computing are to reduce end-to-end latencies and lessen the bandwidth of the network.

To achieve these improvements in edge-centric infrastructures, new types of computing and storage resources, so called "Fog Nodes" [BMZA12] or "Edge Nodes", are placed at the edge of the network. Cloudlets are resource rich computers in the near vicinity of mobile users. They can be installed within a wireless network, instantiate virtual machines and run custom software on them. Examples for locations of cloudlets can be in a private home network, wifi-hotspots in airports or within a railway carriage [VSDTD12]. Compared to usual data centers, a micro data cloud includes all the components of a data cloud in one standalone system that can be deployed anywhere indoors or outdoors and therefore is ideal for edge computing.

2.3.1 Mobile Edge Computing

Mobile Edge Computing (MEC), a niche of edge computing, is an emerging technology that provides cloud and IT services within the close proximity of mobile subscribers that focuses on applications for mobile network providers[AA16]. There are already some telecommunication operators that use Cloud Radio Access Networks (C-RAN) to move cloud resources away from the core network to the base stations. If computing resources are moved closer to the cell site, response times can be improved significantly[KMJ16]. Just recently ETSI ISG (industry specification group)[ETS17], who are the founders of the Mobile-edge Computing industry initiative, announced that they will rename *Mobile Edge Computing* into *Multi-access Edge Computing*[Mor16]. Nevertheless the abbreviation *MEC* will remain.

2.3.2 MEC Platform

As described in their Introductory Technical White Paper[PNC$^+$14], ETSI ISG design the MEC server platform to consist of a hosting infrastructure (IaaS) and an application platform. Through virtualization, the application platform provides an abstraction layer for the hosting infrastructure. Hence developers will be able to deploy and execute their MEC Applications within Virtual Machines, „allowing complete freedom of implementation“[PNC$^+$14].

2.3.3 Existing Architectures using MEC

Facing the challenge of processing and storing high cumulative data rates of video material from cameras, [SSX$^+$15] proposed *GigaSight*, a system that stores crowd-sourced video content that allows users to keep control about privacy preferences, for instance denaturing video frames by cutting off faces. Users send their captured video material to GigaSight's architecture which consists of two main components: a centralized cloud storage and so called *Cloudlets* that in the architecture form a tier between smartphones and the centralized cloud. Computation, therefore processing the videos, is solely done on virtual machines at near cloudlets. The centralized cloud only stores the final results.

One key challenge in this approach was to ensure the privacy of the user's video content. Therefore each uploaded video by a user was assigned to a distinct virtual machine at the cloudlet.
While [SSX+15] focused on processing uploaded videos, [LFHAB16] created a new architecture for video streaming using MEC. With their system they propose to improve the performance of Dynamic Adaptive Streaming over HTTP, often referred to as *DASH*. Compared to a traditional DASH system architecture, they are placing a MEC server between a content provider and the client requesting a video to stream. The MEC service in general is responsible for selecting optimal quality/bitrate video and representations for each client. The main goal of the MEC service therefore is to reduce network congestion. During testing, compared to standard DASH solutions, their novel strategy allowed up to 54% shorter and 4 times less frequent interruptions.
A different use case for using MEC technology, presented by ITSE ISG, is Active Device Location Tracking [PNC+14]. In their exemplary use case, they propose an architecture for tracking mobile devices solely over the network and therefore being completely independent of GPS. This approach would require to run geo-location algorithms directly at MEC Servers, measuring a device's location via available network information at the cellular base station. Besides location tracking, in the above mentioned technical whitepaper they also present other possible architectures showing how MEC could improve systems in the area of video analytics, augmented reality or content caching for instance.
In their futuristic vehicular scenario, [GWA+17] discuss possible use cases of using MEC within the automotive domain. Their proposed future system called "electronic horizon" shall provide a "detailed preview of the road ahead". Proposed features include functions like for instance "adaptive cruise". With this feature it shall be possible to automatically reduce a car's velocity to catch green lights. The system therefore needs to know about traffic situations in real-time. Another interesting feature is "adaptive headlight adjustment". When "electric horizon" knows of upcoming hazards, this feature automatically adjusts headlights such that drivers can spot them. In their paper they strike out that getting all the data necessary from different sources to enable these features on the one hand, and uploading data from the drivers and sensors of the car on the hand hand, still is a big challenge. They propose how MEC can be the enabler for these features.

2.3.4 Summary of existing Edge Architectures

All presented architectures show how edge computing technologies can be used for modern applications. Similar to [SSX+15], in our approach we also combine edge with cloud resources. While in their approach, they only move a central storage to the cloud and do all the computation in the edge, in our approach we do not restrict the computation to execute only in the edge. Another difference to our approach is that we will not use Cloudlets in the edge. Instead we design our system to be deployable to MEC servers that run at Base Transceiver Stations (BTS). As deploying applications to MEC servers at BTS is not yet possible, these servers are simulated in the thesis.
While in our approach we use GPS to track drivers, the presented exemplary use case

of Active Device Location Tracking[PNC+14] would be a great improvement for our application and in case it is ever realized might become very interesting for future works. Similar to our approach, [GWA+17] proposed a future system in the automotive sector using MEC. Besides their new features presented earlier, they also mention a feature to detect curves ahead. Our cornering-assistance-application can therefore be seen as one first implementation of a feature of the "electronic horizon".

2.4 Curve Detection using map data

Due to freely available geographic data, like for instance OpenStreetMaps (OSM) [Ope17a], nowadays it is possible to gain information of road segments easily. One existing algorithm to detect curves from public available map data has been implemented by Adam Franco in his project “Curvature”[Fra17]. His open-source algorithm is able to read in Open Street Map data, analyze the shape of every road and find out how twisty a road segment is. "The goal of this program is to help those who enjoy twisty roads (such as motorcycle or driving enthusiasts) to find promising roads that are not well known"[Fra17]. Compared to the presented algorithm, in our approach the focus lies not on analyzing complete roads and classifying them in terms of how "twisty" they are. Instead we want to detect every single curve and its properties as accurate as possible. While Curvature is an application to find fun roads to drive on, the goal of our approach is to assist drivers safely through curves. While our algorithm is based on similar concepts, for instance using OSM data and formulas for calculating the radius, the detection method is different and was designed independently.

2.5 Summary

In this chapter existing approaches of driver monitoring and assistance applications were presented and compared in terms of features. We also presented existing MEC applications and examined their architectures. Lastly we presented a detection algorithm that uses similar concepts to our approach. As it has been pointed out in Section 2.2, currently no system comprehensively deals with detecting cornering maneuvers and especially there is not yet a solution to assist drivers during cornering. Using state-of-the-art MEC architectures as described in Section 2.3.1 and curve detection algorithms described in Section 2.4, the thesis will combine these technologies and present a novel approach in the field of driver assistance applications and MEC.

CHAPTER 3

Data

3.1 Overview

Before designing the architecture of the system and making concrete choices about implementations, at first data that is required and processed by the system is analyzed and described. This chapter reflects the way how data is transferred to and through the system.

3.2 Data Sources

In order to detect upcoming curves and give assistance while cornering, the system requires to receive, process or combine data from multiple sources. External data sources, provided by third parties are used to enhance and combine the driver's data with useful contextual information. While map data will be used for curve detection, weather data will help to deduct assistance tips.

3.2.1 Driver

The most important source is the driver. More specifically the source is the GPS that can be obtained from a client application installed on the driver's smartphone or possible other means within the car. During a trip, the application at certain points will collect location information from the GPS sensor. Today's GPS sensors of different smartphone manufacturers and operating systems usually contain similar readings, specifically in Android for instance, a location can consist of latitude, longitude, timestamp and other information such as bearing, altitude, velocity and accuracy[Goo17a]. For detecting upcoming curves, most importantly latitude and longitude are needed. To also detect slopes of curves, altitudes can be helpful. For providing accurate assistance on the detected curves, bearings (i.e. the driver's direction, also called azimuth) and timestamps will be

Attribute	Description	Example
DriverID	Unique identifier of a Driver.	931
Timestamp	Unix timestamp of GPS sensor readings.	1495639690
Latitude/Longitude	Location in earth coordinates	48.2084114,16.371282
Altitude	Altitude in meters above the WGS 84 reference ellipsoid.	325m
Bearing	Horizontal direction of travel of the device in degrees (0.0, 360.0].	172°
Speed	Speed in meters/second over ground	25m/s

Table 3.1: Summary of driver's data attributes

important. In case the GPS is obtained from a smartphone application, all necessary data attributes can be collected very easily. The data attributes are summarized in Table 3.1.

Another very important data concern that will affect the reliability of curve detection and assistance are GPS inaccuracies of mobile phones. In their evaluation of GPS sensors of mobile phone's from 2011, [ZB11] revealed that positions provided by mobile phones have a median horizontal error of between 5.0 and 8.5 meters. An evaluation of all GPS locations from the trip database provided by the Austrian Road Safety Board (KFV) (see Section 7.2.1) showed that the highest available accuracy provided by all 26 different Android test-devices was 3 meters. This high accuracy value of 3 meters was achieved in 84.65% of all recorded locations. Although this shows that GPS sensors perform quite well on smartphones on Austria's roads, still, GPS inaccuracies and outages are problematic for traffic applications.
In the near future, GPS inaccuracies might be not be such a big issue anymore. On September 21, 2017, Broadcom announced that they are currently "sampling the first mass-market chip that can take advantage of a new breed of global navigation satellite signals and will give the next generation of smartphones 30-centimeter accuracy instead of today's 5 meters" [Moo17].

3.2.2 Map Data

To calculate curves from locations, information about roads in form of static map data is needed. There are many online-services providing map APIs such as Google Maps[Gooa], Bing Maps[Mic], Here Maps[Her] or OpenStreetMaps[Foue] to name the most famous ones. Requirements for the application regarding map data are to cover all roads of Austria and have access to road data via an API, ideally not restricted by query limits. For the application the decision was made to use OpenStreetMaps since it is open-source, provides many APIs and allows downloading any map data of the world. In OpenStreetMaps, each geographic coordinate within a road segment holds a “highway” tag, indicating it is part of a road [Foud]. Having a GPS coordinate it is therefore possible to query if the point is part of a road segment. Since one road segment is formed by multiple geographic points, so called “nodes” [Foud], it is possible to detect if a road segment forms a curve and calculate its properties such as radius or length. Map data for a specific area, i.e. in the area of a driver’s current location, can be queried using OverpassAPI [Fou17] via a Bounding Box (see Section 3.2.3). OverpassAPI can either be used as an online-service with querying limitations or installed on an own private instance without any limits. Data attributes queried from OpenStreetMaps using OverpassAPI are summarized in Table 3.2.

Attribute	Description	Example
Bounding Box	Bounding Box of an area.	[52.5311,13.3644;52.5114,13.4035]
Way	Ordered list of nodes.	{’id’:1234,’nodes’:[2345,3456],’tags’: {’highway’:’residential’,’name’:’Foo’}}
Node	Specific GPS point	{’type’: ’node’,’id’: 2345, ’lat’: 50.7468,’lon’: 7.1563}

Table 3.2: Summary of map data attributes

3.2.3 Bounding Box

In order to efficiently store and query curves, they can be aggregated by their location using a so called *Bounding Box.* A Bounding Box is a special bounding volume for an object in form of an axially parallel box where the object touches all four sides of the box[Kow12]. Each curve belongs to a bounding box that is of dynamic size. For a location of a driver, a bounding box can be calculated that represents a greater area around its location. To avoid that multiple bounding boxes are overlapping, the boxes are calculated based on *Geohashes*[geo].

Geohashing is a way of encoding latitude and longitude pairs into one string. Using geohashing, the world map is divided into a grid. Depending on how many bits are used for creating a geohash, the precision varies. The simplest geohash of a location could be encoded by using only one single bit. In this case, the whole world map is divided horizontally into two pieces. The left side of the grid would be encoded to *0* and the right side to *1.* To increase the precision of a location, sub-dividing the world map can be done until we get to street-level or beyond. At that point, more and more bits are needed to generate the geohashes. To reduce the size of the resulting hash, the bits can be represented as alphanumeric characters using 32 bit encoding. An example of a geohashed location at street level using a precision of 8 characters would be: *5pf666y7.* The chosen character precision of a geohash also defines the hash’s bounding box. The

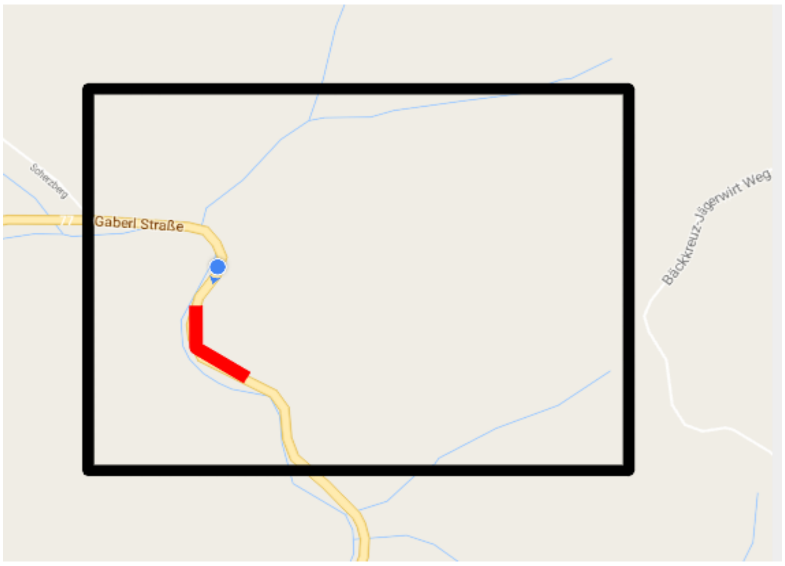


Figure 3.1: Example of a location and the bounding box of the corresponding geohash

larger the precision of the geohash, the smaller the corresponding bounding box will be. Figure 3.1 shows a location and its geohashed bounding box using a precision of 6 characters.

3.2.4 Weather Data

Generating appropriate cornering assistance tips depends a lot on the properties of the curve, but also on current weather conditions. Different weather situations can completely change road surface conditions or visibility conditions and therefore require an adapted driving behavior by the driver. In general the following weather conditions will have an impact on the driver:

- Strong Winds / Storms
- Heavy Rain / Wet Roadway
- Fog / Mist
- Snow
- Glaze

A requirement for our cornering-assistance-application therefore is to receive weather data that allows to distinguish between these conditions. Similar to map services, there are lots of available online-services providing weather APIs. OpenWeatherMap[Opeb] offers an API to query the current weather for a specific location with lots of different specific

Attribute	Description	Example
Latitude/Longitude	Location of weather information	48.2084114,16.371282
Weather	Short description of current weather condition	{'id': 803, 'main': 'Clouds', 'description': 'broken clouds'}
Temperature	Current temperature in Kelvin	281.83K
Wind	Current speed and direction of wind	{'speed': 6.7,'deg': 350}
Rain	Volume of rain for the last 3 hours	{"3h": 1.85}

Table 3.3: Summary of weather data from OpenWeatherMap (OWM)[Ope17b]

conditions allowing to detect all above mentioned conditions. The service can be used for free with a limitation though of not more than 60 queries per minute[owm]. As an alternative, ZAMG provides weather data of 21 weather stations in Austria completely for free[fMuG]. Compared to OpenWeatherMap, the drawbacks are that the exact location cannot be queried and no explicit weather conditions are given.
For the thesis, the free plan of OpenWeatherMap will be used with respect to the given limitations. Queried data attributes from OpenWeatherMap servers are summarized in Table 3.3.

3.3 Cornering Assistance Data

Data sources as specified in Section 3.2 produce large amounts of data. The system will process data of input sources and produce data that is stored within the system itself, consumed by the end-user or possibly shared to third parties. This section describes data that is used to assist drivers.

3.3.1 Curves

One of the central outcomes of the system will be detected curves within the area of the driver's location. Detected curves along with their properties will need to be stored at some places within the system. Key properties of a curve are: *Radius*, *Length*, *Slope* and *Location Points*. After successful deployment of the cornering assistance application and its distributed system, it is expected that a collection of curves will emerge and will grow with users and duration of execution. Having a broad collection of curves and their properties throughout Austria's road network might be of interest for third party organizations or open data platforms such as Austria's Open Government Data [dSW17]. Although the thesis will not consider designing or implementing redistribution services, the storage of curves is designed in a way that its possible to continuously export and share output data.

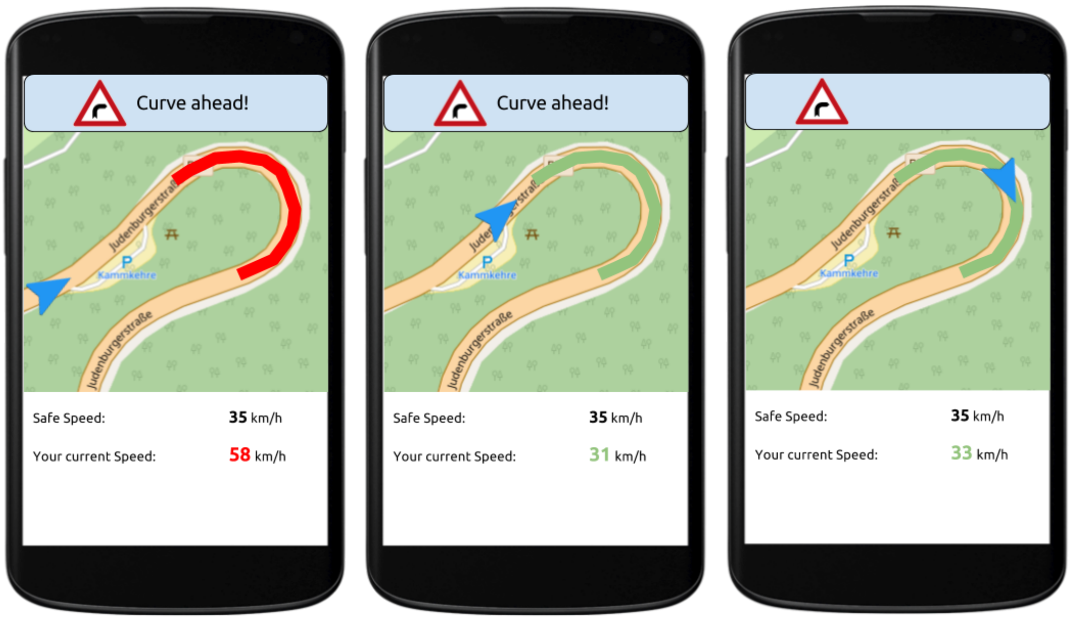


Figure 3.2: Example of how recommendations can be used to create a GUI for safe cornering

Map extract used from `http://www.openstreetmap.org/`

3.3.2 Recommendation and Assistance

While drivers are cornering, meaningful assistance shall be presented to the end-user via a graphical user interface deployed on the application of the drivers. The assistance shall be given in form of *Recommendations* that are computed by the system's algorithm. To guide drivers safely through a curve, the assistance shall begin before a curve is approached. Firstly, the assistance therefore signals a warning on the screen when a dangerous curve is approaching. Warnings are shown as soon as the driver is less than a specified threshold of meters away from an upcoming curve. Secondly, the recommendation outputs a value that determines the recommended speed a driver shall have to safely drive through the curve. Figure 3.2 sketches how these recommendations could be used to create a user interface that assists drivers step by step while cornering. As stated earlier in the thesis, the client application with user interfaces that actually provide the assistance to the drivers are not the focus of this thesis. This section just showed how the implemented algorithms and produced data can finally be wrapped into an assistance application.

3.4 Data Fusion

After data is collected or queried from the different sources and before the analysis of data can be initiated, as a first step data fusion needs to be done. The goal of performing data fusion is to combine relevant information from two or more data sources into a single one

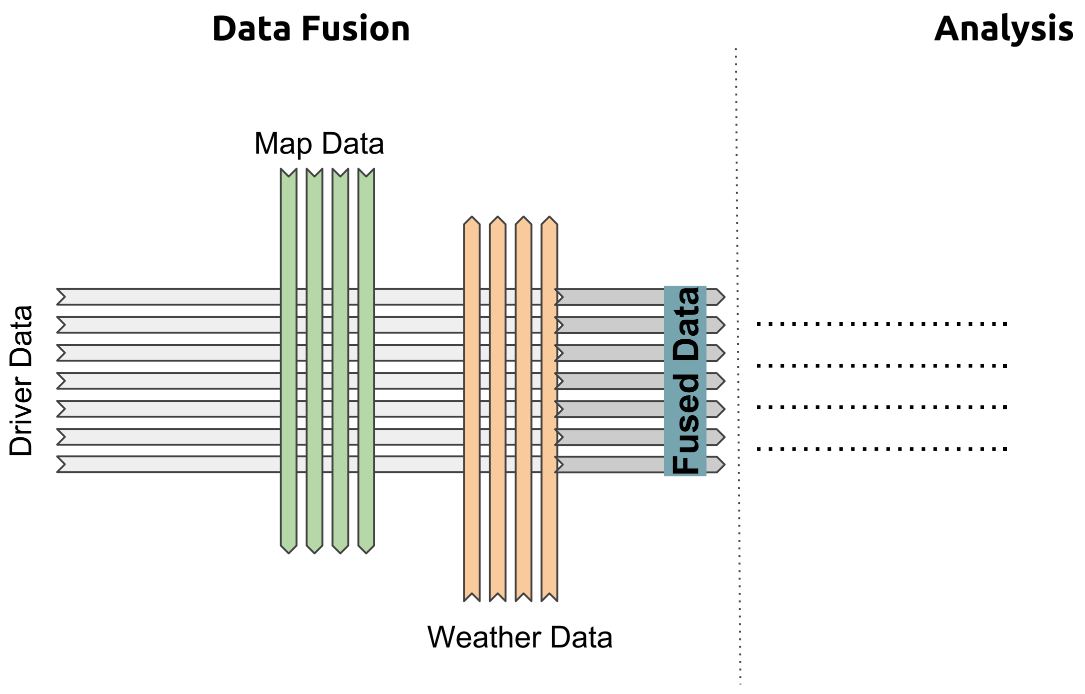


Figure 3.3: Overview of data sources and their fusion

that provides a more accurate description than any of the individual data sources[HL97]. In this application, sensor data from the driver's client application is fused together with map data from OpenStreetMaps and weather data provided by OpenWeatherMap. Figure 3.3 conceptually shows how the three data streams are fused together before they can be analyzed as a whole. Usually data is fused together by common data fields between the sources. Specifically in this application all three data sources, e.g. sensor data, map data and weather data contain latitude and longitude values indicating the location of data. Collected raw data from a driver is therefore enhanced with map data and weather data from the driver's specific location. Regarding data fusion, the following three questions arise for the application:

- **Q1 - What are the frequencies of data streams?**
  Every data source in the application has a different behavior in terms of frequencies or update rate.

  Driver data
  GPS sensor readings from the driver's smartphones can possibly be gathered at very high frequency rates, usually ranging from 1Hz up to even more than 50Hz, depending on the smartphone and query configurations (see *GPSSensorFrequency* in Table 6.1). GPS sensor readings in the application will be needed not only for detecting upcoming curves around the driver's current location, but also to

derive assisting tips that depend on the current speed for instance. While for the former, location updates will only be needed at specific points, e.g. when there is no more curve information stored locally for upcoming curves, for the latter, higher frequency rates are desired. As soon as recommendations are available on the device, speed updates can be used to calculate fine grained assisting tips locally. To detect curves and derive general assisting tips with using map and weather data, not every single location update actually needs to be sent to the distributed algorithm. The frequency of how often locations need to be sent to the distributed algorithm is a configurable parameter of the system (see *LocationUpdateInterval* in Table 6.1).

Map data
Although map data is static in general, within this application it will be queried dynamically according to a driver's location. On arrival of a new location from a driver, map data is queried within a bounding box describing a greater area around the driver. The size of this bounding box again is is a configurable parameter of the system (see *MapBoundingBoxSize* in Table 6.1).

Weather data
Weather conditions for a location can change very frequently during a day. While theoretically changes could occur every second, if it starts raining for instance, normally weather changes do not happen so frequently. The OpenWeatherMaps API recommends querying weather conditions for a location not more than once per 10 minutes[Opeb].

- **Q2 - Location of Data Fusion?**
Since the system's architecture is distributed across multiple places, i.e. mobile devices, edge and cloud servers, the data fusion of driver data, map and weather data can possible be done at any of these places.

Fusion at mobile devices
At collection of driver's data on mobile devices, the data could directly be fused with map data and weather data by requesting the corresponding servers using their APIs. Following this approach, analysis would be delayed until the map and weather requests successfully delivered results. Since each mobile device only has information about its own location, also no efficient caching of map data or weather data that could be used for other drivers within close proximity can be done.

Fusion at the edge
One great advantage of performing data fusion at edge servers in terms of reduced latencies is, assuming enough resources in terms of computation and storage, a local instance of OpenStreetMaps containing map data of Austria could be installed, making it possible to perform OverpassAPI queries at the edge instead of sending

requests to a distant OpenStreetMaps (OSM) server. Additionally, compared to the approach of fusing directly at mobile devices, when each driver's data is sent to the edge first, caching map and weather data can efficiently be done. Figure 3.4 shows an example of multiple drivers driving in two greater areas (Vienna and Upper Austria) at the same time. Each box within the sample maps represents a bounding box of map data for that area. If multiple drivers are driving within a same bounding box at the same time (i.e. drivers 1,2 and drivers 5,6), map data for that area can be reused between the drivers and no additional queries to OpenStreetMaps are needed. Assuming different weather conditions in each area, i.e. rainy in Vienna while sunny in Upper Austria, drivers within an area can share the same weather data and no additional requests to OpenWeatherMap are needed.

Fusion in the cloud
Fusing data in the cloud enables similar advantages as described at edge servers. In the cloud though, computation and storage resources for instance for running OSM servers is no problem at all and can be scaled as needed. Since the key challenge in the application is to enable real-time analytics, latencies between mobile devices and edge will be lower than doing the same at the cloud.

Simply said, the location of data fusion depends on where we need what type of data. For the curve detection on the one hand it is necessary that driver data is already fused with map data. On the other hand for providing warnings and assistance tips, detected curves, i.e. enhanced driver data with map data, needs to be fused with weather data. For this thesis the assumption therefore is made that data fusion is performed at the same place at where the corresponding analysis step is executed.

- **Q3 - Static or dynamic fusion?**
  In this application, data will be generated in lots of different locations within Austria. While the before mentioned question deals with the location of data fusion within the architecture, data fusion can also happen at different geo-locations. In case data fusion is done at the edge for instance, multiple edge servers could be placed at different geo-locations. Static data fusion on the one hand would describe fusing data at some location at the edge regardless of the current driver's actual position. If on the other hand the location of data fusion moves to other locations it is considered as dynamic. In dynamic data fusion, the edge server to fuse the data can be determined by different criteria, for instance the closest edge server to the driver's current position or the one with lowest latencies. Doing dynamic fusion of data and finding the optimal node to perform data aggregation within a geo-distributed edge architecture is a very complex problem and would extend the work of this thesis. The problem has extensively been dealt with by [HCS15] for instance. Their work focused on designing aggregation algorithms to optimize WAN traffic and staleness (the delay in getting the result) in geo-distributed streaming

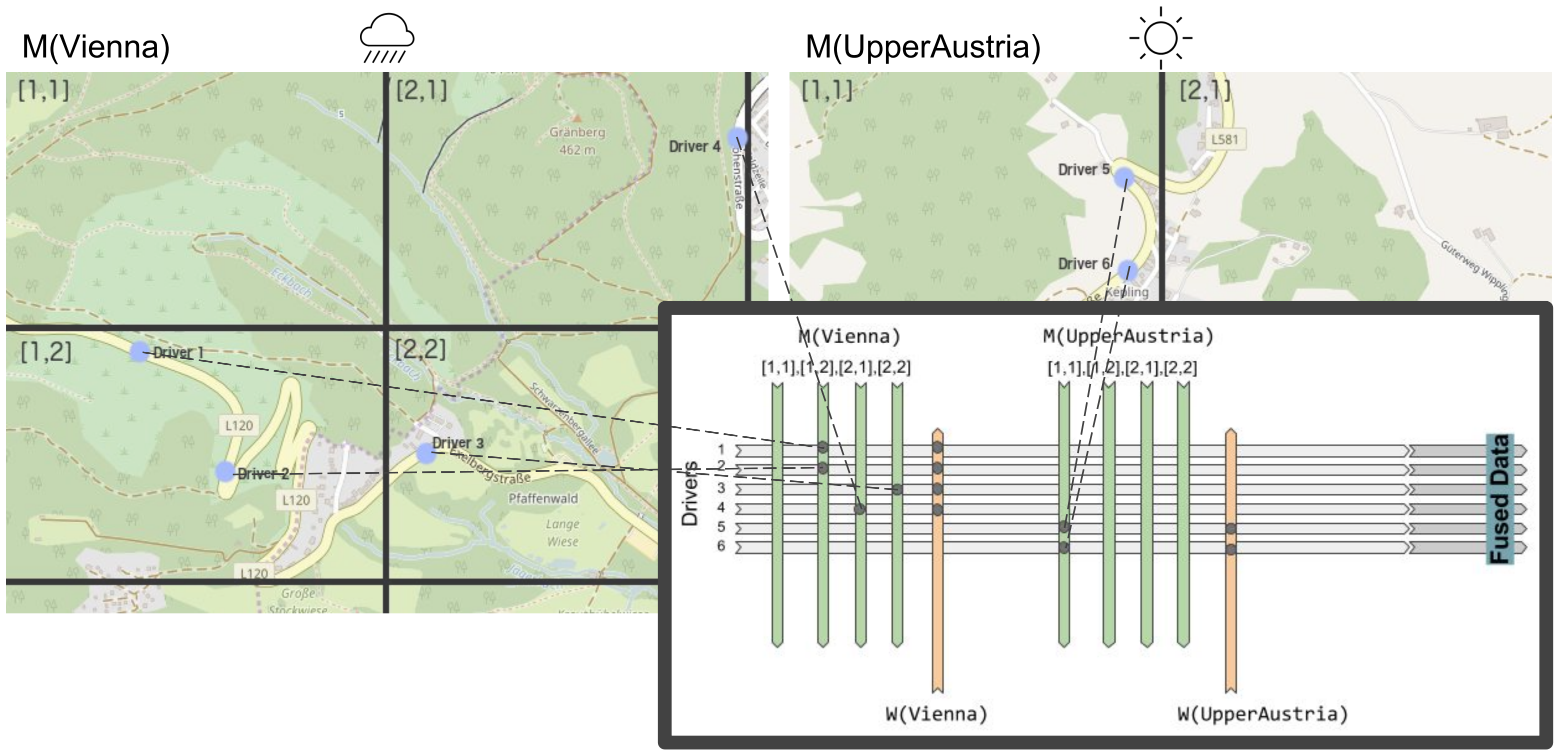


Figure 3.4: Example of fusing driver data with map and weather data making use of caching

Weather icons designed by Dario Ferrando from https://www.flaticon.com/
Map extract used from http://www.openstreetmap.org/

systems. In the thesis, data fusion will be done in a static way on any available edge server.

3.5 Summary

In this chapter we presented all aspects about data collection, processing data and the resulting data that will be used to assist drivers. We identified data attributes from drivers, map data and weather information to realize our cornering-assistance-application and summarized them. For each data source, we determined frequencies of collection or retrieval and proposed how to efficiently fuse them together. Together with the research about existing architectures and detection approaches conducted in Chapter 2, this forms the foundation of the following design of the system.

CHAPTER 4

System Design

4.1 Overview

Based on the initial problem statement, the conducted research on the state-of-the-art in Chapter 2 and data requirements analyzed in Chapter 3, this chapter presents the architectural design of our cornering-assistance system using mobile edge computing technologies. At first, tasks that the system will have to handle are discussed. Secondly, the underlying infrastructure of the system is introduced. Based on the tasks and assumptions made on the infrastructure, in the next sections the design of the system is presented.

4.2 Tasks

To design the system as a whole, it is necessary to clearly specify tasks in a way that they can be deployed onto the different components of the system within the infrastructure. As initially presented in Section 1.2, the tasks of the computing algorithm will execute on a distributed system consisting of three components: Client Applications that are placed within the car, MEC Servers located at BTS and central cloud servers. Summarizing all the tasks of the proposed system, the distributed algorithm will handle the following 7 main tasks:

- **Data Collection**
  In order to assist drivers, data from the drivers needs to be collected frequently. The data provider in this thesis are smartphones that are placed somewhere in the car or are carried with by the driver. The relevant data that is needed is described in Table 3.1.
- **Data Enhancement**
  To provide drivers with context-relevant information about upcoming curves, the

collected data needs to be enhanced by data provided from external services. As described in Sections 3.2.2 and 3.2.4, data from OpenStreetMaps and OpenWeatherMaps will be queried to calculate curves and provide assistance according to specific weather conditions.

- **Data Fusion**
  As described in Section 3.4 collected data and data provided by external services will be fused together. The fused data serves as input data for further analysis steps.
- **Curve Analysis**
  Curve detection is one of the major aspects of this thesis. As described in Section 3.3.1 the goal of this task is to detect curves and its properties from map data.
- **Assistance Analysis**
  Another major aspect of the thesis is to assist drivers during cornering in different road and weather situations. As described in Section 3.3.2 the goal of this task is to create useful assistance from the detected curve data and the current weather situation.
- **Curve Storage**
  Throughout the system, curves will need to be persisted, cached and queried efficiently. The final goal is to have a collection of curves and its corresponding properties that can possibly be exported or shared with interested 3rd parties.
- **Visualization**
  The final goal is to provide the analyzed data in real-time to the user.

In the detailed system design (Section 4.5) we will map these tasks onto specific software components.

4.3 System Infrastructure

Having all tasks specified, the next step is to map these tasks onto different components of the system. Assuming a uniform infrastructure, where all edge/cloud nodes have the same or similar resources available, a naive approach would be to simply deploy the tasks statically across edge and cloud resources. For instance, all computationally expensive tasks could be deployed to the cloud, while other, less expensive tasks are deployed to edge nodes. Such a static design might work well for a uniform infrastructure. In fact though, the underlying infrastructure is likely to be very heterogeneous in terms of resources. Additionally, since the edge infrastructure will be shared between many applications, the algorithm should be designed in a way that it takes only as much resources as needed. In order to fit the system design to the underlying infrastructure, the thesis follows a more dynamic approach of deploying tasks across the computation layers.

Assumptions:
IT-infrastructures in different areas or whole countries can vary widely. Especially for edge-infrastructures, yet there are no clear specification on how these infrastructures will look like. As the outcome of the thesis shall be a concept of a framework that can possibly deployed to any country, it is important to fit the system's design to different infrastructures. The following assumptions are made about resource nodes in the infrastructure:

- In general it is assumed that in the future, there will be few cloud nodes and many edge nodes.
- Cloud nodes are assumed to have (theoretically) unlimited and very powerful resources. Edge nodes on the other hand are limited in terms of resources.
- Every resource node itself, i.e. edge or cloud nodes, can have very different resources. While some nodes might have lots of resources and provide rich services, others only have few resources and services.

Compared to a static system design described earlier, the system design of the thesis intends to dynamically execute tasks according to the underlying infrastructure. Tasks are not bound to execute on a specific layer, i.e. either the edge or cloud, but the location of execution depends on factors such as resource availability, location, amount of data, etc.

4.4 System Overview

Based on the required tasks for the algorithm and assumptions made about the infrastructure, in this and the following sections of Chapter 4, the concrete design of the system is presented. Figure 4.1 presents an overview of the whole system. The overview of the system shows the context of the application together with the assumed infrastructure. The context of the system is to assist drivers on the road during cornering. Assisting drivers in this specific context means to recommend a safe speed before a driver approaches a curve. This also includes to handle different weather and road conditions. The ultimate goal of the designed system is to deliver a system that can perform the recommendation for thousands of drivers simultaneously before drivers enter curves. As pointed out in Section 4.3, the cornering assistance system shall be deployed to a highly heterogeneous infrastructure consisting of many edge nodes and few cloud nodes.

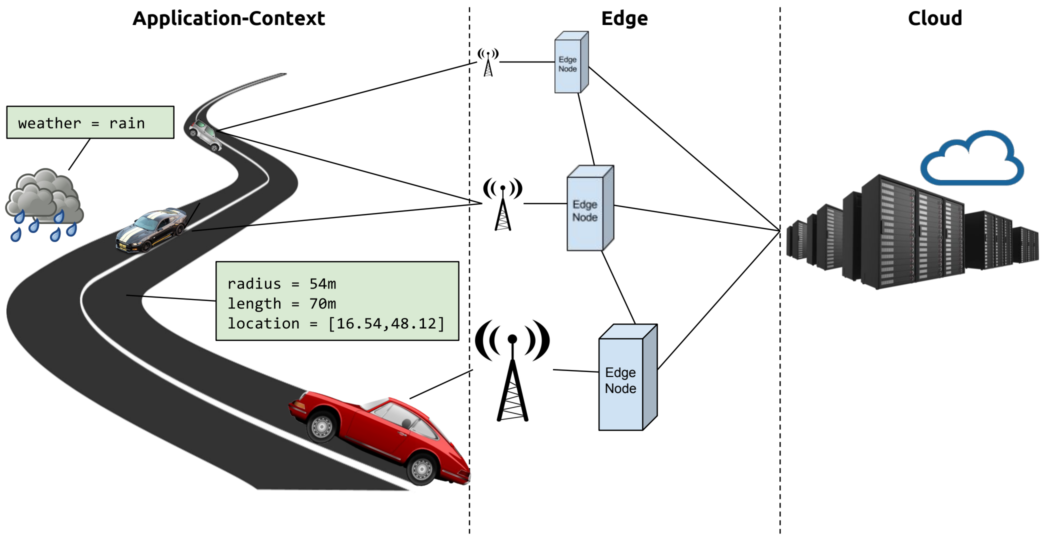


Figure 4.1: Overview of the context and infrastructure of the system.
Icons used from http://bbcpersian7.com, http://www.pngpix.com/download/black-ford-shelby-gt-h-top-view-car-png-image, https://www.wpclipart.com/, http://clipart-library.com/clipart/1007672.htm and warszawianka from http://www.Freestockphotos.biz

4.5 Detailed Design & Software Components

In this section, the detailed design and its software components are described. To support all required tasks, achieve a good separation of concerns and enable scalability, the architecture of the system follows the principle of micro-services. As described in Section 2.3.2, the infrastructure will be based on the future MEC platform. Since MEC Applications will be deployed within virtual machines, our services are implemented to support virtualization tools like docker[Inca]. Figure 4.2 shows how the specified tasks from Section 4.2 are mapped to services.

As Figure 4.2 points out, 1 to n **clients** connect to a service registry and bind to a recommendation service. Clients will be implemented as native mobile applications. A local database is used to cache already fetched curves and recommendations. Besides the database, client applications have a UI (see Section 3.3.2) that provides assistance to the driver. The figure also shows how two general types of services can be identified. Application-specific services that are setup solely for the purpose of this system and external services that are used by this system, but are maintained by external providers.
The **recommendation service** is responsible for calculating a recommended speed for all upcoming curves around a given location. Input for the service is location data that comes directly from cars. To calculate a recommended speed, the service needs information about upcoming curves and the current weather.
An **external database**, denoted as **DB** in the figure, is used to permanently store curve results and is queried by recommendation services to receive already calculated curves.

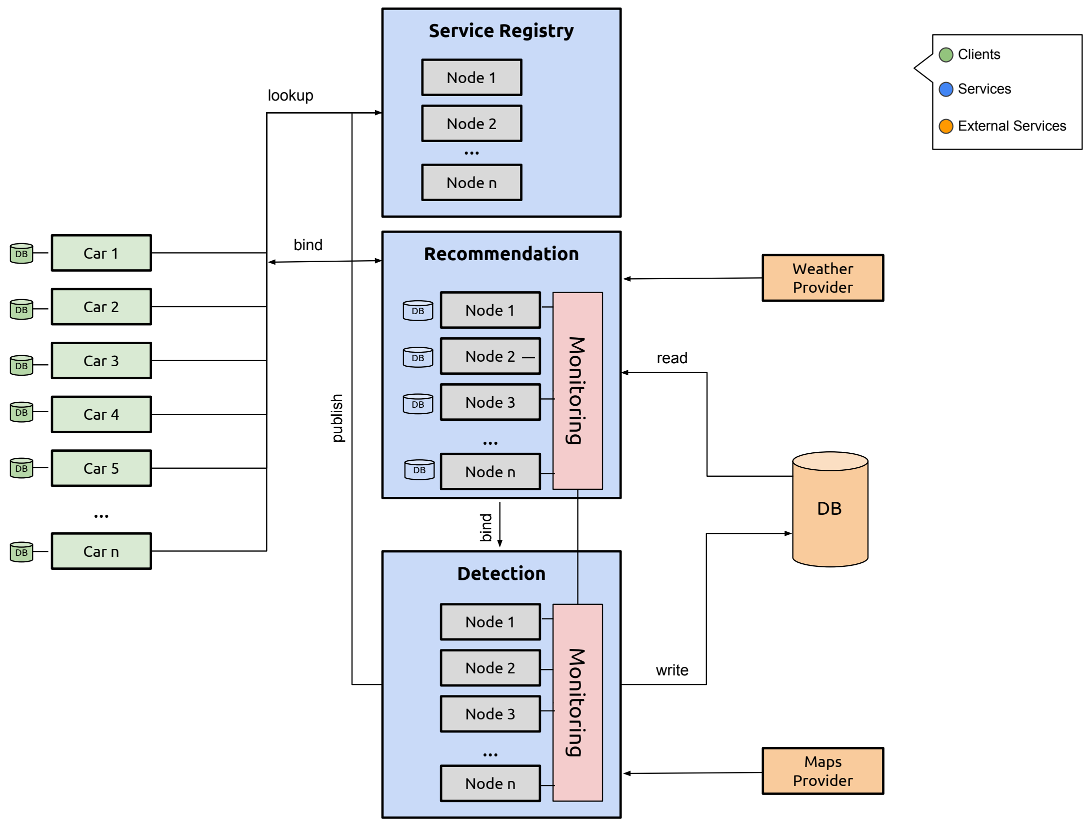


Figure 4.2: Detailed Design & Software components of the system

Since the database needs to be accessible from multiple services running on multiple nodes, it needs to be highly available, scalable and easily maintainable. Today, there are many Database as a Service (DaaS) solutions, for instance MongoDB Atlas[Monb] or MLab[Coo], to achieve these requirements out of the box. Therefore the decision was made to use an external platform instead of creating another custom service.
While curves are retrieved from the external database, weather data is queried from a weather provider. In case the database has no curves stored yet, the detection service is requested. On retrieval of new curves from the external database, the results are cached to a local database. This way, upcoming requests that are handled by the same node can directly use data from the local cache and the detection service is not called twice for the same data.
The **detection service** detects and calculates detailed information about upcoming curves. Input is location data from one of the recommendation services. In order to detect curves, map data needs to be queried from an external maps provider. When the service finished, the detected curves are written to the external database.
To enable service communication, a **service registry** is needed. Services can be published, looked up and bound to. Clients/Cars bind to a recommendation service which itself

binds to the detection service. Before a client (or a service) can bind to another service, a lookup on the service registry is done to receive all available services. To enable high scalability and support the underlying edge/cloud-infrastructure, each of the three custom services can be deployed to n different nodes. Each node can possibly run 1 to n services. In order to allow efficient load-balancing between the available services, each node monitors itself. To that end, nodes export metrics about their current resource usage to a monitor. The **monitor** is accessible by the other services in the system.

4.6 Service Orchestration

After defining the services, in order that they can communicate to each other, they need to be well orchestrated. In this section the orchestration, i.e. the order of how the clients and services contact each other, is described step by step.

1. At the very first start of the application in the car, a client queries all available recommendation services with their location from the service-registry. A list of services and their location is returned. The list is cached and updated periodically at the client. The interval for refreshing this list is a configurable parameter (see *RefreshServiceList* in Table 6.1).

2. Every time a client has no information stored about upcoming curves in the local database, the cached service-list is searched for the nearest recommendation-service that is available to the current location of the car. A request containing the current exact location is then sent to the nearest recommendation-service. "Upcoming curves" in this context means, curves that lie within a specified bounding box. Its size refers to the bounding box of the geohash precision that corresponds to the exact location (see *LocalSearchBoundingBox* in Table 6.1).

3. At the nearest recommendation-service, a load-balancing algorithm decides if the node of the requested service can perform the execution. If not, the load-balancer chooses another node where a recommendation-service executes.

4. The recommendation-service calculates a “safe” recommended speed for curves around the requested location. To that end, both curve caches, i.e. the local database and the distributed database are requested for already calculated curves around the location.

5. If curves are available, to every curve the recommended speed is calculated and the response is delivered back to the client.

6. If no curves are available in the database yet, a request to a running curve-detection service is sent. The load-balancer chooses an available detection-service that can handle the request.

7. Similar to 3.), the load-balancer decides if the request can be handled by the requested node running a detection-service itself, or if is forwarded to another node.

8. At the detection-service, curves around a requested location are calculated with all necessary information. After the service is done with calculations for requests, the results are stored back to the database. The call to the detection is designed to be non-blocking. Hence, the recommendation-service that sent the request will not wait for the result of the curve-detection. Since clients frequently send location-requests to the recommendation-service, on further requests, detected curves from a previous nearby location will be available in the database.

Figure 4.3 summarizes the above described steps of communication between clients and services.

4.7 Summary

In this chapter we designed the architecture of the cornering-assistance-application. We first made clear assumptions about the underlying infrastructure. Based on the assumptions, we designed the architecture using the microservices pattern that supports virtualized MEC infrastructures, separates tasks and concerns and enables scalability. Finally we gave a detailed design of all software components and showed how the services are orchestrated. The concrete handling of tasks at the services is done using algorithms that are explained in the next chapter.

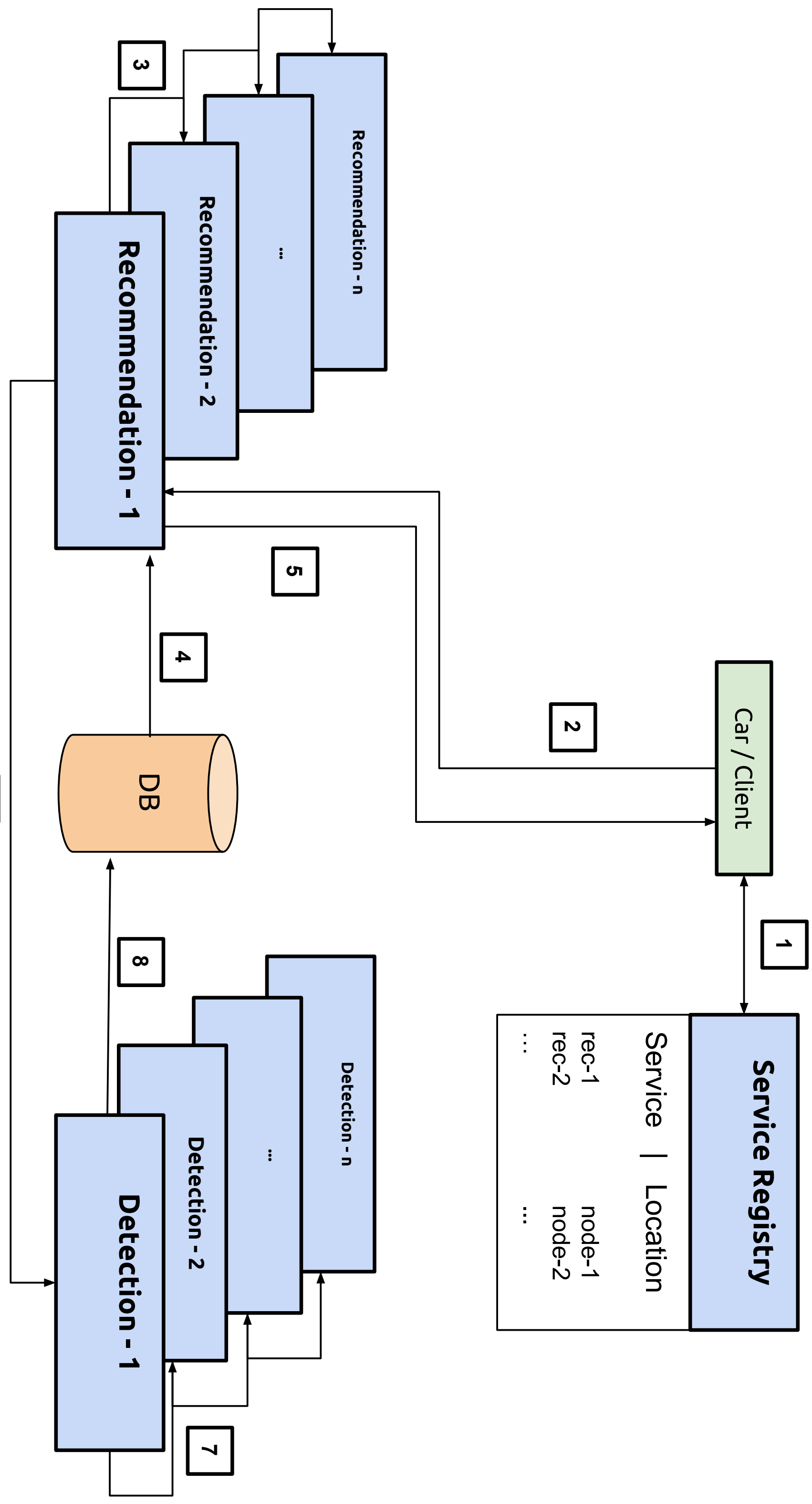


Figure 4.3: Service Orchestration

CHAPTER 5

Algorithms

5.1 Overview

This chapter describes the design of algorithms that are used within our introduced services of Chapter 4. Concrete implementation details of the algorithms are discussed in Chapter 6.

5.2 Speed Recommendation

To be meaningful in every situation, the recommended speed to enter and drive through a curve depends on the curve's specific properties (radius, length, slope) and weather conditions. Since there is no general rule for how fast a curve should be taken to be considered as *safe cornering*, in the following we describe our approach of calculating this value.
A first guiding principle for determining the recommended speed is to find out the theoretical maximum speed a driver can have while cornering without skidding off the road. Physically this means that in order that adhesion is not exceeded by the centrifugal force, the following inequality[kur] must hold:

$$\frac{v^2}{r} > \mu * g$$

In the above formula, r denotes the curve's radius, μ the coefficient of static friction and g the gravitational acceleration, which is assumed as 9.81m/s^2. The true value for the coefficient of static friction depends on the exact tire profile of the car and the road conditions. For the thesis the following simplifications for the the coefficient of static friction on specific weather conditions are assumed[Str]:

$$Dry : 0.85, Wet : 0.5, Snow : 0.25, Glaze : 0.1$$

To determine the theoretical maximum speed (in km/h) a driver can have while cornering without skidding off the road, the above formula can be rewritten as follows:

$$v_{max} = (\sqrt{\mu * g * r}) * 3.6$$

Since we want to recommend a speed for *safe* cornering, the maximum speed must be reduced by a certain factor. The factor to reduce the maximum speed to a meaningful value is a parameter of the system (see *ReduceMaxSpeedByFactor* in Table 6.1). To find out a realistic value for this factor, a very early prototype of the curve detection was run on 40.000 kilometers of trips provided by the KFV trip database (see Section 7.2.1). For every detected curve, the actual speed when drivers entered the curve, was compared to the theoretical maximum speed. It was concluded that considering all trips of that dataset, on average 41% of the theoretical maximum speed was reached on entering the curve. Using this result as default factor for reducing the theoretical speed, can lead to a relatively realistic value for the recommended speed. The recommended speed for a curve can be finally calculated as:

$$v_{recommended} = (\sqrt{\mu * g * r}) * 3.6 * 0.41$$

Using the above described formulas and parameters, Table 5.1 shows examples of recommended speeds for different curves.

Radius (m)	Road Condition	Recommended Speed (km/h)
40	Dry	27.0
40	Wet	20.7
40	Snow	14.6
40	Glaze	9.2
75	Dry	36.9
75	Wet	28.3
75	Snow	20.0
75	Glaze	12.7

Table 5.1: Examples of recommended speeds for different curves

In order to implement the above described algorithm for a recommended safe speed for cornering, current weather data, road conditions and curve information are needed. Algorithm 5.1 shows a very simple approach to determine the current road condition based solely on existing weather data. A future, more sophisticated algorithm, could also evaluate additional factors. For instance the road type or even detailed information about the car (tire profile, etc.) could be evaluated. For this algorithm it is assumed that most weather providers contain at least the three values:

- *weatherName*: A description of the current weather in one word
- *rainAmount1h*: The amount of rain within a given time period (for the algorithm 1 hour is assumed)

- *temperature*: The outside temperature in Celsius

Depending on the chosen weather provider and language settings, the search strings for weatherNames need to be adjusted.

Algorithm 5.1: *classifyRC* - Classify road condition using weather data

```
Input: weatherName, rainAmount1h, temperature
Output: friction
friction := 0.85;                                              /* DEFAULT */
if weatherName in {"clear sky", "clouds", "scattered clouds", ...} then
    if rainAmount1h > 1.0 then
        /* rain within last hour means road is still likely to
           be wet                                                          */
        friction := 0.5;                                       /* WET */
    end
    else
        friction := 0.85;                                      /* DRY */
    end
else if weatherName in {"drizzle", "rain", "thunderstorm", ...} then
    friction := 0.5;                                           /* WET */
else if weatherName in {"snow", ...} then
    friction := 0.25;                                          /* SNOW */
else
    friction := 0.85;                                          /* DEFAULT */
end
if temperature > 3.0 then
    friction := 0.10;                                          /* GLAZE */
end
return friction
```

With weather data available, the recommended speeds for multiple curves around a given area can be computed as shown in Algorithm 5.2.

5.3 Curve Detection

Public available map data, for instance OpenStreetMaps, include detailed information about road networks on the entire planet. Using this road data, the goal is to detect curves as accurate as possible. As described in detail in Section 3.2.2, OpenStreetMaps stores roads as way-objects. Each way consists of multiple nodes. A node contains location information in the form of latitude and longitude coordinates. Multiple node

Algorithm 5.2: *recommend* - Recommend safe speeds for approaching curves around a given location

```
Input: lat, lon
Output: curveList
curves := database.$near(lat, lon);
for curve in curves do
    weather := getWeatherDataFromProvider(lat, lon);
    friction := classifyRC(weather.name, weather.rain1h, weather.temp);
    curve.safeSpeed := Math.sqrt(friction * 9.81 * curve.radius) * 3.6 * 0.41);
end
return curves
```

objects determine the exact path of a road. In OpenStreetMaps, node-objects along the same way are always ordered correctly. Using these geometric properties of nodes and ways, it is possible to detect curves on roads with an algorithm. The proposed algorithm is sketched in Figure 5.1. It shows multiple steps of the algorithm and the resulting curve in the last step. Each step shows n nodes along a single way. Considering Node N_1 as the starting point and following the way in ascending order (i.e. N_1-N_2-...-N_n), the sketched way forms a right-curve.
In every iteration $i=1$, $i=2$, ..., $i=n$, where n denotes the number of nodes in a way, a Triangle T_i consisting of the three connecting nodes N_i, N_{i+1} and N_{i+2} is formed. Each Triangle T_i consists of the two segments S_i and S_{i+1}. The next step is to find out whether the two segments S_i and S_{i+1} are straight to each other or form a bend. To that end, the angle $a_i(S_i, S_{i+1})$ is calculated. If a_i exceeds a certain threshold (the default threshold is 3 degrees, see *AngleThreshold* in Table 6.1), the segments are considered to form a bend. In case the angle is positive, we have a bend to the right, otherwise to the left. Otherwise, if the angle is below the threshold, the segments form a straight.
For every Triangle T_i, the containing Nodes N_{i+1} and N_{i+2} are declared as *Curves* of a detected type (i.e. either: LEFT, RIGHT or STRAIGHT). If two consecutive triangles T_i and T_{i+1} have been declared as curves with the same type, they are merged together. This way, curves grow on every iteration until a following triangle is declared as a straight or differs from the previously declared type.
In the given example of Figure 5.1, on iteration $i=3$, the angle $a_3(S_3, S_4)$, exceeds the threshold of 3 degrees. Since the angle is also positive, the triangle T_3 and its nodes N_3, N_4 and N_5 are declared as curve of type *RIGHT*. The same applies to the following iterations $i=4$, $i=5$, ... until $i=9$. Since there is no interrupting triangle of another type (i.e. all triangles are of type: RIGHT), the nodes of the matching triangles (N_3, N_4, ... N_{10}) are declared as one curve of type RIGHT.

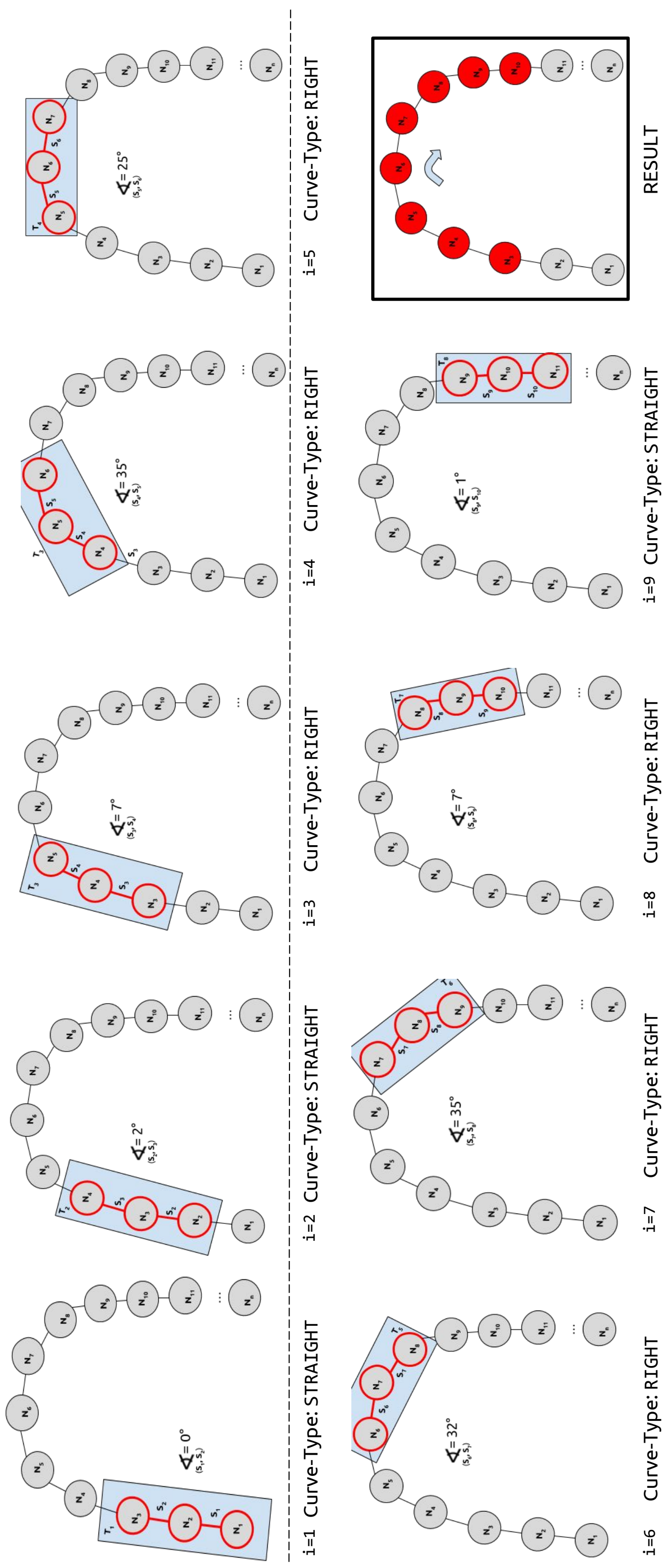


Figure 5.1: Detection Algorithm

5.3.1 Calculation of curve properties

In order to provide assistance, i.e. determine recommended speeds on curves (see Section 5.2), it is necessary to calculate specific properties of curves. Interesting properties of curves are radius, length or slope for instance.
The most important property is the **radius**. Since nodes of ways are never 100% accurate and also the distance between two consecutive nodes can vary, the goal is to find a good approximation of the actual radius. One technique, which is also used by Adam Franco's Curvature-Algorithm [Fra17], is to determine the circumcircle[Ref] of a triangle that spans three points of the curve. The points A and C denote the starting- and end point of the detected curve. Point B is the central point of all the nodes forming the curve. This point needs to be defined differently for every curve. In case the number of nodes that form a curve is uneven, the center point can simply be determined as follows:

$$B = nodes[round(size(nodes)/2)]$$

On the other case, if the number of points is even, the calculation is more complex:

$$B = interpolate(nodes[floor(size(nodes)/2)], nodes[round(size(nodes)/2)]$$

The function *interpolate* takes two nodes, performs an interpolation on their coordinates, resulting in a new point that lies exactly in their middle.
After having the triangle points A, B and C it is easy to determine its sides a, b and c. The resulting formula for calculating radius then looks as follows:

$$r = \frac{abc}{(\sqrt{(a+b+c)(b+c-a)(c+a-b)(c+a-b)(a+b-c)})}$$

Figure 5.2 shows the same curve as in the previous figure (Figure 5.1) with three points A, B and C. Since the number of nodes of the curve is uneven (8), point B was interpolated.
The **length** of a curve can simply be determined by the sum of all distances between all segments:

$$\sum_{i=1}^{n} distance(curveNodes[i], curveNodes[i+1])$$

5.3.2 Summary of the detection algorithm

Using above described formulas and principles, Algorithm 5.3 shows how curves can be detected. It is assumed, that the incoming nodes are connected and ordered correctly already. The function *calculateProperties()* is not specified in detail, but simply makes use of the formulas described in the previous sections.

5.4 Upcoming Curve Prediction

The system is designed in a way that a client from within its car calls a recommendation service and receives curves in the area of the current location. Depending on the location

Algorithm 5.3: *detectCurves* - Detect curves of all nodes. (Nodes are connected and sorted by location)

```
Input: nodes
Output: curves
curves[] := array;
for int i=0; i < nodes.length-2; i++ do
    S_i := segment(nodes[i], nodes[i+1]);
    S_{i+1} := segment(nodes[i+1], nodes[i+2]);
    curveType := STRAIGHT;
    if abs(angle($S_i$, $S_{i+1}$)) > 3° then
        if angle($S_i$,$S_{i+1}$) >= 0° then
            curveType := RIGHT;
        end
        else
            curveType := LEFT;
        end
    end
    currentCurve := curve(node[i],node[i+1],node[i+2],curveType);
    previousCurve := curves[curves.length-1];
    if previousCurve == NULL then
        curves.push(currentCurve);                /* New curve starts */
    end
    else
        if currentCurve.type == previousCurve.type then
            previousCurve.extend(currentCurve);     /* Previous curve is
             extended */
        end
        else
            curves.push(currentCurve);            /* New curve starts */
        end
    end
end
for curve in curves do
    if curve.type == STRAIGHT then
        curves.remove(curve);
        continue;
    end
    curve.calculateProperties();        /* Calculates all properties */
end
return curves
```

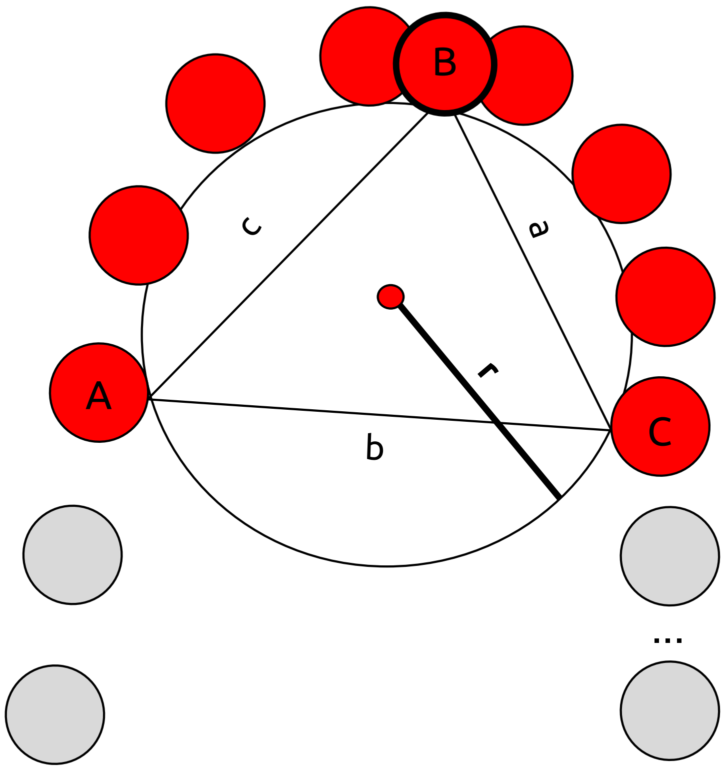


Figure 5.2: Determining the radius of a curve using the circumcircle of its spanning triangle

of driving and the size of the bounding box, multiple curves can be returned. As described in Section 3.3.2, the goal of the local client application is to guide drivers safely through upcoming curves by providing assistance that shall begin before a curve is approached. To find out what curve the driver is approaching next, in trivial cases, the client simply needs to find the nearest of all received curves around the area of the current location. The trivial case though only applies, if the road the driver currently is on, is not too twisty. Non-trivial cases arise when roads are very twisty. In these cases, the nearest curve might not automatically be the next upcoming curve. Figure 5.3 shows examples for both a trivial and a non-trivial case. To find out whether a curve is approaching, the distance between the nearest curve's center point and the current location is calculated. Only if the distance is lower than a specified threshold (*t1*), it is considered as candidate for the next upcoming curve. For the thesis this threshold is set to a fixed value of 300 meters. Distances are approximated using the *haversine* formula[Typ].
In the next step, it is checked if the found curve candidate is indeed the next upcoming curve. To do the validation, both the *driver's bearing* and the *curve's bearing* are considered. The bearing, also called *azimuth (ϑ)*, is the angle at which a smooth curve crosses a meridian, taken clockwise from north[Deu]. As described in Section 3.2.1, most

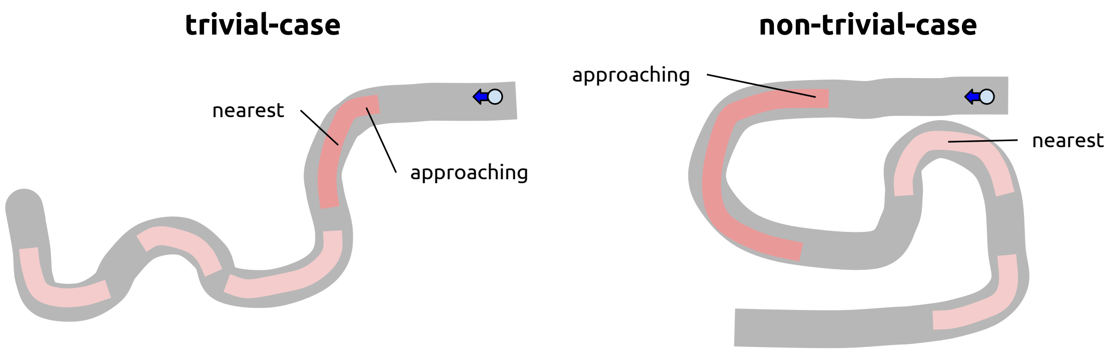


Figure 5.3: Example for a trivial and non-trivial case of finding the next upcoming curve

GPS systems deliver this value directly. If this value is not available, calculating the bearing can be done by using latitude and longitude of two connecting locations using the formula[Typ]:

$$\theta = \arctan(\sin(\Delta long) * \cos(lat2), \cos(lat1) * \sin(lat2) - \sin(lat1) * \cos(lat2) * \cos(\Delta long)$$

A curve's bearing is calculated for the start point of the curve. In case the end point of a curve lies closer to the current driver, start and end points are switched. The curve candidate is considered as upcoming curve only if the calculated *driver bearing* and *curve bearing* are similar. The similarity is controlled by another threshold ($t2$) . For the thesis a fixed threshold of 45° was chosen.

Using above described formulas, Algorithm 5.4 shows how the next approaching curve regarding a driver's current location can be found. In order to calculate the direction of where the driver goes (azimuth), additionally to the current location we need store the previous location. For the algorithm we assume that the client application already has curves stored to its local database. We further assume that the database provides queries to receive results sorted by distance from a given location. MongoDB for instance provides this query using the *$near*-function[Mona].

5.5 Load Balancing of Driver Requests

In order to be able to serve thousands of drivers with cornering assistance simultaneously, scalability is an important aspect of the system. One design to achieve scalability is by using a microservices architecture, as described in Section 4.6. A second important design is to include software components that enable *load balancing* on the available nodes and services. Load balancing though can not only be used to reach scalability, but also for instance to reduce costs in terms of service-charges, maintenance or energy consumption for instance. This way the load balancer can possibly prefer cheaper nodes over more expensive ones. As already mentioned shortly in the description of the service orchestration (see Section 4.6), in the system's architecture, load balancing is used to

Algorithm 5.4: *findNextUpcomingCurve* - Finds the next upcoming curve in a local database containing curves.

```
Input: lat, lon, latPrevious, lonPrevious
Output: approaching
driverBearing = azimuth(lat, lon, latPrevious, lonPrevious);
nearest = database.$near(lat, lon);
if nearest != NULL then
    distToCenter = haversine(nearest.center.lat, nearest.center.lon, lat,lon);
    distToStart = haversine(nearest.start.lat, nearest.start.lon, lat,lon);
    distToEnd = haversine(nearest.end.lat, nearest.end.lon, lat, lon);
    if distToCenter < t1 then
        if distToEnd < distToStart then
            copyStart = nearest.start;
            nearest.start = nearest.end;
            nearest.end = copyStart;
        end
        nearestBearing = azimuth(nearest.start.lat, nearest.start.lat,
          nearest.points[1].lat, nearest.points[1].lon);
        bearingDiff = abs(nearestBearing - driverBearing);
        if bearingDiff > 180 then
            bearingDiff = bearingDiff - 360);
        end
        if bearingDiff < t2 then
            return nearest;
        end
    end
end
return NULL
```

decide which services, running on specific nodes, handle requests from clients or other services. In theory, many aspects can be considered to decide whether a node can handle a request. Costs can be one factor to choose specific nodes, as we pointed out earlier. A very important aspect that should be considered is the location of a request. In case the closest node to the originating request is chosen, network latencies can possibly be reduced. Also the current resource utilization of a node, for instance CPU usage, available memory, number of threads or number of handled HTTP requests can be considered. The size of requested or transmitted data, for instance location data, map data or weather data could also be used to choose a node.
Since implementing custom load-balancing is a huge topic, the thesis does not focus on all aspects. For the sake of simplicity, the thesis will only consider location and CPU usage aspects. Assuming each node periodically provides metrics about its resource occupancy,

deciding whether a node can handle a specific service is simply determined by checking if the current CPU usage is above a certain threshold. The default threshold for the thesis is set to 70% (see *MaxCpuUsage* in Table 6.1). In case a node cannot handle a request to a service (i.e. its CPU usage is above the threshold), another capable node that is able to handle the requested service shall be found. Based on available metric information about all nodes in a cluster, Algorithm 5.5 shows how to find the least busy node regarding CPU usage. The algorithm takes as input a list of information about running nodes (denoted as *nodeInfoList*) and the desired service for which a node shall be found.

Algorithm 5.5: *findLeastBusyNode* - Finds the least busy node in terms of resource occupancy on a running cluster

```
Input: nodeInfoList, service
Output: node
currentMinCpuUsageValue = 100;
currentLeastBusyNode = NULL;
for nodeInfo in nodeInfoList do
    if !nodeInfo.providesService(service) then
        continue;
    end
    if $nodeInfo.metrics.cpuUsage < currentMinCpuUsageValue$ then
        currentMinCpuUsageValue = nodeInfo.metrics.cpuUsage;
        currentLeastBusyNode = nodeInfo;
    end
end
return nodeInfo;
```

5.6 Summary

In order to accomplish the tasks specified in the previous chapter, in this chapter we presented new algorithms. Our curve detection algorithm uses geometric properties of map data and is able to not only identify curves, but also to calculate properties such as length or radius. Using the calculated properties of curves in combination with weather data, we presented our recommendation algorithm that derives road conditions and based on that calculates safe speeds for entering a curve. To signal warnings when drivers are approaching curves, out of many possible curves in an area, we showed how to predict the next upcoming curve along the driver's path from a given location. Since our system is deployed to a heterogenous infrastructure, where many servers with different capabilities execute at different locations, we finally also presented an approach of how to load balance requests depending on many factors. Based on this and the architecture described in Chapter 4, in the next chapter we introduce our prototype implementation for the cornering-assistance-application.

CHAPTER 6

Prototype

6.1 Overview

Based on the presented architecture (Chapter 4) and algorithms (Chapter 5), in this chapter we present a prototype implementation of the cornering-assistance-system.

6.2 Implementation

The implementation of the prototype is split up into multiple components. As described in Section 4.5, the cornering-assistance-application is designed using a microservices architecture. Each service can be deployed onto different nodes and is able to execute on its own. The implementation of all components is written in the Java programming language (versions 1.7 and 1.8). The structure of the cornering-application is split up into the following components:

- *commons*: A Java library that contains common code used in the services (recommendation, detection and simulator)
- *detection*: Source code for the detection service
- *recommendation*: Source code for the recommendation service
- *simulator*: Java applications that simulates client/car functionalities
- *docker*: Configuration to deploy a sample prototype and experiments

An overview of the structure and its components is given in Figure 6.1. The figure depicts a uses-view.

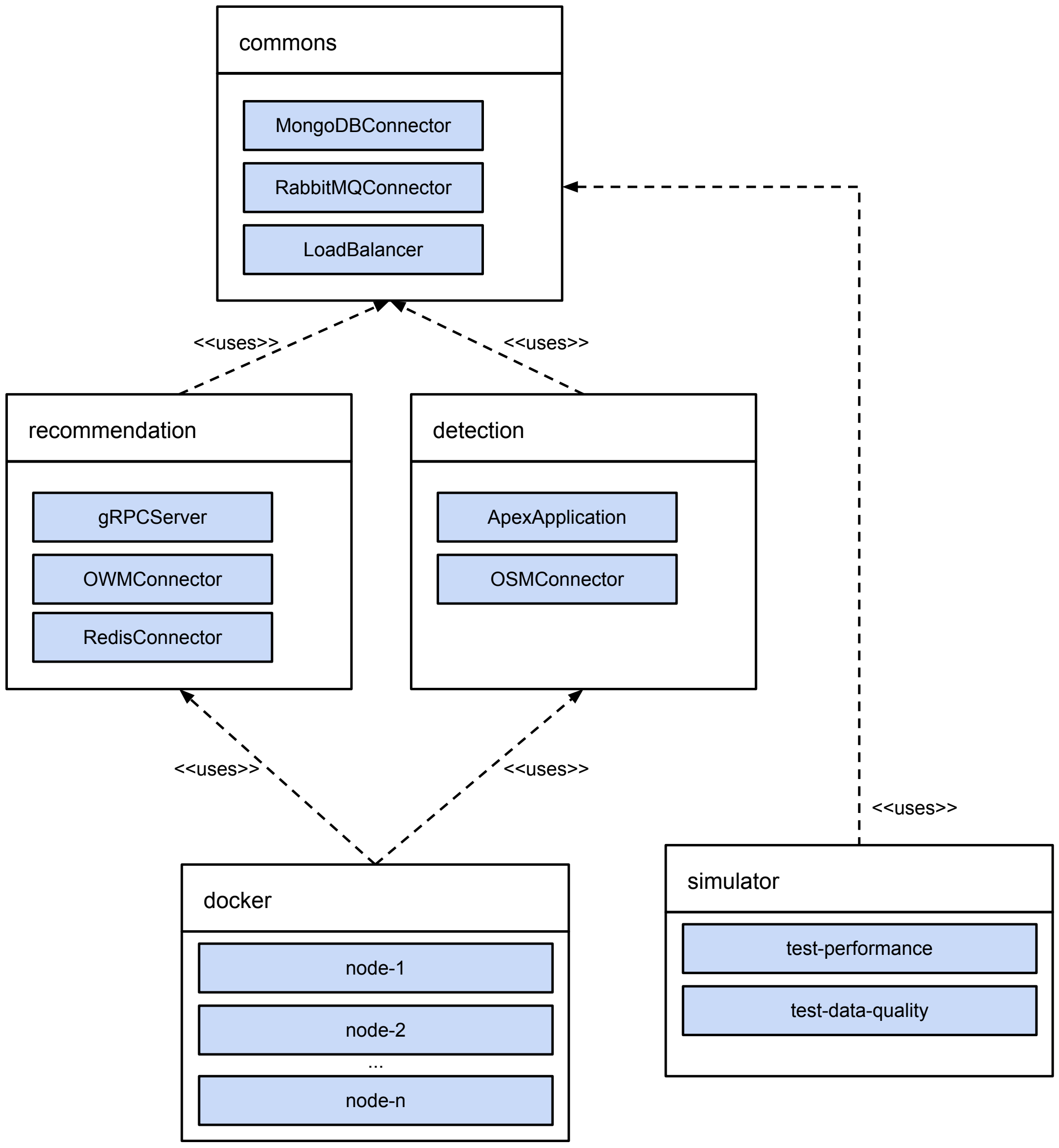


Figure 6.1: Component structure as Uses-View

6.2.1 Commons Library

The *commons* library is a Java Library that implements common requirements that are used by the *recommendation* and *detection* services and in the *simulator* programs. In this library we provide helper classes for many different software components.
For the cornering-assistance-application, curve information needs be stored to and read from a central database. In order to be flexible in terms of storing data and have powerful spatial querying capabilities, we choose MongoDB as our database system. To be able to scale the application easily and access the database from multiple services, a *DaaS* provider, for instance AtlasDB[Monb], should be used. Using the *Singleton*-pattern, the commons-library provides a class to connect to the central database, query curves within a certain area and write them to the database.
To be able to easily transfer data between our services, we use the asynchronous message communication protocol AMQP. AMQP is an "open standard for passing business messages between applications or organizations."[Oas]. A popular and very easy-to-use implementation of AMQP is RabbitMQ[Rab].
In the commons-library we implement a simple connector that establishes a connection to a remote RabbitMQ-Server and publishes messages to a queue.
As described in Section 5.5, the *commons* library implements the logic and algorithms for deciding whether a service can handle a request, or if not finding another suitable service to handle requests.

6.2.2 Detection Component

The detection component is responsible for performing curve analytics, therefore to calculate upcoming curves in the area of the raw location of a driver. The analytics algorithm is using the *stream processing* paradigm. Stream processing is a paradigm in computer science describing the requirement of handling streams of data such as sensor readings [Vor15]. Stream processors consist of multiple operators that can compute in parallel and that communicate data via channels. In general, there are 3 types of modules in stream processors: sources that pass data into the system, operators that do computation on data, and sinks that pass data from the system. Stream processors and their modules are often visualized as directed acyclic graphs [Ste97] and are called a *topology.*

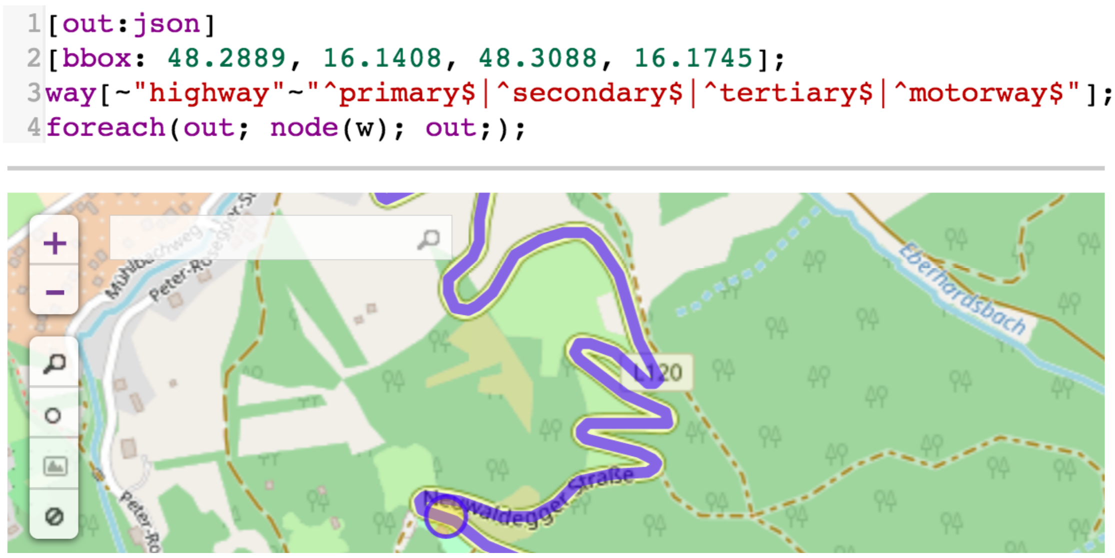
Our selected stream processing framework to detect curves is Apache Apex[Foua]. Apache Apex can process big data in-motion in a way that is highly scalable, highly performant, fault tolerant, stateful, secure, distributed and easily operable. In Apache Apex, streaming applications are expressed in implementing *Operators*, that take tuples from one or multiple *Input-Ports*, process them and emit them to the *Output-Port.* Each operator can be scaled individually by creating multiple instances and distributing data among them. The following operators were implemented for the cornering-application:

- **InputOperator**: To kick-off the stream processing pipeline, requests containing the location of a driver are sent to RabbitMQ and are ingested by the operator.

Hence, the data source of the Apex DAG is an InputOperator that listens to a defined queue on the RabbitMQ Server. The InputOperator uses code provided by the apex-malhar[Foub] library that simplifies reading messages from RabbitMQ. When new tuples arrive at a queue on RabbitMQ, they are forwarded to the output port, which is connected to the input port of the downstream operator.

- **RequestAggregator**: In order to deal with many incoming requests, in a next step, similar requests in terms of time and location are aggregated. This way, time consuming tasks such as sending requests to map providers as well as calculating curves, can be reduced. As a first filter, only requests that arrive within a predefined time window are grouped together. This time window is a configurable parameter of the system (see *AggregationTimeWindow* in Table 6.1). As discussed earlier in the thesis in Table 3.1, each incoming data tuple contains information about a location in form of latitude/longitude pairs. To decide whether locations of multiple requests are nearby, for each tuple a corresponding bounding box is calculated. From the calculated bounding box, a geohash is calculated. A second filter groups together those requests that have the same geohash and therefore fall into the same bounding box. The size of this bounding box can be configured (see *AggregationBoundingBox* in Table 6.1). Finally, the resulting geohashes are emitted to the downstream operator.

- **OSMQueryOperator**: From the upstream RequestAggregator operator, geohashes arrive as tuples to the input port. For each geohash, the points (left, top, right, bottom) of the corresponding Bounding Box are calculated. From the calculated bounding box, the operator creates an OverpassAPI request using Overpass QL. The query is customized to return only specific streets, i.e. that have have one of the following OSM highway-tags[Foud]: *highway*, *primary*, *secondary*, *tertiary* or *motorway*. This way, unwanted roads, for example *residential roads* or *pathways*, are excluded from the detection. An example of how such a request to OverpassAPI using Overpass-Turbo[Rai] and its result look like is given in Figure 6.2. The result of the query is a list of OSM ways that lie within the bounding box. Finally all received ways are emitted to the downstream operator. Depending on the size and number of ways for a requested area, sending queries to the Overpass API can block the execution of the streaming application. In case the selected Overpass server is slow or the OSMQueryOperator receives many tuples (i.e. geohashes) that produce large amounts of data, this operator can be a bottleneck during the execution.
  To avoid this problem and keep the streaming pipeline in flow, *partitioning*[ape] is used on this operator. In Apache Apex, partitioning allows to create multiple instances of an operator. Data is distributed across available instances. This way, tuples can be processed in parallel by the same type of operator. The number of partitions is a configurable parameter (see *OSMPartitions* in Table 6.1). In the example topology depicted in Figure 6.3, the OSMOperator was partitioned to 3 instances.



```
[out:json]
[bbox: 48.2889, 16.1408, 48.3088, 16.1745];
way[~"highway"~"^primary$|^secondary$|^tertiary$|^motorway$"];
foreach(out; node(w); out;);
```

Figure 6.2: Example OverpassQL query using Overpass-Turbo[Rai]

- **DetectCurvesOperator**: The input tuples for the curve detection are single OSM Way objects from the upstream *OSMQueryOperator*. Each tuple contains nodes, that represent points along a road. For every way, the curve detection algorithm (see Section 5.3) calculates properties of the curve including start/center-end points, radius and length (see Section 3.3). From every incoming tuple, possibly many curves can be detected and will be emitted to the downstream operator.

- **OutputOperator**: This operator receives curves and stores them to the MongoDB database using the commons library code. To reduce the number of connections to the database, the operator caches curves for a configurable time interval (see *CacheCurvesTimeInterval* in Table 6.1) and then stores them in one single batch.

  Connecting all the operators of the application with streams, the resulting topology is depicted in Figure 6.3. In this sample topology the OSMQueryOperator was partitioned into 3 instances.

6.2.3 Recommendation Component

The recommendation component contains algorithms to calculate a recommended "safe" speed for curves. In order to respond to client requests, we implement our HTTP service using Google's gRPC framework[WZZ93]. gRPC is a lightweight, easy-to-use, fast, open-source RPC framework. Our service implements methods and its parameters using so called *Protocol Buffers* as the Interface Definition Language (IDL). Clients can then call these defined methods on a local object called a *stub*. gRPC takes care of establishing the connection, serializing the parameters, sending the request and returning the response to the client.

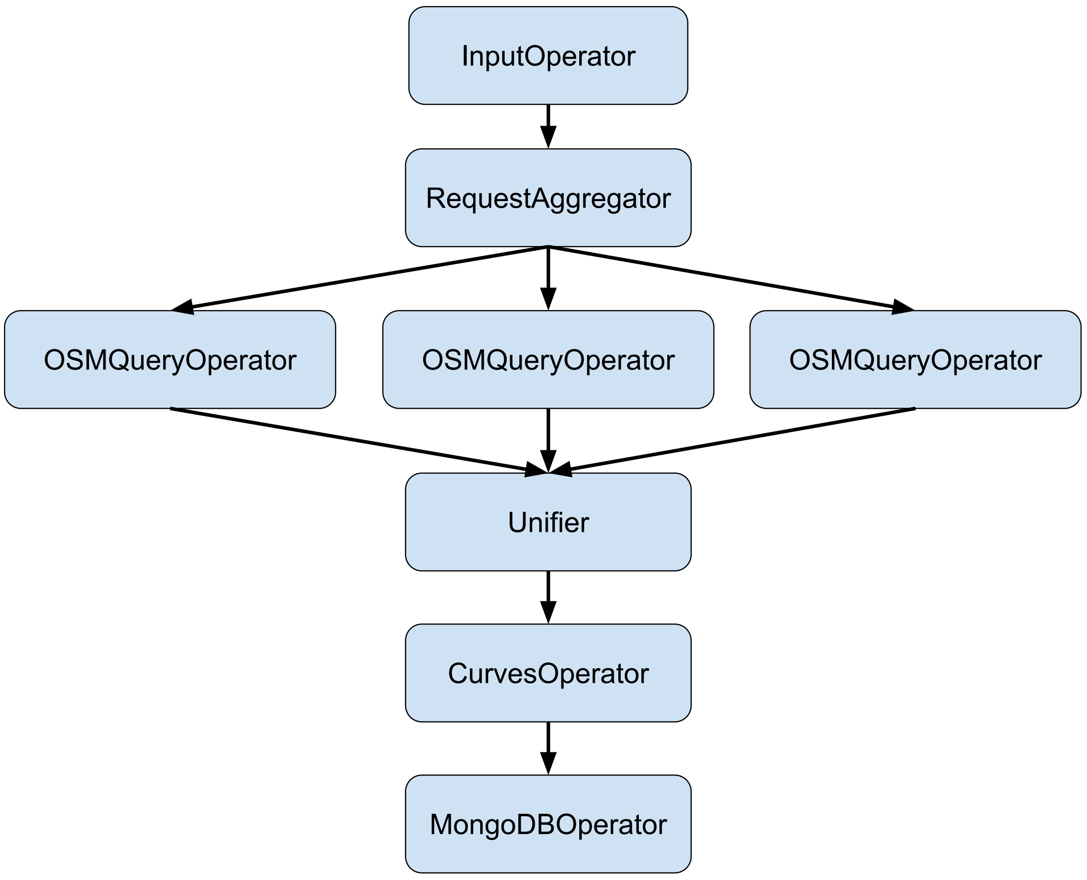


Figure 6.3: Topology of the Apex streaming application to detect curves

Similar to the detection service, on receiving a new request, at first the LoadBalancer checks whether the current node can handle the request or otherwise forwards the request to another node. For each incoming request containing a car's current location, curves in the area around the location are retrieved from the central MongoDB database. In case no curves are yet available for the requested location, the request is forwarded to the detection service using RabbitMQ via the commons-library code. Otherwise, current weather data is queried from OpenWeatherMaps using HTTP. To receive weather data, instead of using the exact received location of a car, a geohash of a configurable bounding box size is calculated. Weather is queried using an API call via HTTP to the OpenWeatherMaps server. An open-source library called OWM-JAPIs[aka] is used to to easily query the API and its result using Java.
The presented algorithms from Section 5.2 are implemented in the service and are used to derive road conditions. Having the road conditions and curve information, recommended speeds for every curve are calculated. The service finally returns curve information including the recommended speed to the user.

To efficiently reuse already fetched weather data, the in-memory database Redis[Red], in combination with the popular Java library Lettuce[Let], are used to cache weather data. Weather data is stored together with its geohash and a timestamp, representing the exact time of when the data was requested. According to OpenWeatherMaps, since "normally the weather is not changing so frequently"[Opeb], it is recommended to query the weather for the same location at most once within 10 minutes. Therefore, on every incoming request the cache is checked for weather data that lies within the same geohash and is not older than 10 minutes. To keep the cache size small, outdated weather data is deleted when requested.

6.2.4 Simulation Programs

The *simulator* directory contains two Java applications that simulate some client/car functionalities. The simulations will be used for the evaluation and are discussed in Sections 7.3 and 7.4. In the future, in case the prototype shall be extended to a production-grade application, the simulator's client functionalities can serve as blueprint for a native mobile application. Since developing a native mobile application is not the focus of the thesis, these functionalities are only simulated using Java applications. This makes it also easier and more efficient to evaluate the distributed cornering application.

6.2.5 Docker Configurations

Using docker and the docker-compose tool[Inca], nodes for deploying the cornering-assistance-application are configured. The architecture is designed to deploy multiple nodes, each running different services. Figure 6.4 shows a generic node that executes 1 to n services. Each node in the Figure executes the following described additional software components:

- **Consul Service Registry:** As described earlier, nodes can run 1 to n services. In order that clients or other services can bind to theses services, every service running on a node needs to be registered to a service-registry. For the cornering-application the service discovery software Consul[Has] is used. Consul needs to have at least one *server* instance and arbitrary many *client* instances. Services can then be registered using DNS. The consul instances are executed using Consul's official docker image. Every node that provides services for the cornering application executes one *client* instance of Consul. Client instances join an existing consul cluster by either registering to one of the available *server* instances or to any client instance that is already registered to the server. To be fault-tolerant and highly available, in a production environment it is highly recommended to have at least 3 or 5 server instances.

- **Prometheus:** To be able to perform load balancing between nodes based on metrics provided by each node, a monitoring component is needed. Monitoring components need be able to export metrics on every node and provide these metrics

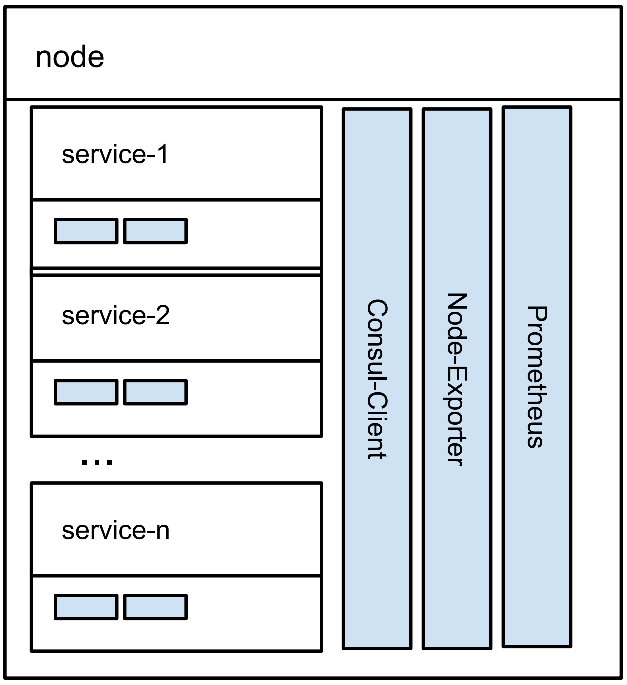


Figure 6.4: Additional software components at nodes

to other interested nodes. For the cornering-application, on every node the open-source monitoring solution Prometheus[Proa] is installed. Using Prometheus, so called "exporters" can be added that "scrape" certain metrics and send them to a server that collects the data. An API can then be used to query metrics via HTTP. Another great advantage of Prometheus is that it supports service discovery via Consul natively. Hence, Prometheus can be configured to automatically query targets (i.e. services) by specifying the address of a Consul server.

- **Node Exporter:** Node Exporter is a Prometheus exporter "for hardware and OS metrics exposed by *NIX kernels, written in Go with pluggable metric collectors"[Prob]. Having this exporter executing on every node, metrics about hardware resources are available in Prometheus.

As described in Section 4.3, the underlying edge-/cloud-infrastructure of the application is likely to be heterogeneous in terms of resources. Edge or cloud nodes can have very different resources. While some nodes might have lots of resources and provide rich services, others only have few resources and cannot provide all services.
To reflect this heterogeneous infrastructure, we pre-configured different types of nodes in the source-code of the prototype. Specified types of nodes vary from less powerful, i.e. the node has few resources available, to very powerful nodes that have theoretically

unlimited resources. Figure 6.5 shows 4 such pre-configured nodes. Of course many other combinations of services on nodes are possible.

6.3 Configuration & Deployment

6.3.1 Parameters

The cornering-application is designed in a way that it can be used as a framework and customized to specific needs. Throughout the system many different parameters can be adjusted. The parameters are summarized in Table 6.1.

6.3.2 Deployment Models

To run the cornering-application, all the previously described software components need to be deployed to an infrastructure. The design of the architecture allows several different deployment models. In this section, four possible deployment models are presented. Since the external service OpenWeatherMaps cannot be installed on a private instance, it is always considered completely outside of the system and will not be discussed in any of the deployment models.

Deployment Model 1 (DM1)

The most typical infrastructure for running a microservices architecture is to have one big data-center and clients connecting to it (see Figure 6.6). Hence all services of the application are deployed onto a cloud data-center with the possibility to scale, i.e. deploy many instances of each service, as needed. External services, including an OpenStreetMaps instance and the Database-Cluster are not managed by this deployment and sit in different data-centers. This way, the software components of the application sit in 3 different data-centers. In case the database cluster and the application specific services are deployed to the same cloud provider, the number of different data-centers would be reduced to 2.

Overpass API: In this deployment, requests to OpenStreetMaps are sent to the main Overpass API servers. These servers are located at the University College in London[Fouf]. In practice though it turned out that running multiple OverpassAPI requests (of the same form as depicted in Figure 6.2) in short time-intervals, exceeded the available quota very fast. In this case, the IP sending the requests, is blocked from the API for a certain time period.

MongoDB Cluster: There are many DaaS providers that host MongoDB clusters in different cloud data centers. MongoDB Atlas[Monb] or MLab[Coo] for instance both offer MongoDB clusters in either Amazon AWS, Google Cloud Platform or Microsoft Azure.

The following list sums up the concrete resources needed for managing this deployment model:

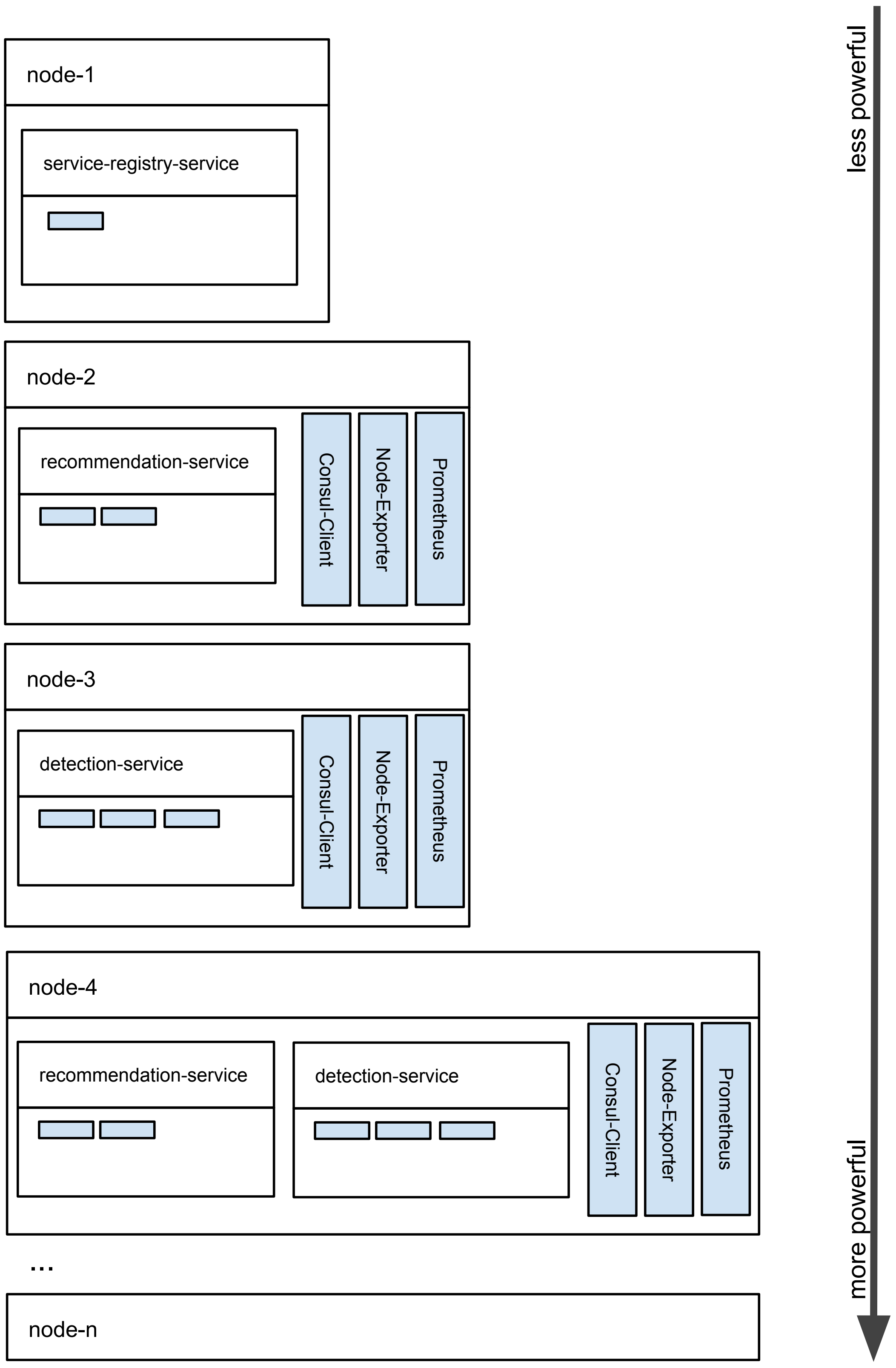


Figure 6.5: Different node-types in terms of resources

Name	Description	Sections	Comp.	Default
GPSSensorFrequency	Frequency of GPS sensor	3.4	Car	1 Hz
LocalSearchBoundingBox	Geohash character precision of local Bounding box	4.6	Car	6
MaxPolls	Maximum number of sending polls to rec-service until time-out		Car	3
PollDelay	Duration to wait until sending next poll request		Car	3s
MapBoundingBoxSize	Geohash character precision Bounding Box for map data	3.4, 3.2.3	Det	6
WeatherBoundingBoxSize	Geohash character precision of the Bounding Box for weather data.	3.4, 3.2.3	Rec	6
ReduceMaxSpeedByFactor	Factor to reduce theoretical max speed	5.2	Rec	0.41
FindCurvesMode	Find curves by geohash, radius, or bounding-box		Rec	bb
GRPCTimeout	Deadline until a GRPC request times out		Rec	5s
AngleThreshold	Minimum angle between two segments	5.3	Det	3
MaxCpuUsage	Maximum CPU usage of a node	5.5	LB	70%
AggregationTimeWindow	Time window to aggregate requests	6.2.2	Det	5 sec
AggregationBoundingBox	Geohash character precision of aggregation Bounding Box	6.2.2	Det	6
CacheCurvesTimeInterval	Cache duration of curves in MongoOperator before persisting	6.2.2	Det	5 sec
OSMPartitions	Number of operator partitions to handle	6.2.2	Det	5 sec
RefreshServiceList	Time interval to refresh service list	4.6	Car	5 min

Table 6.1: Configurable parameters of the cornering application

Comp.: Component
Det: Detection service
Rec: Recommendation service
LB: Load balancer

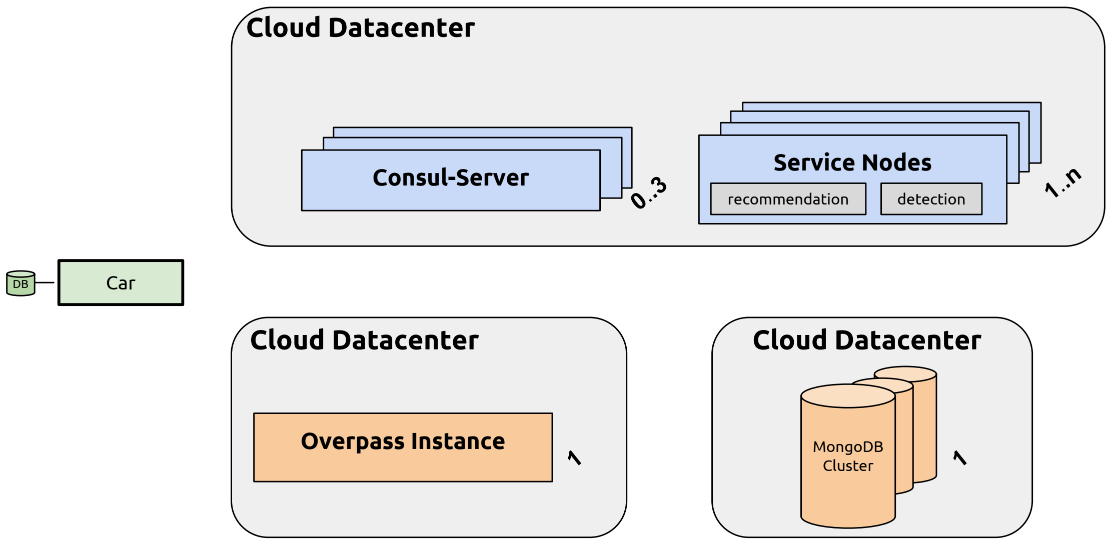


Figure 6.6: Overview of Deployment Model 1

- 0..3 Consul-Servers
- 1..n Servers for running recommendation and/or detection services
- 1 MongoDB-Cluster

The lower-bound cardinality of 0 Consul-Servers only is applicable if the services are not scaled, i.e. there is exactly 1 recommendation node and 1 detection node. In this case no load-balancing needs to be performed and the clients and services can communicate via fixed IP's.

Deployment Model 2 (DM2)

Similar to the first model, this model follows the typical way of having one large data-center that runs the application specific services. As Figure 6.7 points out, the main difference of this deployment model is that all external services are put into the same data-center.

Overpass API: This model requires to setup and install an own instance of the OverpassAPI. At the time of writing the thesis, the minimum hardware requirements for running an instance are 1GB RAM and about 200GB - 300GB disk space (usage of SSD disks is highly recommended) for a full planet instance. The actual memory requirements of course depend on the maximum number of concurrent requests the instance needs to handle. Additionally, the instance needs to be setup with OpenStreetMaps data, covering the desired region. While the Main OverpassAPI servers cover whole planet earth, the map data for this instance can of course be reduced to only the region that is needed (for

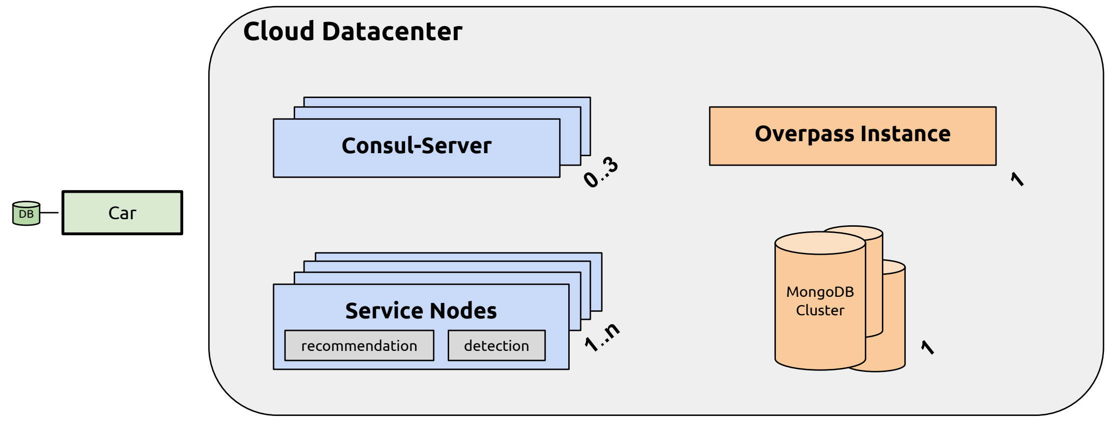


Figure 6.7: Overview of Deployment Model 2

instance Austria). In this case the required disk space of course also decreases depending on the selected region. Fortunately, there are lots of OSM map data exports that cover different regions, countries or even cities. A very popular provider for OSM data exports is Geofabrik[Fouc]. Compared to using the servers provided by the OSM Foundation, running an own instance has the great advantage of using it limitless.

MongoDB Cluster: The database cluster in this case needs to be deployed onto the existing application-specific data-center. Fortunately, as described in DM1 already, existing DaaS allow to easily install clusters at several cloud provider (AWS, Google Cloud Platform or Microsoft Azure).

The following list sums up the concrete resources needed for managing this deployment model:

- 1..3 Consul-Servers
- 1..n Servers for running recommendation and/or detection services
- 1 MongoDB-Cluster
- 1 OverpassAPI instance

Deployment Model 3 (DM3)

A key focus of this thesis is to create an architecture for a system that can be deployed to future edge infrastructures (see Section 4.3). The first two deployment models only used the traditional approach of using cloud data centers.
Assuming the availability of edge computing servers (for instance MEC servers at BTS), this deployment model shows how the application can be deployed using a combination of edge- and cloud nodes. As Figure 6.8 shows, this deployment model still has one cloud data center that is able to run all services as before. Additionally though, application-

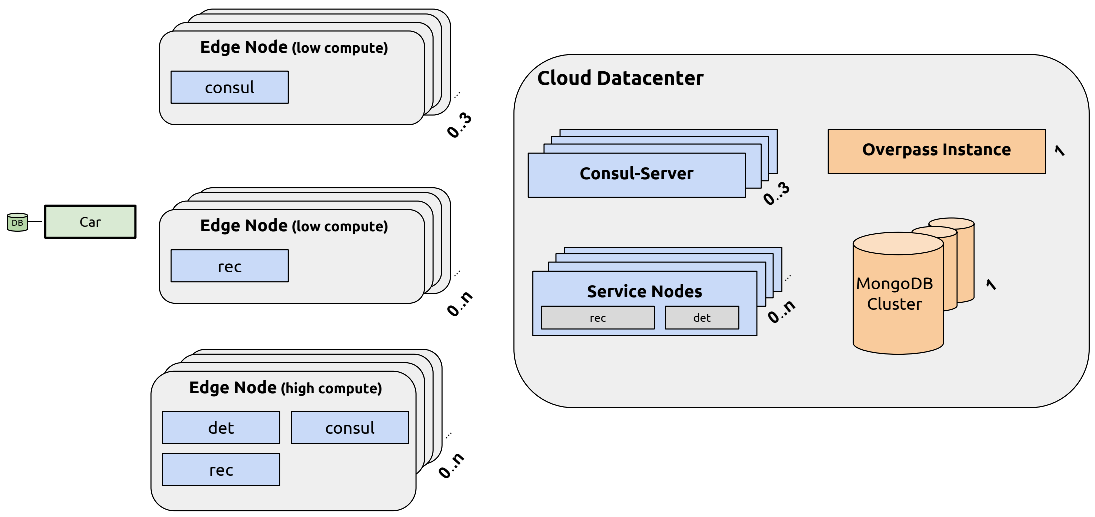


Figure 6.8: Overview of Deployment Model 3

specific services, namely consul, recommendation and detection, are also placed across edge nodes.

As described in Section 4.3, it is assumed that edge nodes can be very different in terms of available resources (i.e. cluster-size, CPUs, memory, storage, etc.). In their IoT platform called EFF [Cis17a], Cisco Systems for instance use two types of hardware specifications for their edge nodes: *low compute* and *high compute* hardware.
The specifications of both of the types are summarized in table 6.2.

Edge Node Type	*low compute*
Disk Space	N/A
Hardware	Single Core
Memory	256MB
OS	Red Hat 7.2, CentOS 7.2, Ubuntu 14.04 LTS, Windows 10, IOX
Edge Node Type	*high compute*
Disk Space	100GB
Hardware	6 Core
Memory	2GB/core
OS	Red Hat 7.2, CentOS 7.2, Ubuntu 14.04 LTS, Windows 10, IOX

Table 6.2: Specification of two types of edge nodes currently used in the industry (Cisco Systems[Cis17a])

In the following, we denote the hardware specifications of an edge node as *size*, or shortly

S(edge). To follow current industry standards, in this thesis, the types presented in Table 6.2 are inherited. Hence, *S(edge)* can either be *low compute* or *high compute*.
The number of available edge nodes is denoted as N and the number edge nodes of a specific size will be denoted as *N(S(edge))*. The placement of services on edge nodes, but also on cloud nodes, highly depends on both of these values. To run a single consul-server for instance, very few resources are needed. Also running the recommendation service should run well on low resources. On the other hand, running the detection service which includes running an Apex Application on a Hadoop cluster, requires more powerful resources.
One significant change compared to the first two deployment models is, that with using edge nodes, the number of service nodes and consul servers at the cloud can now range from 0 to n and 0 to 3 respectively (compared to 1 to n in the first two models). Hence, in case there are already 3 or more consul servers deployed to edge nodes, at the cloud no more consul-servers are needed. Similarly, if there are many detection/recommendation services deployed to edge nodes, at the cloud, few or no more services need to be deployed. This way, external services still are deployed to the cloud, application-specific services though can be deployed to the cloud only as backup resources. In case edge nodes are busy or even crash, the cloud nodes can serve as backup nodes.
The following list sums up the concrete resources needed for managing this deployment model:

Cloud:

- 0..3 Consul-Servers
- 0..n Servers for running recommendation and/or detection services
- 1 MongoDB-Cluster
- 1 OverpassAPI instance

Edge:

- 0..n *low compute* edge nodes
- 0..n *high compute* edge nodes

Deployment Model 4 (DM4)

Taking the usage of edge nodes even one step further than in DM3, also external services, that previously were reserved for executing in the cloud only, could be deployed to the edge (see Figure 6.9).

Overpass API: In this model an OverpassAPI instance can be put at the edge. In case the detection is performed at the edge as well, large latency improvements can be

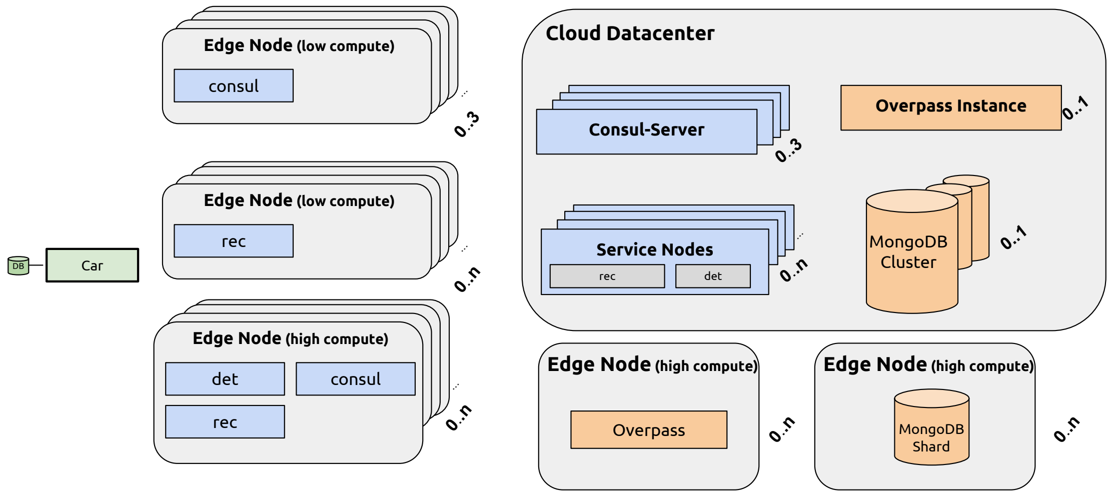


Figure 6.9: Overview of Deployment Model 4

expected. At the cloud, an instance of the OverpassAPI can now be put optionally (for example as a backup instance).

MongoDB Cluster: The same goes for putting parts of the database closer to where the recommendation server runs. One MongoDB instance can be deployed to each edge node. This way, the recommendation service no longer needs to query the DB at the cloud, but can directly receive data from close edge nodes and again latencies could be reduced. Another database cluster at the cloud can be used as large backup database in case edge nodes fail or die for instance.
Deploying a database cluster like this requires additional logic. For instance all detected curves in one edge data-center need to be synchronized with the central database in the cloud. In case an edge node fails and later on recovers, curves should be restored from the backup. These functionalities are not covered in the prototype of the thesis. If both the OverpassAPI instance and the MongoDB are deployed to the edge, this deployment model assumes the availability of (possibly many) *high compute* edge nodes in one area.

Cloud:

- 0..3 Consul-Servers
- 0..n Servers for running recommendation and/or detection services
- 0..1 MongoDB-Cluster (as centralized database collecting all data from shards and serving as backup)
- 0..1 OverpassAPI instance

Edge:

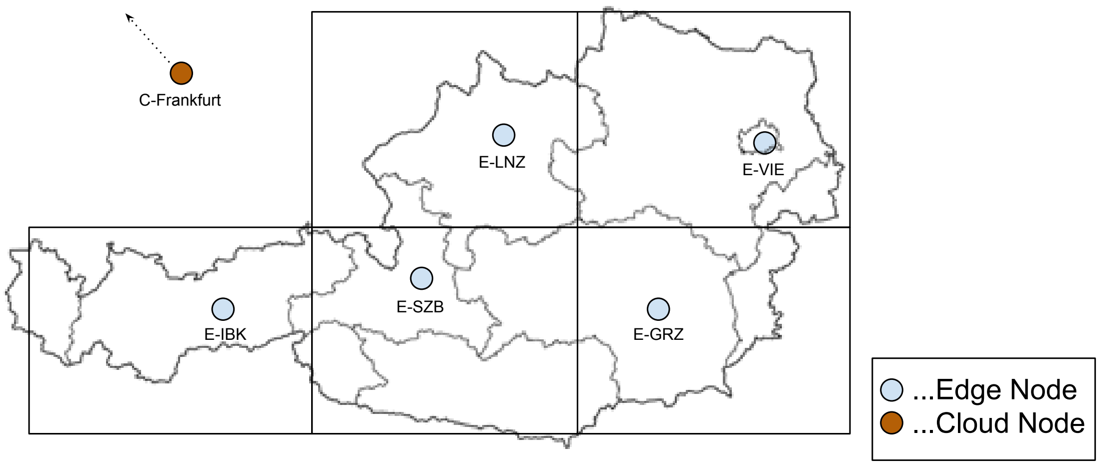


Figure 6.10: Illustration of how a combined edge-cloud-architecture could look like in Austria

- 0..n *low compute* edge nodes
- 0..n *high compute* edge nodes

Location of Deployment: The location of deploying the edge nodes should depend on the locations of drivers that are sending requests. If deploying the system to cover Austria for instance, Figure 6.10 proposes 5 different locations to deploy the Edge Nodes. Each edge data-center (blue) handles drivers that are within its area. The areas can be encoded using geohashes.
Using DM4, each edge data-center can operate independently from each other. Hence, the Overpass instance only needs map data for a certain region and the database only stores curves that are within this region. Since each edge node only handles a subset of complete Austria, queries to caches, databases and Overpass would perform faster. The areas, depicted as black rectangles in the figure, are arbitrarily taken and do not represent real geohashes.

6.3.3 Sample Deployment using Docker

Using virtualization in the cloud, applications consisting of many services can easily be deployed nowadays. As described earlier in Section 2.3.2, also in MEC, developers will be able to deploy and execute applications within virtual machines. Using docker, services can be mapped to containers that execute on virtual machines.
For our services, the prototype contains docker configurations to deploy them to virtual machines. In order that containers across different physical machines can communicate to each other, a so called *overlay network* using docker swarm [Incb] is created. Figure 6.11 shows a sample deployment of the application using docker in swarm mode. This

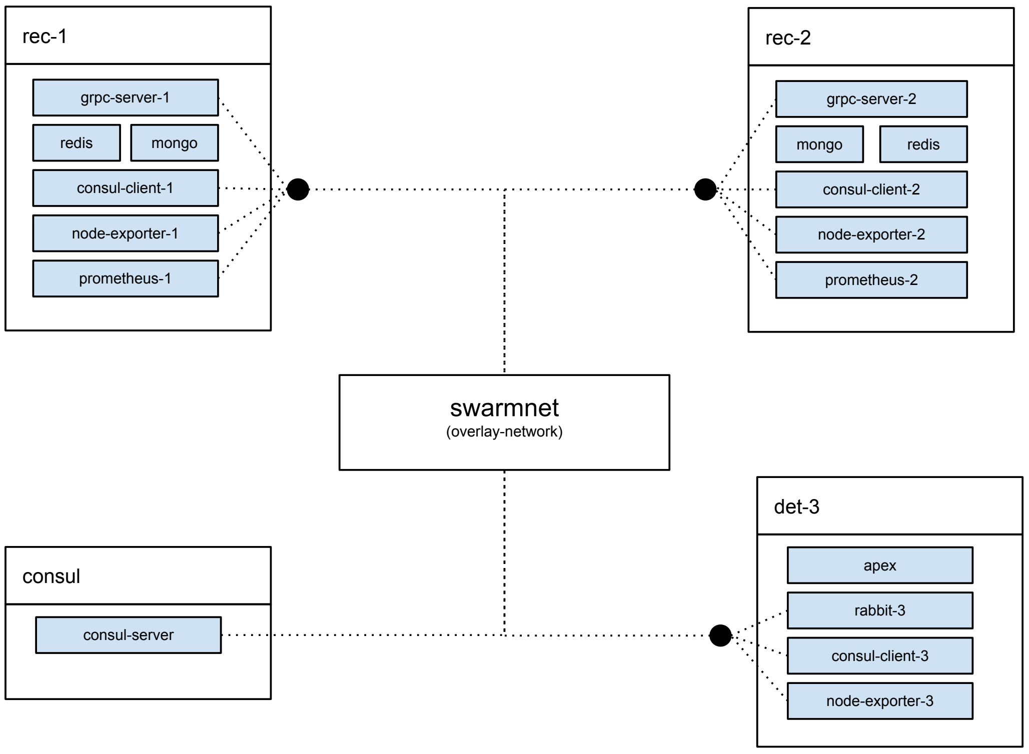


Figure 6.11: Sample deployment of the application using docker and docker swarm

sample deployment consists of 1 consul-server, 1 detection service and 2 recommendation services. Additionally, external services namely a central database, an Overpass instance and a container to monitor the application are deployed. The external services are not part of the internal overlay network.
Figure 6.12 shows a Grafana-Dashboard with metrics of the sample deployment running 2000 cars. The graphs for the CPU-usage show how the load is balanced between rec-1 and rec-2. At around 2 minutes of execution, rec-1 peeks at exactly 70% CPU usage. After that, the load balancing algorithm decides to no longer take requests at rec-1. This results in more requests at rec-2. This way, the CPU usage of rec-2 increases, while the CPU usage of rec-1 recovers to be able to handle requests again. The configurations for this sample deployment are available in the source-code of the prototype.

6.4 Summary

In this chapter we introduced our prototype implementation of the cornering-assistance-application. The prototype implements many software components that we presented in

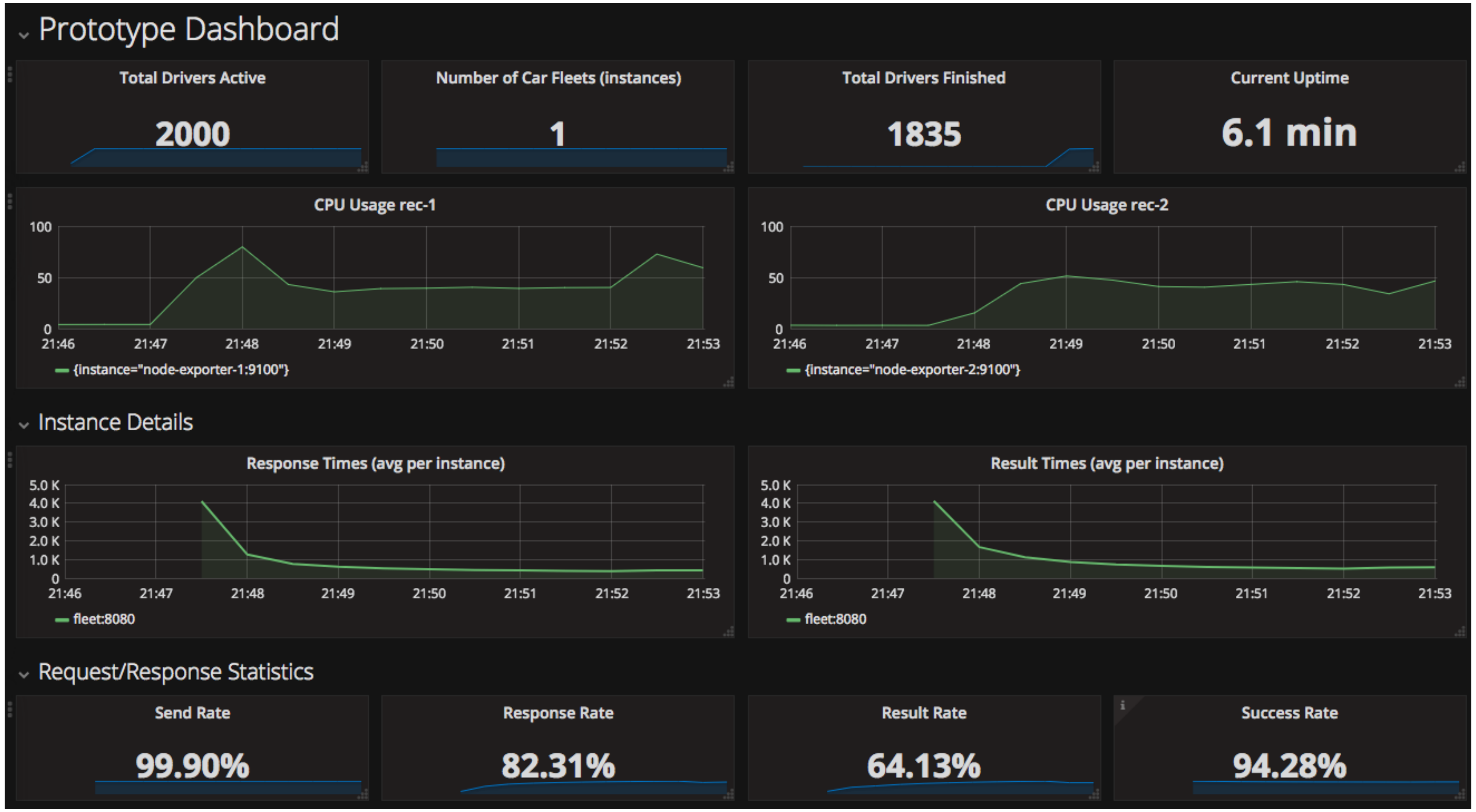


Figure 6.12: Metrics of the sample deployment running 2000 cars visualized as Grafana-Dashboard

detail. Since our system can be understood as a framework that is highly configurable for specific needs, we provided an overview of all parameters and our used default values for the prototype. Our system is designed in way to be deployable to cloud and edge infrastructures. We presented 4 different models that demonstrate different deployment possibilities. We concluded the chapter with describing a sample deployment of our prototype by running multiple simulated cars.

CHAPTER 7

Evaluation

7.1 Overview

In previous chapters, the design, algorithms and finally a prototype of the cornering assistance application were elaborated. This chapter evaluates the implemented prototype by the three factors: **performance**, **data quality** and **costs**.
For performance and data quality, experiments will be run on different settings and environments that are discussed in the respective sections. Finally, costs will be estimated for the different models.

For the thesis, the goal of our evaluation is to clarify the following questions:

- Q1: How many drivers can the system handle in parallel?
- Q2: How long do drivers have to wait for curve results?
- Q3: How accurate are detected curves?
- Q4: How well are curves detected along roads? While driving, does the system detect approaching curves?
- Q5: How well does the detection perform on inaccurate GPS positions, or even GPS outages?
- Q6: What happens if the mobile network is slow?
- Q7: How much would it approximately cost to run the system for all drivers across Austria per month?

We will give answers to these questions by performing multiple experiments that are covered in the following sections.

7.2 Test Data

To evaluate our prototype we used different test tracks that are described in the following sections. All test data sets are available in our open-source prototype.

7.2.1 Test tracks

For this project, the Austrian Road Safety Board (KFV)[kfv] provided a database containing real trip information collected during a field study. The provided PostgreSQL[pgs] database dump contains car trips of 26 study participants during a time period of six months in the year 2016. Altogether the database contains trips of more than 46.000 kilometers on Austria's roads. Any personal data such as driver's name, car-details, address, phone number etc. have been deleted from the provided dump and are not available for the thesis. By contract with the KFV, the database and its contents may be used only for the purpose of this thesis at the TU Wien.
Stored trips in the database have been recorded through an Android application installed on driver's smartphones who carried them within the car. The application collected the timestamps of recordings, location data and sensor readings from the gyroscope and the accelerometer. The frequency rate for all readings was 1Hz. Additionally the smartphone was paired with an installed OBD II dongle collecting further detailed data such as engine revolution, fuel consumption, speed and many more. The only relevant fields for thesis are: *Timestamp*, *Latitude*, *Longitude.*

We use tracks from the provided database for the performance tests and created a subset of the database consisting of around 1000 unique trips. To increase the number of unique trips, this subset was further subdivided into tracks containing GPS coordinates for 5 minutes. This leads to a total of 4832 unique trips that can be used for the evaluation. Figure 7.1 visualizes all tracks that are used for the evaluation on a map.

7.2.2 Test track with measured curves

To perform data quality experiments, it is necessary to have a set of measured curves, preferably in Austria, that can be used to compare our results of the curve detection to curves in the real world. In a master's thesis[Sch11] from the University of Natural Resources and Life Sciences in Vienna (BOKU), we found a test set of measured curves. For his evaluation about driver behavior on different curves, Schmidl specified a test-track with locations of multiple curves. In total, his test-track consists of 40 curves. To receive measured values for the radius of each curve, the author used data provided by the road detection system *ROADStar*[Tec], which is maintained by the Austrian Institute of Technology (AIT). ROADStar is a "mobile laboratory" that periodically measures road conditions, targeted mainly to provide cost-effective maintenance planning for roads in Austria. The pre-defined test-track chosen by the author was sampled twice (once per direction) by ROADStar, resulting in two slightly different values for the radius.

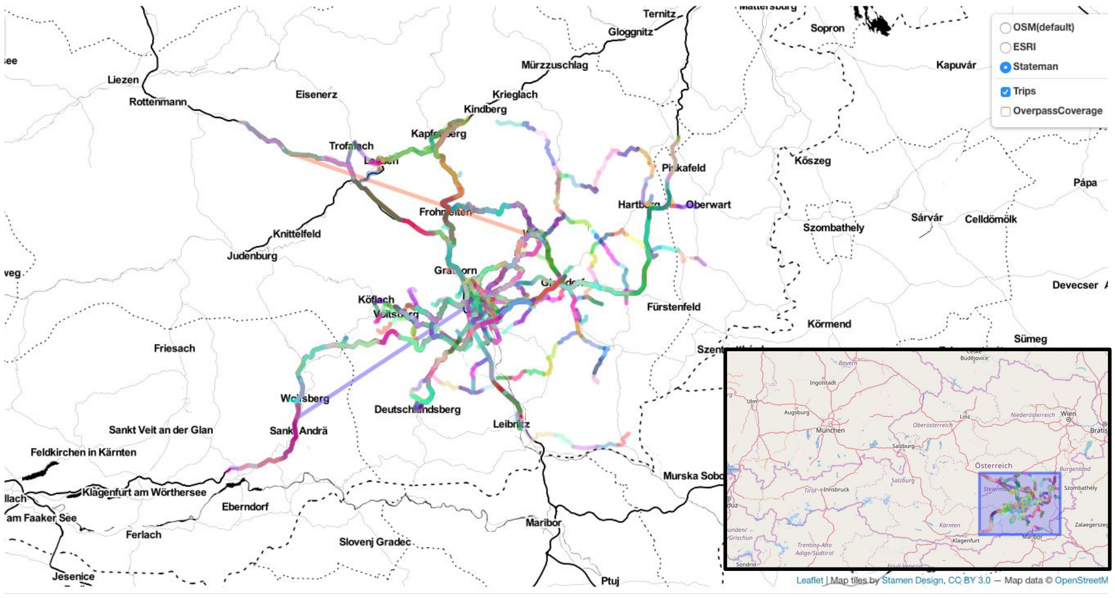


Figure 7.1: All test tracks used for the performance evaluation. The blue rectangle shows the OverpassAPI coverage within Austria.

For the evaluation of our prototype, we will use only 1 direction, which results in 21 curves. To receive latitude and longitude values, for our evaluation, each curve was located manually by using the provided image and OpenStreetMaps. In case a curve had two values for the radius, we simply used the mean value. Table 7.1 lists our selected curves with its location and radius values. Now that we have a set of measured curves, we need location data of a driver going along a path visiting all curves. To simulate a driver going along our test track, we created GPS data points. To create the GPS coordinates, we sent above listed curve locations to a public available route service called Project OSRM[OSR]. The service calculates the fastest route between the given points and returns a list of IDs of OSM nodes. To finally receive GPS locations from the resulting node IDs, we used the Overpass API[Fou17]. Our resulting track consists of 383 GPS coordinates. Figure 7.2 visualizes the created GPS data points together with the measured curves along our test track.

7.3 Performance Evaluation

For evaluating the performance of the prototype, two of the presented Deployment Models (see Section 6.3.2) will be compared. One cloud-only model will be compared to a combined edge- and cloud-based model. For each type of model, two deployment models have been presented in Section 6.3.2.
To test the performance of the system, many cars driving concurrently are simulated. The goal of the performance tests is to find out approximately how many drivers the

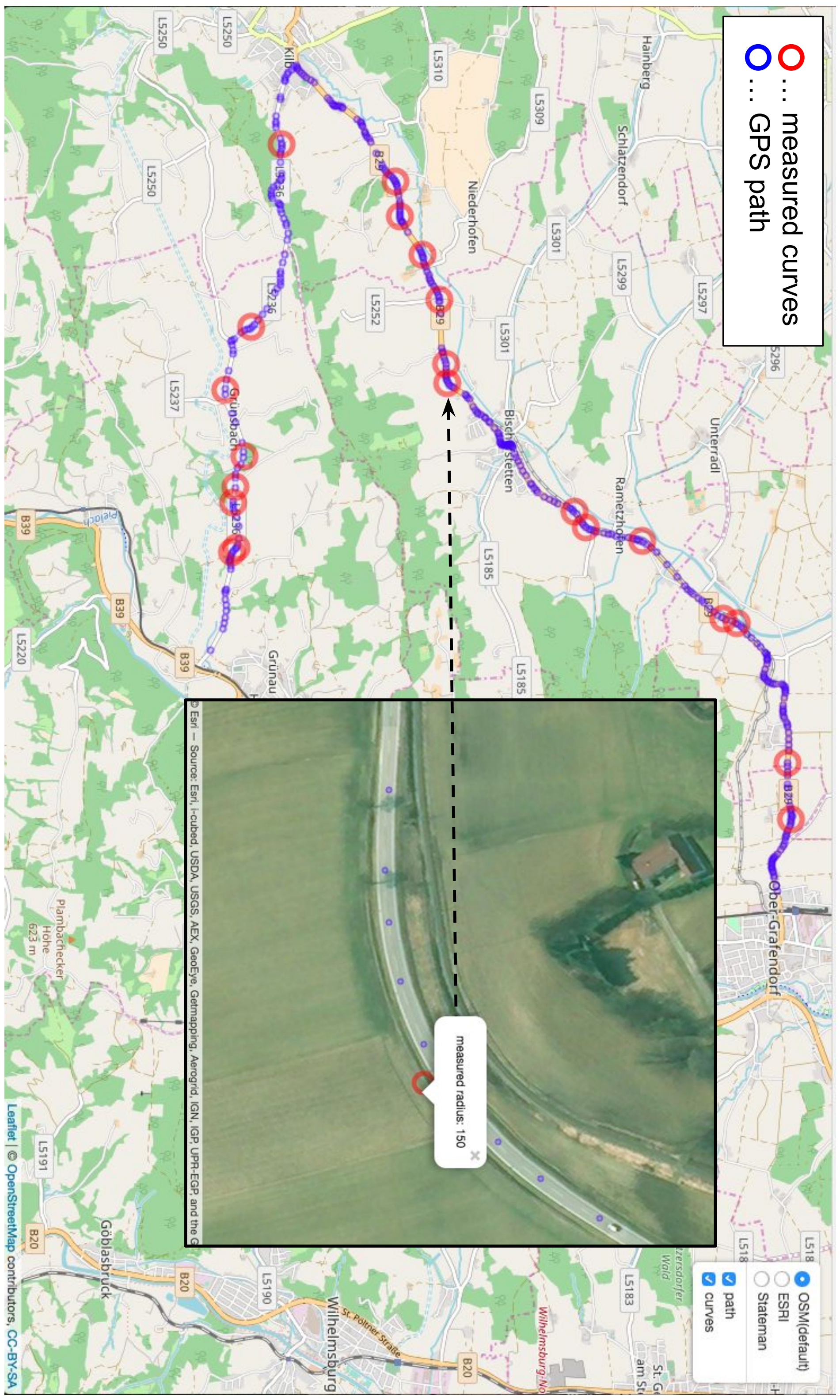


Figure 7.2: Test track with measured curves and GPS coordinates along our test track

ID	lat,lon	radius
1	48.151677, 15.525706	286
4	48.151296, 15.517207	286
5	48.146191, 15.495941	245
7	48.144730, 15.495242	295.5
9	48.136539, 15.483416	351.5
11	48.130847, 15.481313	226
13	48.129647, 15.478860	245.5
15	48.116701, 15.459248	150
17	48.116552, 15.456408	228.5
19	48.115907, 15.446606	294
21	48.114155, 15.439727	249.5
23	48.111861, 15.433888	197
25	48.111403, 15.428590	160.5
27	48.099881, 15.422875	212
28	48.096802, 15.450801	107
29	48.094158, 15.460351	176
31	48.096025, 15.470488	108
33	48.095161, 15.475041	168.5
35	48.094973, 15.477507	220
39	48.095331, 15.484366	71
40	48.094820, 15.485075	93

Table 7.1: Selected curves with measured radius available for evaluating data quality

system running on two different deployment models can handle, such that result times are acceptable. Table 7.2 shows the metrics that are of interest for this test.

Metric Name	Description
response-time	Average time (in ms) that passed between all sent recommendation requests and all received responses. "How fast does the recommendation-service respond"
result-time	Average time (in ms) that passed between all sent recommendation requests and only received responses that contain curves. "How long do clients have to wait for curves as final result?"
response-ratio	Ratio of all sent requests (recommendation & poll) to any received response. "How many requests are answered?"
result-ratio	Ratio of all sent requests (recommendation only) to received curve responses. "How many requests are answered with curve results?"
number-of-drivers	Total number of drivers that were constantly driving during an experiment.

Table 7.2: Metrics used for performance experiments

7.3.1 Simulation Application for Performance Evaluation

In order to be able to test the system with multiple drivers, a simulation application as well as test-tracks are needed. Our simulation application to run the performance tests is part of the prototype implementation, introduced in the previous chapter of the thesis. The application is wrapped in a SpringBoot[Spr] application that contains the simulation application and a web-server. The simulation application can run a configurable test instance. Each test creates a number of configurable drivers. Every driver runs in its own thread. To avoid blocking the application (i.e. GPS locations shall be emitted continuously), each driver has two extra single-threads to send requests and handle responses.

$$numThreads = numDrivers * (1 + 2) = numDrivers * 3$$

A driver instance simulates GPS tuples that are read from available .csv-files. The GPS emits latitude, longitude pairs as they were recorded. Each driver can access the remote recommendation service via a gRPC-stub. Within a test run, Prometheus metrics are captured using the instrumentation Java client[Proc].
Our simulation application implements a web-server. Its main purpose is to export captured Prometheus metrics to an endpoint. A monitor running Prometheus can scrape the metrics from the provided API endpoint to see test results on the fly. Additionally, the web-server provides a simple UI to start tests, specify parameters and stop tests. The UI was implemented using the framework AngularJS[ang]. Figure 7.3 shows the components of the SpringBoot application and the UI to start tests. When running the performance application with thousands of drivers, in the worst-case, also thousands of OverpassAPI requests would be sent. Since the main-server is very limited in usage (see Section 6.3.2), an own private instance is setup in the cloud. On the private instance, the database needs to be initialized with map data. To reduce the installation size on the instance, a custom map export was created using BBBike's extract tool[Sch]. Since most of the trips in the database provided by the KFV lie within the "Graz & Umgebung" area, the database was initialized with this area only. The exact coverage of the private OverpassAPI instance can be seen in Figure 7.1.

7.3.2 Experiment 1 - Performance of cloud-only model

Experiment 1 tests a cloud-only model. In general both DM1 (Section 6.3.2) and DM2 (Section 6.3.2) could be used for testing this type of model. Since having limitless query possibilities, i.e. running an own instance of OverpassAPI, is crucial for doing the evaluation, DM2 qualifies best and will be used for this experiment.

Testbed for Experiment 1

To provide an appropriate cloud environment for the system, multiple resources hosted on Google Cloud Platform (GCP)[gcp] are used. For the evaluation, the presented list of required resources for DM2 (see Section 6.3.2) will be covered by virtual machines running at GCP. In order to avoid high costs for the evaluation, the minimum requirements of

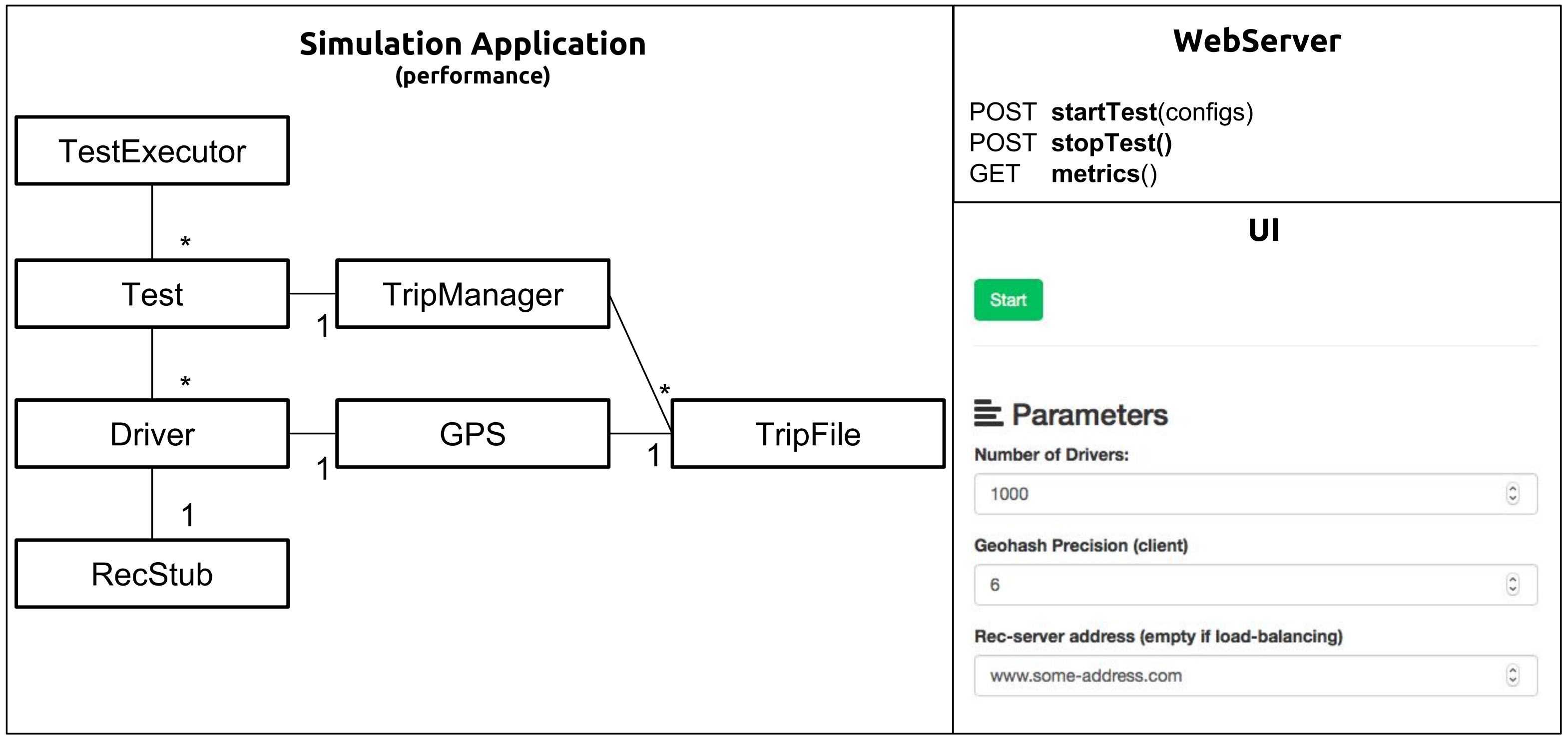


Figure 7.3: Components of the application to run performance tests

Name	Service	Type	Specs
Local Fleet		MacBook Air	8GB RAM, 2 x 1.8GHz, IntelCore i5, Network: LTE, Max Java Threads: 2009
Cloud Fleet	GCP	n1-standard-1	default, Network: 75-300mbps
Recommendation	GCP	n1-standard-1	default
Detection	GCP	n1-standard-4	default
OverpassAPI	GCP	n1-standard-8	default
MongoDatabase	GCP	n1-standard-1	default

Table 7.3: Hardware specifications for Experiment 1

this list, i.e. the lowest cardinalities of each element, are used. This implies that load balancing, as described in Section 5.5, is turned off. A positive effect of turning off load balancing is that it makes it easier to compare cloud- and edge-based models. Since load balancing and using many VMs in the cloud can theoretically be scaled almost unlimitedly, it would not be possible to find an actual maximum number of drivers that can be handled. Using load balancing in the cloud would also make it hard to compare the system to the edge model, where we have very limited resources (especially for the evaluation). The downside of course is that the possible maximum number of drivers will drop significantly. Also instead of having to pay for a hosted MongoDB cluster, for instance AtlasDB, a single self-hosted MongoDB instance suffices for the evaluation tests.

To simulate a large number of cars, multiple so called "fleets" of cars can be started on different machines. Therefore, the simulation application, which was presented earlier in Section 7.3.1, is deployed to each available machine. This way, every machine can start an arbitrary number of drivers. A local machine and a GCP virtual machine are used to create the fleets. To keep the system independent from the fleets and provide a more realistic setting, they are deployed in different regions. Hence, the system and the fleets never run in the same private network (VPC). The available internet speed at the cloud fleet is very fast. During the test runs it varied between 75mbps and 300mpbs. At the local machine, the maximum network bandwidth was throttled to Long Term Evolution (LTE) using Apple's Network Link Conditioner[NSH]. To give a realistic answer to question Q2, only result- and response times are used where the network was bound to LTE . Tables 7.3 and 7.4 list all specific resources and specifications that are used to provide the deployment for this experiment. Specifications for GCP resources are taken from the GCP web-page[Goo17b].

Software Component	Parameter	Value
Car Simulation	Application	performance
	LocalSearchBoundingBox	6
	Poll Delay	3s
	Max Polls	3 (6)
	GRPC Timeout	5s
Recommendation	Find Curves Mode	"geohash"
	Geohash-Precision	6
	Simulate WeatherAPI	true
	WeatherBoundingBoxSize	4
Detection	AggregationTimeWindow	1s
	Aggregation BB Size	6
	OSMPartitions	5
	Overpass-Server	private
	AngleThreshold	2°

Table 7.4: Software specifications for Experiment 1

Caching

A very important factor for evaluating the performance is the cache-hit-rate. On every request to the recommendation, a lookup is done first at the local cache on the requested service. In case nothing is found, the distributed database (which the detection writes to) is queried. Depending on how many curves are already in the caches, responses can be handled faster since the detection does not have to be called. To see this behavior, for each run, one of the two following cache configurations are available:

- *full-cache*: The distributed cache already has a predefined set of curves stored.
- *empty-cache*: The distributed cache is completely empty.

While empty-cache shows how the system behaves in its initial deployment phase, full-cache shows how the system behaves when its already deployed for a while. To be able to import data for the full-cache scenario, an initial test run with 100 drivers constantly driving for around 1 hour was executed. This resulted in the a cache data set of 20.000 curve-objects which corresponds to a database size of 5.2MB.
Before each full-cache test run, both caches at recommendation and the central database service were cleared and restored with the above mentioned pre-calculated curve set. In case the system is run on empty-cache, the software parameter *MaxPolls* is increased to 6. This is because in such a scenario, the detection will take longer to store the curves. To reduce the number of timeouts, the client polls the recommendation service for curves more often.

Test Runs of Experiment 1

To find out the number of drivers that can be handled, multiple runs will be executed on the given testbed. The number of drivers is increased on every following test run. This way we will find out the approximate boundaries of the prototype running on the configured test environment. For each test run, the metrics: *number-of-drivers*, *response-time*, *response-rate*, *result-time* and *result-rate* will be analyzed. To be consistent, all runs are executed for approximately the same duration. The duration for each test-run was set to approximately 7 minutes. In test runs 1-9, the goal is to increase the number of drivers and find the maximum the system can handle. The last two test runs 10 and 11 are designed in a way to find out a realistic value for response- and result times. In case a simulated driver finished its trip, i.e. it reached the end of the .csv-file to read in GPS coordinates, a new trip is started immediately. This way the number of drivers is always constant.
In Experiment 1, we run the simulation application natively on a local MacBookAir laptop. On the provided machine, a maximum of 669 drivers (= 2009 Java Threads / 3) can be started. To avoid running out of memory, the maximum number of drivers was bound to 500 on the local machine. On Linux (64 Bits) machines the maximum number of Java threads is much higher and during the test runs no problems occurred when running with 1500 or even 2000 drivers.

Run	Drivers	C/L	rT	rR	resT	resR	cache
1	100	50:50	288	97%	351	87%	full
2	500	50:50	361	90%	441	78%	full
3	1000	50:50	383	82%	391	70%	full
4	1500	67:33	958	77%	1021	65%	full
5	2000	75:25	4224	11%	4342	9%	full
7	100	50:50	174	99%	1457	61%	empty
8	500	50:50	184	99%	527	64%	empty
9	1000	50:50	226	99%	/	0%	empty
10	500	0:100	322	88%	383	77%	full
11	500	0:100	321	99%	726	83%	empty

Table 7.5: Test results of Experiment 1

Drivers: Total number of drivers that were constantly driving.
C/L: Ratio of how many drivers were running on the cloud instance compared to running locally.
rT: response-time
rR: result-rate
resT: result-time
resR: result-ratio
cache: Test started with "full-cache" or "empty-cache"

Evaluation of maximum number of drivers - cloud-only model

Using the results of the test runs from Experiment 1 (Table 7.5), we evaluate question Q1 and find out the maximum number of drivers running on the cloud-only model. As expected, the more drivers are added to the system, latencies increase, while response-and

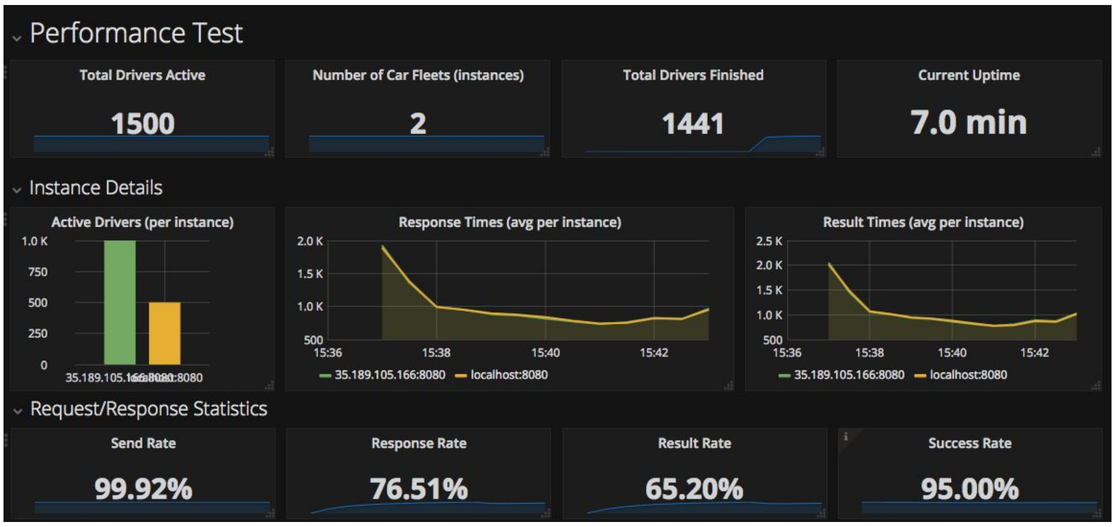


Figure 7.4: Prometheus metrics of successful test run 4 running in the cloud with a "full-cache" with 1500 constant drivers.

result rates decrease. The response-rate and with it the result-rate, slightly drops with adding more drivers. The main reason for this is that at the recommendation service, the single MongoDB instance can only handle up to 500 connection concurrently. If all connections are taken, this will lead to a gRPC-timeout, resulting in an unanswered response. In a production system, a hosted MongoDB cluster can be configured to serve more connections. The largest *M100* cluster hosted at AtlasDB for instance can serve up to 16000 connections[Monb]. In Figure 7.4 the Grafana[Lab]-Dashboard shows informations and metrics collected by Prometheus from the two target car fleets of run 4. Figure 7.5 shows that with caching, the streaming application at the detection service never receives more than 75 requests per second, resulting in very good latencies of only of around 500 ms. Test Run 5, with 2000 drivers, showed the limits for the prototype running without load balancing. On average, only around 10% of the requests resulted in responses or results. The reason for the failing requests is that the single recommendation server reached its limits after around 5 minutes of execution. As depicted in Figure 7.6, the CPU usage of the instance peaked to 95% and finally reached 100%. The gRPC-server could no longer handle requests, resulting all following requests to fail. In case more recommendation servers would be available with load balancing enabled, this state would have been avoided.
Test runs 7-9 show how the system performs when it is freshly deployed (empty-cache). While the response-times stay low and 99% of the requests are responded, the result times went up compared to running with full-cache. Since every request in the very beginning of the test run results in calling the detection, in the worst case, when no aggregates could be found, the Apex streaming application has to handle a request from every driver at the same time. While the system still was able to handle up to 500

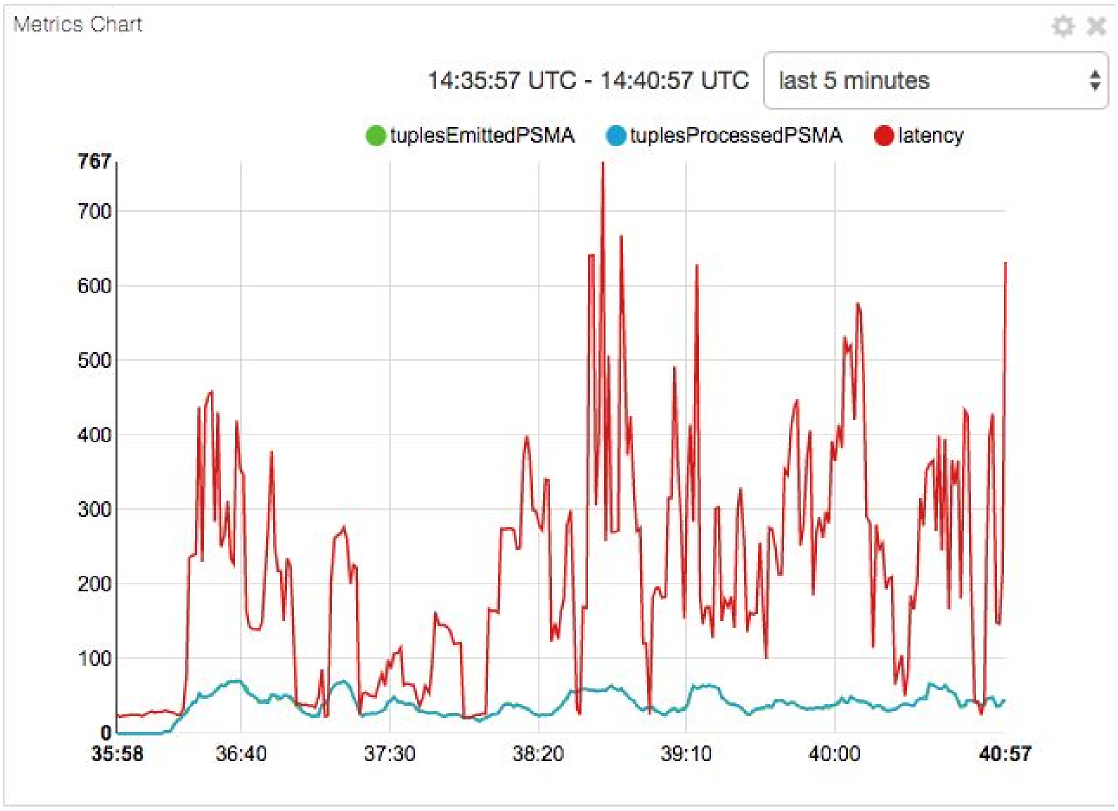


Figure 7.5: Apex metrics of successful test run 4 running in the cloud with a "full-cache" with 1500 constant drivers.

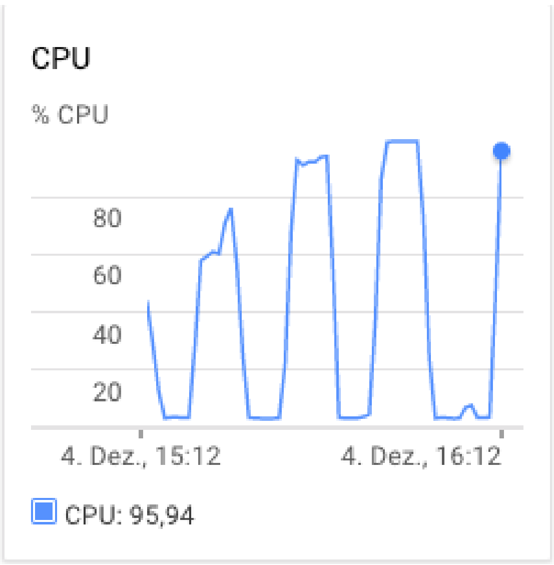
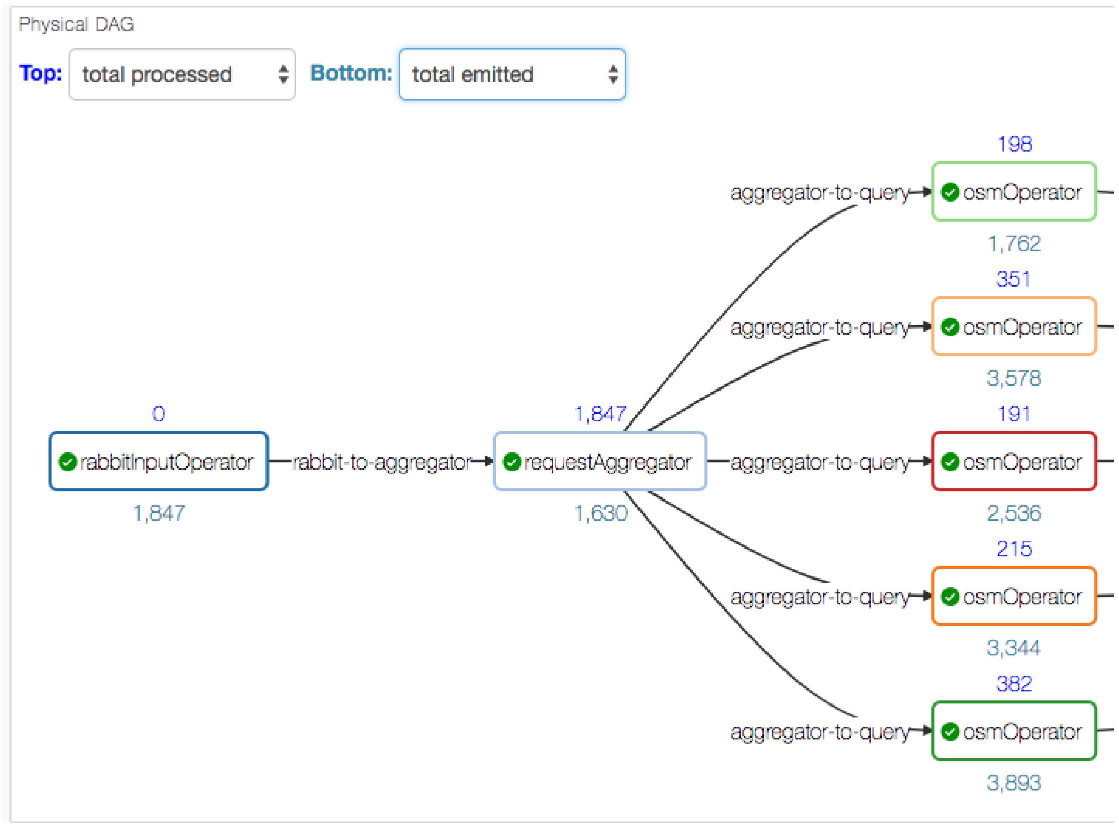


Figure 7.6: Failing recommendation service when running 2000 cars due to CPU limits.


Physical DAG

Top: total processed Bottom: total emitted

Figure 7.7: Screenshot of the Apache Apex Operator Widget showing that the aggregation only reduces incoming requests by 12%, eventually causing the application to fail to respond to requests.

drivers, at 1000 drivers the detection service failed. The reason for the failing detection service is the OSMOperator. When many requests are incoming at the same time, the single OverpassAPI instance blocks the application flow. Even though the OSMOperator was partitioned to 5 instances, still this did not resolve the bottleneck. As presented in 6.2.2, the Apex application contains a RequestAggregator that tries to aggregate similar requests by time and location. As Figure 7.7 shows, in test run 9 this resulted in 1630 tuples that the OSMOperators and OverpassAPI needed to handle. This caused extreme latencies, i.e. no tuples exited the pipeline and the recommendation service could not serve any further curves. The configured TimeWindow of only 1 second and geohash character precision of 6, were only able to reduce the tuples by around 12%. To decrease the tuples at the Aggregator, in another test run, the TimeWindow was increased to 3 and the Geohash precision was set to 4 (causing a very large area to create aggregates). Using this configuration, at 2000 incoming tuples, the operator was able to aggregate the tuples by almost 87% to only 266. Having larger aggregates though also implies having larger results in terms of data. While the latencies of the OSMOperators were

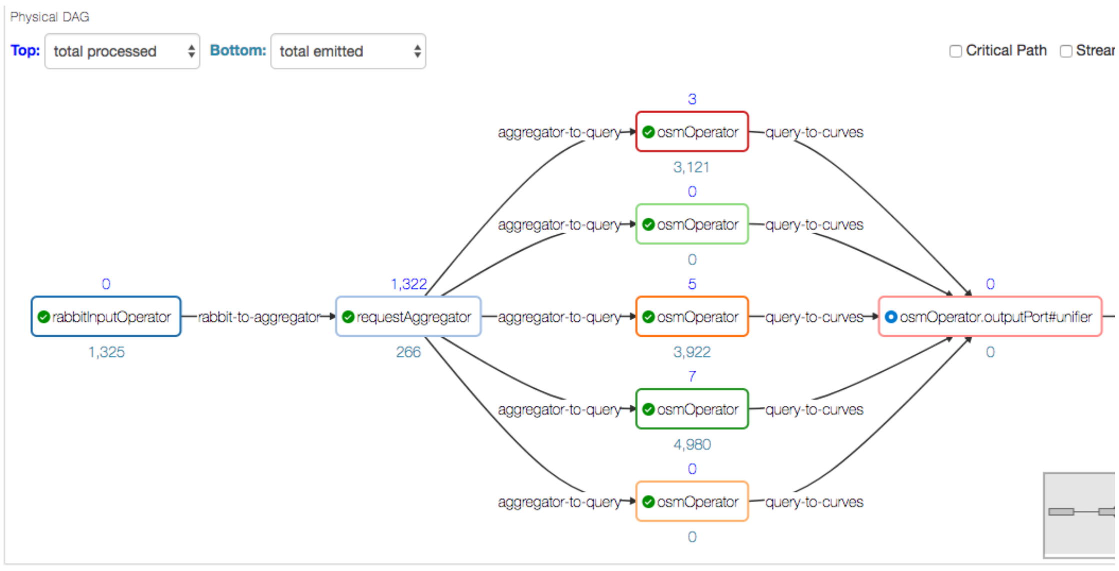


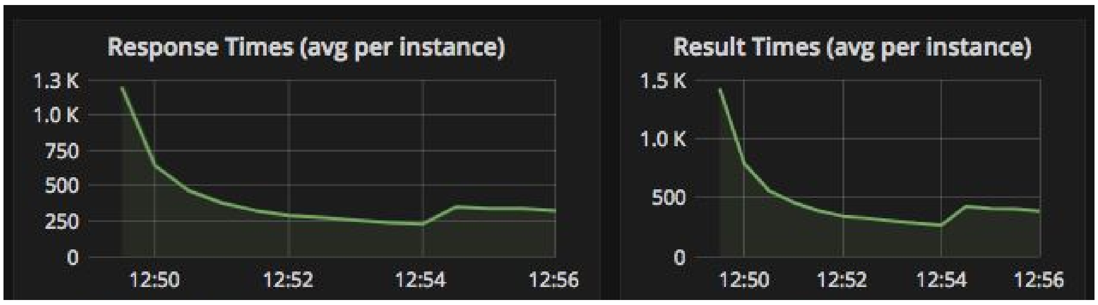
Figure 7.8: Failing unifier operator (marked in blue) when trying to increase aggregates to reduce the amount of tuples before querying Overpass.

reduced, the downstream operator to unify the results was not able to handle the amount of emitted ways and failed, causing the pipeline to be halted. Figure 7.8 shows the failing unifier operator in blue. One solution to fix this problem would be to also partition the downstream DetectCurves operator. When the OSMPartitions are set to 5, this would also lead to 5 additional containers for the DetectCurves operator. Having 5 additional containers would increase the already very high memory usage even more. While on a cloud instance this could be easily achieved by buying a larger VM, at the edge this is not yet a solution since computational resources are very limited.

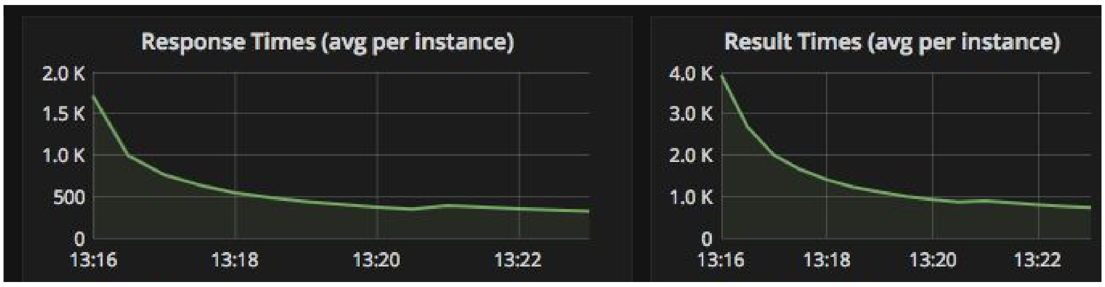
Running the application on the given testbed when it was already deployed (full-cache), **the maximum number of drivers lies between 1500 and 2000 drivers**.
Test runs 7-9 showed how important caching is for running the application. When the caches are empty, the system fails at running 500 to 1000 drivers. Before deploying the system to a ready-to-use state, it should be considered to run the application in detection-mode first for a certain time. This can be achieved by simply providing geohash locations that are known to be frequently driven to the detection service. Another option would be to scale the Overpass-server to multiple high-compute instances. While this would resolve the bottleneck, if low costs are of importance, this should be only considered as a second option.

Evaluation of result time - cloud-only model

With the given results, we can evaluate question Q2 - "How long do drivers have to wait for curve results?". To that end, we find out the average result-time when the prototype is deployed to the cloud-only model. With full-cache, the prototype showed very good



Run 10: 500 drivers, LTE network, “full-cache”



Run 11: 500 drivers, LTE network, “empty-cache”

Figure 7.9: Comparison of result times when running 500 drivers at LTE speed with full-cache vs. empty-cache

results until test run 4, with drivers receiving curves with an average delay of only around 1 second. Since in test runs 1-9, the cloud-fleet had very high network bandwidths available, these results are not used for determining the actual result time. This has been done in test runs 10 and 11, where a realistic scenario of 500 drivers with LTE network was executed. As it can be seen in Figure 7.9, the maximum result-time on full-cache, at the very beginning of the run, was at around 1.5 seconds. When the system was up for around 7 minutes on full-cache, this value dropped and on average drivers received curves in 383ms. Compared to when all caches were empty, the maximum result-time was at around 4 seconds and after 7 minutes of run time dropped to an average of 726ms.

When running 500 drivers at LTE speed (full-cache), **on average a driver receives curve results in less than 0.4 seconds**. On a system start-up (empty-cache) this value almost doubles and a driver still receives curve responses in no more than 0.8 seconds.

7.3.3 Experiment 2 - performance of cloud/edge model

The second experiment tests the combination of using both edge-and cloud computing models. Both DM3 (Section 6.3.2) and DM4 (Section 6.3.2) could be used for testing this type of model. As discussed in Section 6.3.2, DM4 shows how to use edge nodes to cover many specific areas. This deployment model suited well because of the fact that the private OverpassAPI covers only a specific area. Hence, Experiment 2 deploys DM4 using a simulated edge node located in Vienna. Instead of using actual data from the Vienna area, the available test tracks in Graz & Umgebung are used. The actual location of the data though is not of importance and any other region could also be used.

Testbed for Experiment 2

As deploying applications to MEC servers is not yet possible, edge resources, i.e. edge nodes, are simulated in the thesis. For the evaluation, an edge node is represented by a server provided by the TU Wien. The local server is provided by the DSG group[dsg] and runs in their "DSG Cloud" which is implemented in the open-source framework OpenStack[Opea]. The resources of the used virtual machines are configured in a way to be comparable to a *high compute* edge node that is currently used in the industry and was presented earlier in Table 6.2. For the lack of having multiple edge nodes, similar as in Experiment 1, the lowest cardinalities of the elements in the presented list for DM4 are used. Hence, instead of *n* edge nodes, only 1 edge node with *S(edge) = high compute* is deployed. Besides the edge node, DM4 requires an additional cloud data-center. Similarly to Experiment 1, GCP is used to provide resources in the cloud. As noted in the description of DM4, synchronization between edge databases and a central database is not covered in the prototype. Therefore a central (backup) database is not deployed at the cloud. To run an OverpassAPI instance that can handle many requests at a time, many resources are needed. To keep resources at the DSG Cloud low, the same instance, as it was deployed in Experiment 1, will be reused.
To increase the number of drivers that can be executed in one test, compared to Experiment 1, the simulation application was deployed via a Linux-based docker image on the local laptop. This way, it was tested that around 5000 drivers could theoretically run in parallel. The cloud fleet is no longer executed, as the goal is to provide an edge scenario with cars running in close proximity to the server. Table 7.6 breaks down the specification of the VMs for running the simulated edge node and the cloud node. The software specifications are the same as in Experiment 1. As presented in the very last line of the table, when summing up all resources of the VMs running in the simulated edge node, the CPU and memory specifications almost exactly match the one's specified by Cisco's *high compute* edge node (see Table 6.2). Differences are 1 additional CPU core, around 1.5 GB additional RAM and slightly more disk space of 20GB in total. To fit the specification of the simulated edge node, for the detection, an instance type of "m1.large" results in around 8GB of memory, compared to 15GB in Experiment 1. In Experiment 1 the actual used memory for the yarn resource manager was decreased to 12GB. In order to be able to run the Apex application with the same partition size as in Experiment 1, each operator memory was reduced from 512 MB to 256MB.

Test Runs of Experiment 2

Test runs will be executed in the same way as described in Experiment 1. This time though we want to simulate an edge scenario. Hence, drivers shall send requests within

Name	Service	Type	Specs
Local Fleet		MacBook Air	8GB RAM, 2 x 1.8GHz, Intel-Core i5, Network: LTE, Max Java Threads: 2009
OverpassAPI	GCP	n1-standard-8	default
Recommendation	DSG Cloud	m1.medium	CPU Cores: 2 Memory: 3.75GB DiskSpace: 40GB
Detection	DSG Cloud	m1.large	CPU Cores: 4 Memory: 7.68GB DiskSpace: 40GB
MongoDatabase	DSG Cloud	m1.small	CPU Cores: 1 Memory: 1.92GB DiskSpace: 40GB
Simulated **Edge Node (total)**	DSG Cloud		CPU Cores: 7 Memory: 13.5GB DiskSpace: 120GB

Table 7.6: Hardware specifications for Experiment 2

close proximity to the requested server. This was achieved by using only the local laptop (the same as in Experiment 1) sending requests to the simulated edge node in the DSG Cloud. During the tests, the laptop was run inside the TU Library, hence the actual distance to the simulated edge node was around 500m. For each test run, the metrics *number-of-drivers*, *response-time*, *response-rate*, *result-time* and *result-rate* will be analyzed. Again the test-runs were canceled at around 7 minutes. Test runs 1-4 executed on full-cache and 5-7 on empty-cache.

Evaluation of maximum number of drivers - cloud/edge model

Table 7.7 shows the results of Experiment 2. Until test run 5, the prototype running in the simulated edge node performed very well. Same as in Experiment 1, also in Experiment 2 the maximum number of drivers when running on full cache was found after running 2000 drivers in parallel. The reason again was the single gRPC server at the recommendation server that was no longer able to handle more requests, leading to failing responses and results. Since the specifications for the VMs are almost the same, it should not be too surprising that the values are very similar to Experiment 1 running in the cloud. The cloud prototype's response and result times are slightly better. The reason for this was the higher network bandwidth between the simulated drivers and the servers. When running on empty cache, compared to the cloud prototype, the edge

Run	Drivers	rT	rR	resT	resR	cache
1	100	395	99%	489	86%	full
2	500	493	84%	587	84%	full
3	1000	513	72%	584	84%	full
4	1500	951	88%	996	76%	full
5	2000	4551	8%	4612	7%	full
6	100	291	99%	1649	70%	empty
7	300	284	49%	1846	25%	empty
8	500	450	42%	/	/%	empty

Table 7.7: Test results of Experiment 2

Drivers: Total number of drivers that were constantly driving.
rT: response-time
rR: result-rate
resT: result-time
resR: result-ratio
cache: Test started with "full-cache" or "empty-cache"

prototype already failed earlier, at 500 drivers. The reason for this is the lower memory capacity for running the Apex application, causing the OSMOperators to fail.

Experiment 2 showed, that similar to Experiment 1, **the prototype is able to handle between 1500 and 2000 drivers** on hardware that can be compared to an edge node currently used in the industry. Running the prototype on an empty-cache reduced this number to only 300 drivers. This is due to the lower available memory for running Apache Apex to detect curves.

Evaluation of result time - cloud/edge model

At the last successful test run 4, the average delay between a driver sending requests to the simulated edge node and receiving results was at almost exactly 1 second.

The scenario showed that with using current LTE network standards, the response times are almost as low as the one's measured between simulated drivers and the cloud services. Hence, running 500 drivers at LTE speed and full-cache **on an edge node, on average a driver receives curve responses in around 0.6 seconds** (+ 0.2 compared to Experiment 1). A system start-up (empty-cache) was only possible to test for a maximum 300 drivers. In such a scenario, on average drivers receive curve responses in 1.8 seconds (+1.0 seconds compared to Experiment 1).

7.4 Data Quality Evaluation

An important measure of how well a system performs, is to evaluate the quality of data it produces. The goal is to find answers to the stated questions Q3 to Q6, stated in the introduction of the chapter (see Section 7.1). Table 7.8 shows the metrics that are of interest for this test.

Metric Name	Description
detection-rate	Percentage of successfully detected curves.
approaching-rate	Percentage of successfully classified curves as they approach while driving.
radius-error	Average error of calculated radius compared to measured radius.

Table 7.8: Metrics used for data quality experiments

To simulate a driver going along our predefined test-track (see Section 7.2.2), the GPS coordinates are replayed in sorted order. In order to realize a realistic driving speed, GPS coordinates can be emitted at a configurable speed. By calculating the distance to a previously emitted coordinate, the time delay (in seconds) to emit the next location is calculated with:

$$timeDelay(i) = \frac{haversine(location(i), location(i-1))}{\frac{speed}{3.6}}$$

To stay consistent across all following experiments, the constant speed is fixed to 75km/h.

7.4.1 Simulation Application for Data Quality Evaluation

Our simulation application to run the data quality tests is part of the prototype implementation. The application reads in our predefined test-track and replays the GPS coordinates in sorted order. The application is very similar to the application described in Section 7.3.1. In this application we implement the "Upcoming Curve Prediction" algorithm (Algorithm 5.4) and only simulate one driver at a time on the predefined track. Additionally, a UI running on the web-server exactly visualizes all results of the distributed detection algorithm as well as the local detection to predict upcoming curves. Alongside the results of the detection, the UI shows the curves of the predefined test-set, including its measured radius. Using the UI, the evaluation can be done by comparing the prototype's results to the expected results. To provide results to the UI in real-time, messages are sent to a RabbitMQ Server running on the same host as the application. The messages are then consumed by a WebSocket. In order to implement Algorithm 5.4 that detects upcoming curves while driving, a local database is needed that is able to perform queries by distance. To that end, a local MongoDB instance running on the application's host is used. On arrival of new results from the detection service at the recommendation gRPC stub, curve results are stored to the local MongoDB instance. Using MongoDB, we are able to query objects in the database by their distance to a given location using using *$near*[Mona].

7.4.2 Testbed for data quality experiments

Compared to the performance experiments, hardware and software configurations for running the data-quality evaluation do not change and are the same for all following

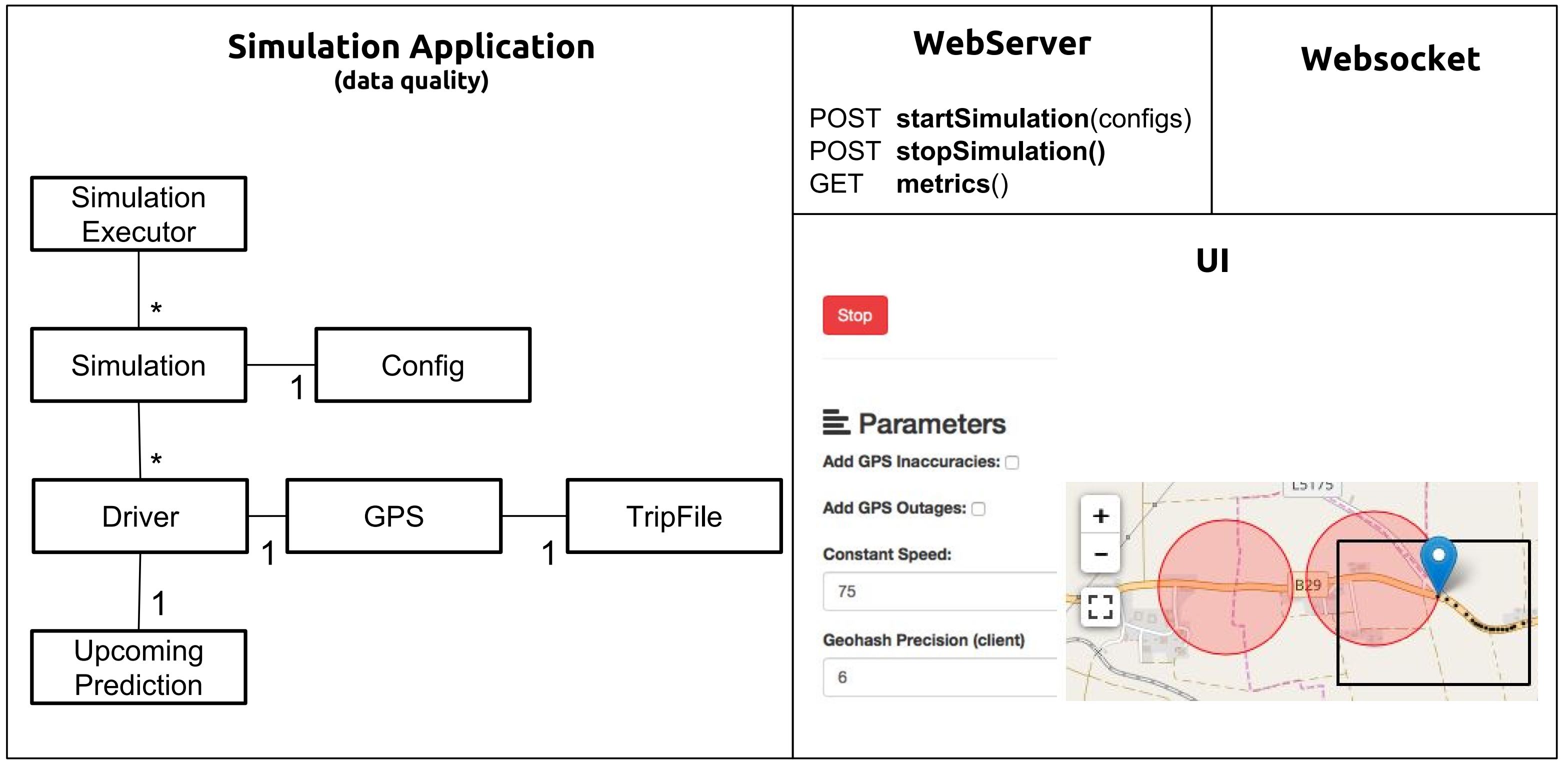


Figure 7.10: Components of the application to run data quality tests

Name	Service	Type	Specs
Car	/	MacBoookAir	CPU Cores: 2 Memory: 8.0GB Processor: 1.8GHz Intel Core i5
Recommendation	GCP	n1-standard-1	default
Detection	GCP	n1-standard-2	default
OverpassAPI	Main Server		
MongoDatabase	GCP	n1-standard-1	default

Table 7.9: Hardware specifications for data quality experiments

Software Component	Parameter	Value
Car	Application LocalSearchBoundingBox Poll Delay Max Polls	data quality 6 3s 3s
Recommendation	Find Curves Mode Geohash-Precision Simulate WeatherAPI WeatherBoundingBoxSize	"geohash" 6 false 4
Detection	AggregationTimeWindow Aggregation BB Size OSMPartitions Overpass-Server AngleThreshold	1s 6 5 Public Main-Server 2°

Table 7.10: Software specifications for Data Quality Experiments

experiments. Tables 7.9 and 7.10 show the specific hardware and software setups that we use for the data quality evaluation.

7.4.3 Experiment 3 - detection accuracy

In this experiment we give answers to Question Q3 - "How accurate are the detected curves?" and Questions Q4 - "How well are curves detected along roads? While driving, does the system detect approaching curves?". To that end, our predefined test-track is simulated once from the start to the end. To monitor results, Prometheus runs on the localhost of the test laptop. The monitoring client takes track of the following metrics: *detection-rate*, *approaching-rate* and *radius-error*.

ID	Detected	Approachedm	dmR	dR	recSpeed	error
1	true	true	286	341	78	55
4	false	false	286	/	/	/
5	true	true	245	262	68	17
7	true	true	296	459	91	163
9	true	false	352	703	112	352
11	true	true	226	260	68	34
13	true	true	246	267	69	22
15	true	true	150	162	54	12
17	true	true	229	242	66	14
19	true	true	294	360	80	66
21	true	true	250	256	68	7
23	true	true	197	191	58	6
25	true	true	161	190	58	30
27	true	true	212	292	72	80
28	true	true	107	531	97	424
29	true	false	176	439	88	263
31	true	true	108	246	66	138
33	false	false	169	/	/	/
35	true	true	220	253	67	33
39	false	false	71	/	/	/
40	true	true	93	234	65	141

Table 7.11: Test results of Experiment 3

Detected: The distributed algorithm detected the curve.
Approached: The simulation labeled the curve as approached while driving.
mR: Measured Radius
dR: Detected Radius
recSpeed: Recommended Speed
error: The error of the detected radius in meters

Evaluation of detection accuracy

Using the results of Experiment 3 (7.11), we evaluate questions Q3 and Q4 and receive an average radius error of 103.06 meters. **The overall detection-rate is 86% and the overall approaching-rate is 76%**. A detection-rate and approaching-rate of around 80% indicates that the system performs very well when detecting curves. Having an average radius of over 100m though, in general would suggest that the prototype cannot yet be classified as reliable when recommending speeds. If the results are studied in more detail, it is clearly visible that the detection seems to have problems with the curves: 2,7,9,28 and 29. Figure 7.11 visualizes these specific curves and on examination the following problems were found: Curves 7 and 40 are overlapping with other curves. Overlapping curves are still very error-prone for the detection. Curve 9 is a very large curve (radius > 300m). Curves with very large radius barely have angle differences between points. The curve detection algorithm does not perform well if angles are too low. Curve 29 has very few data points. If there are too few data points, the curve

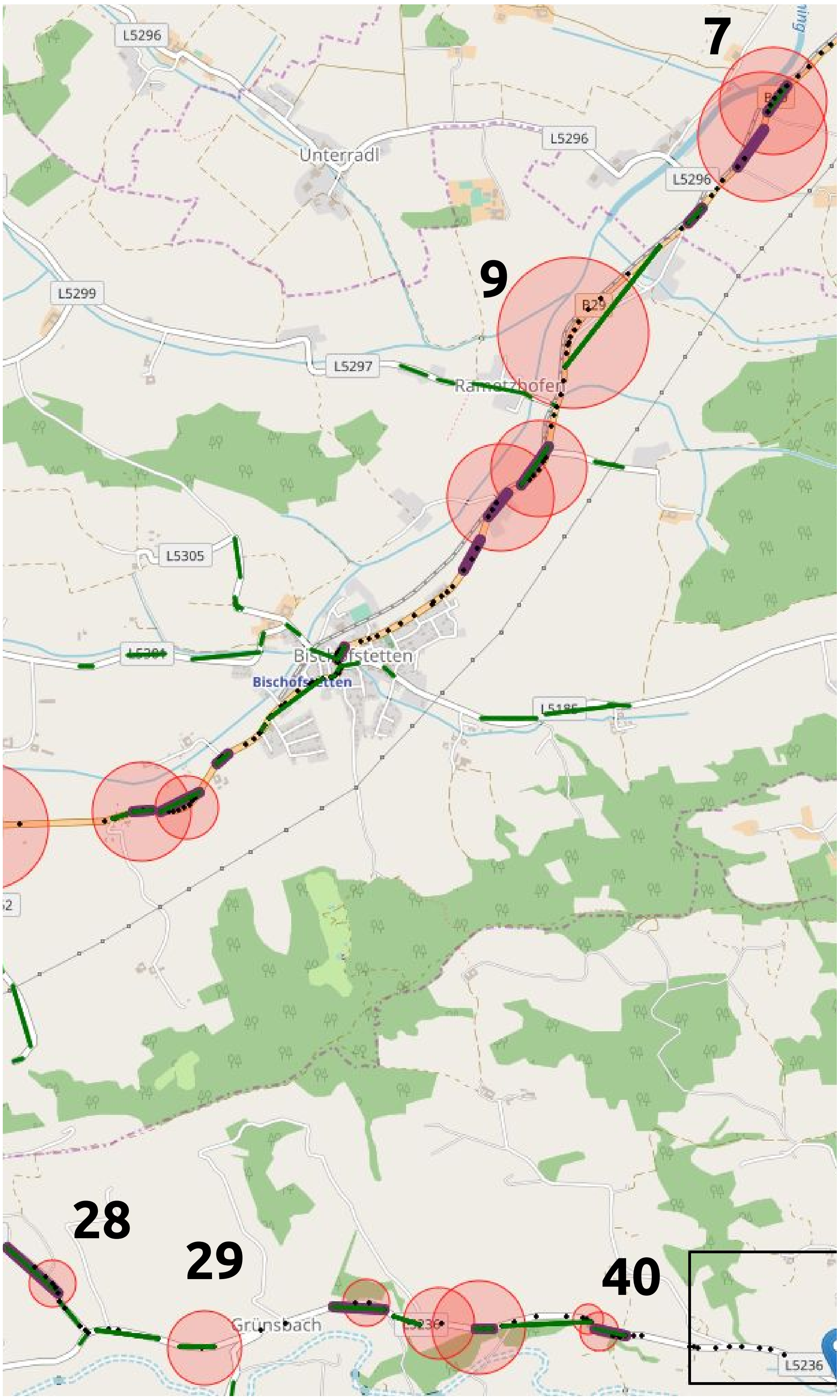


Figure 7.11: Problematic curves revealed on evaluating data quality for Experiment 1

Run	Detection Rate %	Approaching Rate %
1	86%	67%
2	86%	76%
3	86%	76%
4	86%	67%

Table 7.12: Test results of Experiment 4

detection is very error-prone. Discarding these 4 problematic curves the average error of the radius drastically drops to 46.64 meters.

7.4.4 Experiment 4 - GPS problems

The fourth experiment evaluates Question Q3 - "How well does the detection perform on inaccurate GPS positions, or even GPS outages?". This time our predefined test-track is simulated in four different test runs.
In the first run, inaccuracies are added to the GPS positions. This has been done by simply offsetting the GPS coordinates by a certain amount of meters. According to an evaluation from the year 2009 by Zandbergen et al.[Zan09], the iPhone 3 had an average accuracy of 8 meters. According to this study, the GPS error never exceeded 30 meters. Hence, for this experiment the worst case scenario of 30 meters is used as upper bound for the maximum error. For each GPS coordinate, a random error between 0 and 30 meters was added. This resulted in an average GPS error of 15 meters for all coordinates. The other 3 runs simulate GPS outages at random locations. To simulate outages we simply skip tuples for a certain duration. The number of outages and their duration are configurable and were changed as follows: In test run 2 we defined 5 outages with each having a duration of 10 seconds. In the next test run 3 we increased the duration to 20 seconds. In the last test run 4 we defined 10 outages with a duration of 20 seconds.
Since the accuracy was already determined in the previous experiment, the monitor for this experiment only takes track of the *detection-rate* and *approaching-rate*.

Evaluation of data quality with GPS problems

Table 7.12 shows the test results of Experiment 4. When the GPS was inaccurate, the detection-rate was not affected and stayed at 86%. The approaching rate though dropped by 9% to 67%. Figure 7.12 shows the two curves (25 and 40) that were not marked as approaching, causing the rate to drop. In the figure, green lines indicate curves (highly approximated using only 3 points) that were found by the remote detection. Purple lines indicate curves that have successfully been marked as approaching at the time of driving. As the figure shows, especially for curve 25 for instance, the locations (marked by black dots) are heavily offset, causing the local approaching algorithm to not detect an upcoming curve. When 5 outages occured with a duration of 10 or even 20 seconds, both the detection-rate and the approaching-rate were not affected and both stayed at 86% and 76%. Only when the number of outages was doubled to 10, with each outage lasting

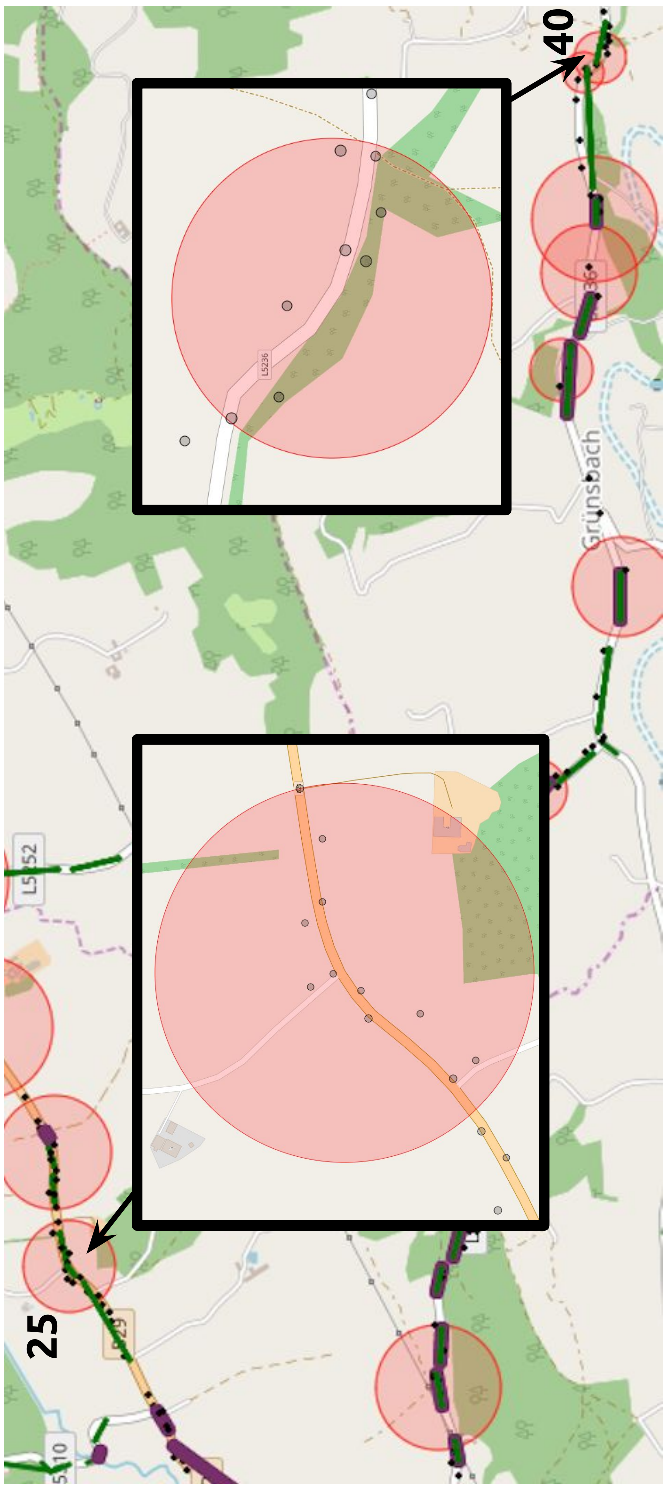


Figure 7.12: Test-drive that shows inaccuracies of GPS locations (0-30m) causing curve 25 and 40 to not be detected as approaching.

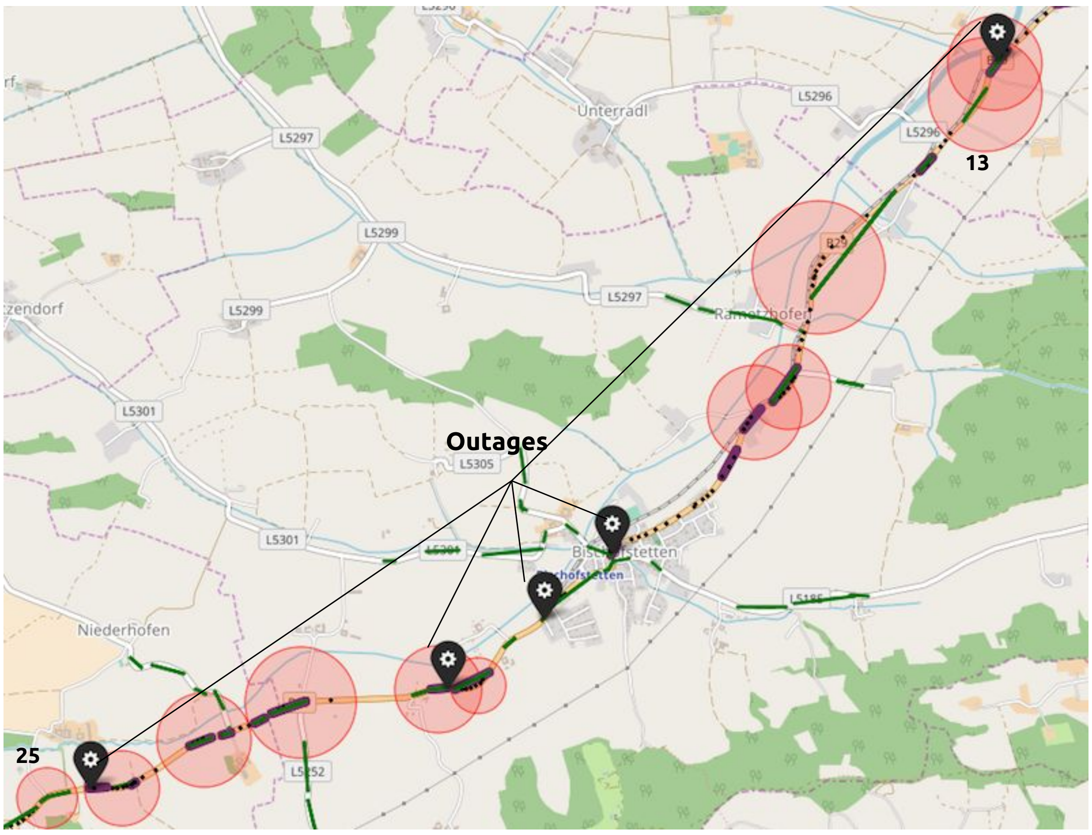


Figure 7.13: Test-drive that shows outages (black markers) causing curve 13 and 25 to not be detected as approaching.

20 seconds, the approaching-rate slightly drops by 9% to 67%. Figure 7.13 shows the two curves (13 and 25) that were not marked as approaching because outages happened right before. In the figure, the large black markers indicate outages. The outages are also visible by looking at the locations (black dots) which have holes. The output measures for this specific test of course highly depend at what exact time the outages happen. Since the tests places the outages randomly (at certain intervals given by the number of outages), the results though show that again, the system is very fault-tolerant even to complete outages. Again this is achieved by using geohashes and pre-calculating curves.

In general the detection is very fault-tolerant to inaccuracies and outages. This shows how efficient it is to pre-calculate curves for an area around the current location using geohashes. If cars would connect to a remote service to fetch curves on every GPS update, the rates would drop significantly when GPS errors occur. Using the approach of the thesis to only connect to a remote service at certain points and performing offline detection using the local cache, makes the system highly fault-tolerant.

7.4.5 Experiment 5 - Network problems

In Experiment 5 we evaluate Question - Q6 "What happens if the mobile network is slow?". As introduced in Experiment 1 (see Section 7.3.2) already, Network Link Conditioner can be used to throttle the network bandwidth. Network Link Conditioner provides many pre-configured profiles. 2 profiles that will be used in the experiment are summarized in Table 7.13.

Profile	Bandwidth	Delay(ms)	Packets Dropped(%)
LTE	50mbps%	50	0
Edge(2G)	240kbps%	400	0

Table 7.13: Network Profiles provided by Apple's Network Link Conditioner[NSH]

The values of the table only specify the downlink properties. Since the uplink values are almost the same for every profile, they are left out here. For all previous data quality experiments, LTE network conditions were used. In the first test run of this experiment, we simulate a very slow network connection using Edge (2G). In all previous test runs, both the recommendation and the central database made use of caching and already had curves in their caches. To simulate a worst-case scenario, additionally to using only Edge (2G) network, we also clear all caches. The monitoring client tracks the metrics *average-result-time* and *average-response-time*. Figure 7.14 shows the average response

Run	Detection Rate %	Approaching Rate %	rT	resT
1	86%	76%	1127ms	1127ms
2	86%	76%	1976ms	5891ms

Table 7.14: Test results of Experiment 5

and result times when executing different runs of the experiments. Compared to the first test-run of Experiment 4 with WiFi network and full caches, the average result time increased by a factor of almost 67 compared to the worst-case test run.

Evaluation of data quality with network problems

Table 7.14 shows the test results of Experiment 5. If the network is slow, both detection rates and approaching rates stayed unaffected at 86% and 76%. Previous runs of Experiment 3 and Experiment 4 used the standard WiFi connection and had full caches. This caused very low average response rates and result rates of 88ms. Although on a slow network the average result times increased to a factor of 12, the detection and approaching rate still stayed the same. Even on the worst-case scenario, when average result times rise up to a factor of 67 compared to the standard connection, the detection system works well. After seeing the results of Experiments 3 and 4, this is not too

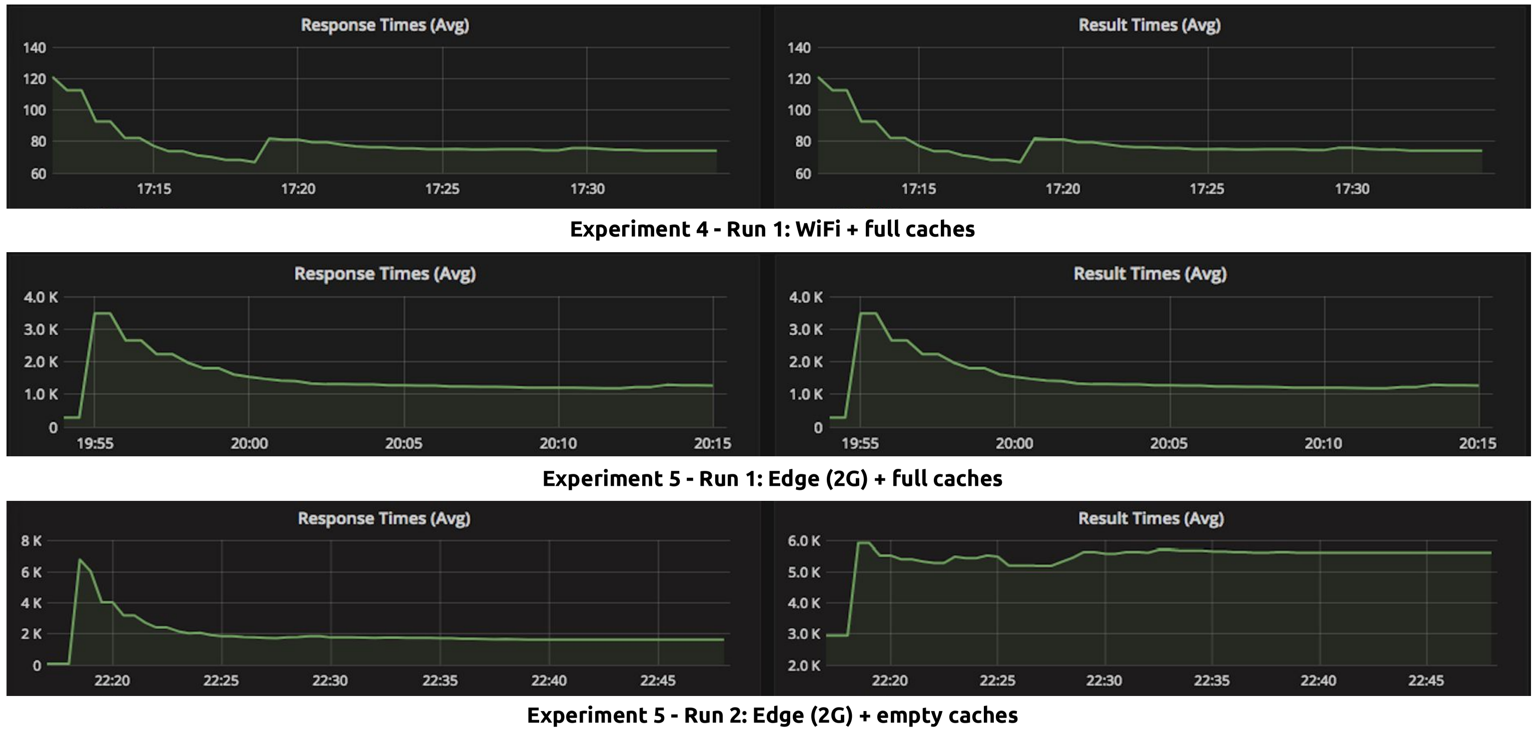


Figure 7.14: Screenshot of the Prometheus monitor showing average response-and result times of selected test runs during data quality evaluation.

surprising anymore. Again, pre-calculating curves for areas around the current location (using geohashing) makes the system highly tolerant against slow and even very slow network conditions. Experiment 5 showed that the system is also highly tolerant for slow network connections.

7.5 Cost Evaluation

This final section of the evaluation estimates costs for running the system on the presented edge and cloud models. An estimated cost shall be found for theoretically running the system in Austria for all drivers across the country. The evaluation gives an answer to the last question Q7 - "How much would it approximately cost to run the system for all drivers across Austria per month?".

7.5.1 Cost Estimation Basics

The basis for evaluating monthly costs to run the system across Austria is to use the results from the performance tests of Experiment 1 and Experiment 2 (see Sections 7.3.2 and 7.3.3). Both experiments showed the maximum number of drivers the prototype system can handle when deployed on the lowest cardinalities of their respective deployment model. For each experiment, the costs of the testbed it executed on are calculated per month. An estimate can then be calculated by scaling the costs to the estimated number of cars in Austria that are on the road for the duration of the experiments.
Statistics from 2016 provided by Statistik Austria and the VCÖ showed, that an average driver in Austria goes around 34km a day by car[VCÖ16]. According to the same statistics, there are around 3 million car drivers in Austria. It follows that in total, drivers in Austria go around 102 million kilometers by car a day. The average kilometers of all drivers in Austria for a certain duration (in minutes) can then be calculated as follows:

$$totalAvgKm = \frac{\frac{kmPerDay}{24}}{60} * durationMinutes$$

The approximate duration for the performance tests of Experiment 1 and 2 took 7 minutes. Using the formula above, the total average distance that is driven in Austria in that duration is around : 0.5 million kilometers.

It remains to find out how many kilometers the drivers covered during the test runs of the experiments. In a separate test run, which again executed for 7 minutes with 500 drivers simultaneously, the total traveled distance was around 3000 kilometers. Hence, this test run covered 0.006% of all drives in Austria at that time. To receive the kilometer coverage for any test run the following formula is used:

$$kmCoveredInExperiment = \frac{maxNumberOfDrivers}{500} * 3000$$

Service	**Price in $ per month**
recommendation	28.43
detection	108.6
MongoDB (Atlas)	8.99
Overpass	223.2
Total:	369.22
Max Drivers:	1750
Scale Factor:	47.62
OWM subscription:	2000.00
Scaled Costs to cover Austria:	19582.26

Table 7.15: Estimated costs for running the prototype on a cloud model

Finally we can receive a factor to scale the costs to cover all drivers in Austria using the formula:

$$costScaleFactor = \frac{totalAverageKm}{kmCoveredInExperiment}$$

7.5.2 Costs for cloud model

Table 7.15 breaks down the costs for running the prototype on the cloud model. The prices are calculated for the specified resources (Section 7.3.2) using the GCP Price Calculator Tool[Goob]. For using the OpenWeatherMaps (OWM) API with all drivers across Austria, we assume to subscribe to an Enterprise license that allows up to 200.000 requests per minute[owm]. As described in Experiment 1 (see Section 7.3.2), to save costs during the evaluation, the hosted MongoDB provider Atlas was not used. An own database instance was sufficient. For the price estimation, a comparable "M2" AtlasDB configuration[Monb] to the single GCP VM instance was selected.

7.5.3 Costs for edge model

To estimate the costs for the simulated Edge Node of Experiment 2 (see Section 7.3.3), the earlier discussed (6.3.2) *high compute* Edge Node of Cisco System's EFF platform is examined. On their product data sheet[Cis17c] (see screenshot in Figure 7.15) they recommend specific hardware for the node. The following costs are covered by the estimation:

- **Server Hardware Costs:** Assuming the recommended "Cisco UCS C220 Rack Server" is used, the price at Cisco at the time of writing this thesis was 3045.00$[Cis]. Further assuming that the server has a service life of 10 years, the monthly costs for the server can be estimated to: 25.38$.
- **Energy Consumption:** According to Cisco's spec-sheet, the energy consumption lies at 915W[Cis17b]. In Austria the price for one kWh can be estimated with

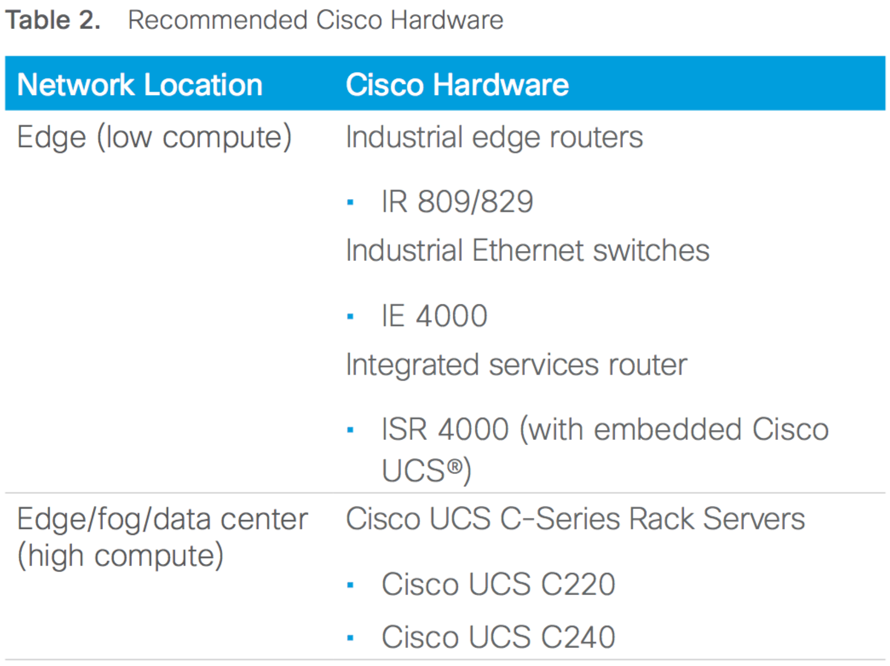


Table 2. Recommended Cisco Hardware

Network Location	Cisco Hardware
Edge (low compute)	Industrial edge routers ▪ IR 809/829 Industrial Ethernet switches ▪ IE 4000 Integrated services router ▪ ISR 4000 (with embedded Cisco UCS®)
Edge/fog/data center (high compute)	Cisco UCS C-Series Rack Servers ▪ Cisco UCS C220 ▪ Cisco UCS C240

Figure 7.15: Hardware recommendation for the "EFF Fog Node Server"[Cis17a]

around 20cents[EC] (0.22$ using the currency of 05.12.2017). To get the monthly energy costs, the following formula is used:

$$monthlyEnergyCosts = \frac{W * 30 * 24 * pricePerKWh}{1000}$$

- **Internet Costs:** Choosing a business product of Austria's well known provider A1[A1] that allows up to 300MBit/s download and 30MBits/s upload, the monthly costs would currently lie at: 114.15$.

The above list of course does not cover all costs that would arise when installing an edge node. If running the edge node at a BTS, additional costs such as cooling, heating, maintenance workers or rental payments can emerge. Table 7.16 breaks down the costs for running the prototype on a cloud/edge model.

7.5.4 Evaluation of costs

The estimated total costs per month to run the system across Austria in the cloud-only model are around **20.000$**. Running the system on a combined edge-cloud model would yield in around **27.000$**. Edge computing is still very new and currently only few companies provide real-world solutions. If edge computing is becoming more popular in the industry, expectedly there will be cheaper edge hardware available. Since using an edge-infrastructure to run this application would yield in approximately 35% higher costs,

Service/Expense	**Price in $ per month**
Hardware Costs	25.38
Power Consumption	144.94
Internet Infrastructure	114.15
MongoDB (Atlas)	8.99
Overpass	223.20
Total:	516.65
Max Drivers:	1750
Scale Factor:	47.62
OWM subscription:	2000.00
Scaled Costs to cover Austria:	26602.87

Table 7.16: Estimated costs for running the prototype on a cloud/edge model

we currently recommend to deploy our cornering-assitance-application to a cloud-only model.

CHAPTER 8

Conclusion & Future Work

8.1 Conclusion

In this thesis we introduced a novel system that assists drivers in real-time while cornering and designed it in a way that it can be deployed to existing cloud and emerging edge-computing infrastructures. We identified relevant data attributes from drivers and external data sources, determined their frequencies of retrieval and proposed how to efficiently fuse them. Using the microservices pattern we designed an architecture that supports virtualized MEC infrastructures, separates tasks and concerns and enables scalability. In order to accomplish the tasks of the cornering assistance application, we implemented new algorithms to detect curves and properties, recommend safe speeds for entering a curve, predict upcoming curves along the driver's path from a given location and also presented an approach of how to load balance requests depending on factors such as server load or location of clients. Based on these contributions we implemented an open-source prototype which is available at: `https://github.com/rdsea/EdgeCorneringAssistance.git`

The evaluation of the implemented prototype shows that more than a thousand drivers can use the system simultaneously and receive results from the distributed system in almost real-time (less than half a second). Using load balancing, the prototype is designed to scale to many thousands of drivers depending on the available hardware and infrastructure. Our assistance-application, using the results from our elaborated algorithms, is able to warn drivers before they approach a curve at very high detection rates of more than 76%. The evaluation showed that our system is highly fault tolerant to GPS errors such as inaccuracies or even full outages. Our system also proved to work just as well on slow network connections, which can occur frequently when being on the road. Results of the accuracy of the system, i.e. calculating properties of curves, show that for many curve types the calculated radius has errors between approximately 5 to 50 meters. However, the evaluation also showed that some types of curves were not detected

at all or that the error of the calculated radius exceeded 100 meters.
To show how much it would cost to deploy such a system using future edge-infrastructures, Austria was used as an example. The costs were evaluated for a theoretical scenario, where all drivers across the country are served by the system. We concluded that the costs would be around 20.000$ per month using a cloud-only model. Running the system on a combined edge-cloud model we estimated around 35% higher costs.

8.2 Future Work

For the system to be considered as safe to operate on the streets, the detection rate and especially the accuracy would need to be tested with more streets and be adjusted to fit all types curves. For the lack of having human experts available, the recommended speed that the system provides was not evaluated. The thesis only gave a first approach on how to recommend speeds for approaching curves. To that end, currently only latitude and longitude data were processed and used for the calculation. An important value though also is the driver's altitude and the slope a curve has. Hence, in future works the detection can be enhanced to recommend speeds more accurately using also height data. Since the focus of this thesis was to implement a distributed detection algorithm, client functionalities were only simulated and written as small Java Applications. Implementing a native application that can actually be installed at the driver's phone or even in the car itself could be another part of future works. While load balancing was implemented based on only factors such as server load or location of clients, we proposed many other factors that could be implemented for this and possibly also other MEC-based applications in the future.

Bibliography

[A1] A1. Internet im büro. https://www.a1.net/business/produkte-angebote/internet/internet-im-buero/s/internet-im-buero. (Accessed on 08/12/2017).

[AA16] Arif Ahmed and Ejaz Ahmed. A survey on mobile edge computing. *2016 10th International Conference on Intelligent Systems and Control (ISCO)*, Jan 2016.

[AIdCA12] Rui Araujo, Angela Igreja, Ricardo de Castro, and Rui Esteves Araujo. Driving coach: A smartphone application to evaluate driving efficient patterns. *2012 IEEE Intelligent Vehicles Symposium*, Jun 2012.

[aka] akapribot. Owm japis. https://bitbucket.org/akapribot/owm-japis. (Accessed on 05/12/2017).

[ang] Angularjs — superheroic javascript mvw framework. https://angularjs.org/. (Accessed on 01/22/2018).

[ape] Best practices - apache apex documentation. https://apex.apache.org/docs/apex/development_best_practices/. (Accessed on 12/12/2017).

[Aus16] Statistik Austria. Unfaelle mit personenschaden. https://www.statistik.at/web_de/statistiken/energie_umwelt_innovation_mobilitaet/verkehr/strasse/unfaelle_mit_personenschaden/019874.html, 2016.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[BVRK12] Ravi Bhoraskar, Nagamanoj Vankadhara, Bhaskaran Raman, and Purushottam Kulkarni. Wolverine: Traffic and road condition estimation using smartphone sensors. *2012 Fourth International Conference on Communication Systems and Networks (COMSNETS 2012)*, Jan 2012.

[CDFE15] German Castignani, Thierry Derrmann, Raphael Frank, and Thomas Engel. Driver behavior profiling using smartphones: A low-cost platform for driver monitoring. *IEEE Intelligent Transportation Systems Magazine*, 7(1):91–102, 2015.

[Cis] Cisco. Cisco ucs c220 m4 rack server data sheet. `https://www.cisco.com/c/en/us/products/servers-unified-computing/ucs-c220-m4-rack-server/index.html`. (Accessed on 04/12/2017).

[Cis17a] Cisco. Cisco edge fog fabric (eff). Technical report, Cisco Systems, April 2017.

[Cis17b] Cisco. Cisco ucs c220 m4 rack server spec sheet. `https://www.cisco.com/c/dam/en/us/products/collateral/servers-unified-computing/ucs-c-series-rack-servers/c220m4-sff-spec-sheet.pdf`, Dec 2017. (Accessed on 08/12/2017).

[Cis17c] Cisco. Edge fog fabric data sheet. `https://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/edge-fog-fabric/datasheet-c78-738866.html`, Apr 2017. (Accessed on 05/12/2017).

[Coo] ObjectLabs Cooperation. mlab cloud mongodb hosting. `https://mlab.com/company/`. (Accessed on 05/12/2017).

[Deu] MathWorks Deutschland. Azimuth between points on sphere or ellipsoid. `https://de.mathworks.com/help/map/ref/azimuth.html`. (Accessed on 02/11/2017).

[DLR$^+$15] Saurabh Daptardar, Vignesh Lakshminarayanan, Sharath Reddy, Suraj Nair, Saswata Sahoo, and Purnendu Sinha. Hidden markov model based driving event detection and driver profiling from mobile inertial sensor data. *2015 IEEE SENSORS*, Nov 2015.

[dsg] Distributed systems group. `http://www.infosys.tuwien.ac.at/`. (Accessed on 01/22/2018).

[dSW17] Magistrat der Stadt Wien. Offene daten österreich | data.gv.at. `https://www.data.gv.at/`, May 2017. (Accessed on 05/09/2017).

[DTB$^+$10] Jiangpeng Dai, Jin Teng, Xiaole Bai, Zhaohui Shen, and Dong Xuan. Mobile phone based drunk driving detection. *Proceedings of the 4th International ICST Conference on Pervasive Computing Technologies for Healthcare*, 2010.

[EC] E-Control. Was kostet eine kwh? https://www.e-control.at/konsumenten/strom/strompreis/was-kostet-eine-kwh. (Accessed on 08/12/2017).

[EMAY12] H. Eren, S. Makinist, E. Akin, and A. Yilmaz. Estimating driving behavior by a smartphone. *2012 IEEE Intelligent Vehicles Symposium*, Jun 2012.

[ETS17] ETSI. Etsi - industry specification groups (isgs). http://www.etsi.org/about/how-we-work/how-we-organize-our-work/industry-specification-groups-isgs, 2017. (Accessed on 03/25/2017).

[fMuG] Zentralanstalt für Meteorologie und Geodynamik. Zamg website. https://www.zamg.ac.at/cms/de/aktuell. (Accessed on 01/11/2017).

[Foua] Apache Software Foundation. Apache apex. https://apex.apache.org/. (Accessed on 02/07/2017).

[Foub] Apache Software Foundation. Apache apex malhar documentation. https://apex.apache.org/docs/malhar/. (Accessed on 02/07/2017).

[Fouc] OpenStreetMaps Foundation. Geofabrik. https://www.geofabrik.de/. (Accessed on 04/11/2017).

[Foud] OpenStreetMaps Foundation. Openstreetmap wiki. http://wiki.openstreetmap.org/wiki/DE:Key:highway. (Accessed on 04/04/2017).

[Foue] OpenStreetMaps Foundation. Osm. https://www.openstreetmap.org. (Accessed on 12/04/2017).

[Fouf] OpenStreetMaps Foundation. Osmf server info. https://hardware.openstreetmap.org/. (Accessed on 03/12/2017).

[Fou17] OpenStreetMaps Foundation. Overpass api. http://wiki.openstreetmap.org/wiki/Overpass_API, 2017. (Accessed on 05/03/2017).

[Fra17] Adam Franco. Curvature - find twisty roads. http://roadcurvature.com/, 2017. (Accessed on 03/12/2017).

[fV15] Kuratorium für Verkehrssicherheit. Verkehrsunfallstatistik 2014. http://unfallstatistik.kfv.at/, 2015.

[gcp] Cloud-computing, hostingdienste und apis von google | google cloud platform. https://cloud.google.com/. (Accessed on 01/22/2018).

[geo] Geohash - wikipedia. https://en.wikipedia.org/wiki/Geohash. (Accessed on 12/12/2017).

[GLME+15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, September 2015.

[Gooa] Google. Maps. `https://maps.google.com`. (Accessed on 12/04/2017).

[Goob] Google. Price calculator. `https://cloud.google.com/products/calculator/`. (Accessed on 05/12/2017).

[Goo17a] Google. Location | android developers. `https://developer.android.com/reference/android/location/Location.html`, 2017. (Accessed on 05/03/2017).

[Goo17b] Google. Machine types. `https://cloud.google.com/compute/docs/machine-types`, Nov 2017. (Accessed on 02/12/2017).

[GWA+17] Dennis Grewe, Marco Wagner, Mayutan Arumaithurai, Ioannis Psaras, and Dirk Kutscher. Information-centric mobile edge computing for connected vehicle environments: Challenges and research directions. In *Proceedings of the Workshop on Mobile Edge Communications*, pages 7–12. ACM, 2017.

[Has] HashiCorp. Consul. `https://www.consul.io/`. (Accessed on 06/07/2017).

[HCS15] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 133–144, New York, NY, USA, 2015. ACM.

[Her] Here. Wego. `https://wego.here.com`. (Accessed on 12/04/2017).

[HL97] D. L. Hall and J. Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1):6–23, Jan 1997.

[Inca] Docker Inc. Docker. `https://www.docker.com/`. (Accessed on 08/09/2017).

[Incb] Docker Inc. Swarm mode overview | docker documentation. `https://docs.docker.com/engine/swarm/`. (Accessed on 12/23/2017).

[JT11] Derick A. Johnson and Mohan M. Trivedi. Driving style recognition using a smartphone as a sensor platform. *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Oct 2011.

[kfv] Kuratorium für verkehrssicherheit. https://www.kfv.at/. (Accessed on 12/12/2017).

[KMJ16] Stojan Kitanov, Edmundo Monteiro, and Toni Janevski. 5g and the fog — survey of related technologies and research directions. *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, Apr 2016.

[Kow12] Wolfgang Kowarschick. Multimedia-Programmierung. Hochschule Augsburg, Fakultät für Informatik, gehalten im Wintersemester 2012, 2012. Die Lehrinhalte der Vorlesung „Multimedia-Programmierung“ werden nicht in einem Skript, sondern im GlossarWiki und auch auf der Web-Site http://mmprog.hs-augsburg.de/ zur Verfügung gestellt.

[kur] Kurvenradius – wikipedia. https://de.wikipedia.org/wiki/Kurvenradius. (Accessed on 01/09/2018).

[Lab] Grafana Labs. Grafana. https://grafana.com/. (Accessed on 02/12/2017).

[Let] Lettuce. Lettuce. https://lettuce.io/. (Accessed on 12/12/2017).

[LFHAB16] Yue Li, Pantelis A Frangoudis, Yassine Hadjadj-Aoul, and Philippe Bertin. A mobile edge computing-based architecture for improved adaptive http video delivery. In *Standards for Communications and Networking (CSCN), 2016 IEEE Conference on*, pages 1–6. IEEE, 2016.

[LKA$^+$16] Zhaojian Li, Ilya Kolmanovsky, Ella Atkins, Jianbo Lu, Dimitar P. Filev, and John Michelini. Road risk modeling and cloud-aided safety-based route planning. *IEEE Transactions on Cybernetics*, 46(11):2473–2483, Nov 2016.

[LLL$^+$12] Kun Li, Man Lu, Fenglong Lu, Qin Lv, Li Shang, and Dragan Maksimovic. Personalized driving behavior monitoring and analysis for emerging hybrid vehicles. *Pervasive Computing*, pages 1–19, 2012.

[MCCM13] Javier E. Meseguer, Carlos T. Calafate, Juan Carlos Cano, and Pietro Manzoni. Drivingstyles: A smartphone application to assess driver behavior. *2013 IEEE Symposium on Computers and Communications (ISCC)*, Jul 2013.

[Mic] Microsoft. Bing maps. https://www.bing.com. (Accessed on 12/04/2017).

[Mona] MongoDB. Mongodb manual 3.6. https://docs.mongodb.com/manual/reference/operator/query/near/.

[Monb] Inc. MongoDB. Mongodb atlas pricing. https://www.mongodb.com/cloud/atlas/pricing. (Accessed on 02/12/2017).

[Moo17] Samuel K. Moore. Superaccurate gps chips coming to smartphones in 2018. *IEEE Spectrum*, September 2017.

[Mor16] Iain Morris. Etsi drops 'mobile' from mec | light reading. http://www.lightreading.com/mobile/mec-(mobile-edge-computing)/etsi-drops-mobile-from-mec/d/d-id/726273, 2016. (Accessed on 03/25/2017).

[NSH] NSHipster. Network link conditioner. http://nshipster.com/network-link-conditioner/. (Accessed on 02/12/2017).

[Oas] Oasis. Amqp. https://www.amqp.org/about/what/. (Accessed on 02/07/2017).

[obd] On-board diagnostics - wikipedia. https://en.wikipedia.org/wiki/On-board_diagnostics. (Accessed on 01/16/2018).

[Opea] Openstack. Openstack. https://www.openstack.org/. (Accessed on 03/12/2017).

[Opeb] OpenWeatherMap. Openweathermap. https://openweathermap.org/weather-conditions. (Accessed on 05/26/2017).

[Ope17a] OpenStreetMaps. Openstreetmap wiki. https://wiki.openstreetmap.org/wiki/Main_Page, 2017. (Accessed on 03/12/2017).

[Ope17b] OpenWeatherMap. Weather conditions - openweathermap. https://openweathermap.org/weather-conditions, 2017. (Accessed on 05/26/2017).

[OSR] Project OSRM. Osrm api documentation. http://project-osrm.org/docs/v5.10.0/api/#general-options. (Accessed on 02/12/2017).

[owm] Price and limitation of openweathermap api. https://openweathermap.org/price. (Accessed on 01/17/2018).

[pgs] Postgresql: The world's most advanced open source database. https://www.postgresql.org/. (Accessed on 01/22/2018).

[PKZM12] Johannes Paefgen, Flavius Kehr, Yudan Zhai, and Florian Michahelles. Driving behavior analysis with smartphones: Insights from a controlled field study. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, MUM '12, pages 36:1–36:8, New York, NY, USA, 2012. ACM.

[PNC$^+$14] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[Proa] Prometheus. Monitoring system & time series database. `https://prometheus.io/`. (Accessed on 07/08/2017).

[Prob] Prometheus. Node exporter. `https://github.com/prometheus/node_exporter`. (Accessed on 07/08/2017).

[Proc] Prometheus. Prometheus instrumentation library for jvm applications. `https://github.com/prometheus/client_java`. (Accessed on 01/12/2017).

[Rab] RabbitMQ. Rabbitmq java client library. `http://www.rabbitmq.com/java-client.html`. (Accessed on 02/07/2017).

[Rai] Martin Raifer. Overpass turbo. `http://overpass-turbo.eu/`. (Accessed on 04/04/2017).

[Red] Redis. Redis. `https://redis.io/`. (Accessed on 04/04/2017).

[Ref] Math Open Reference. Circumcircle of a triangle. `https://www.mathopenref.com/trianglecircumcircle.html`. (Accessed on 12/10/2017).

[RSSB10] Michele Ruta, Floriano Scioscia, Eugenio Di Sciascio, and Politecnico Di Bari. A mobile knowledge-based system for on-board diagnostics and car driving assistance. *UBICOMM 2010 : The Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2010.

[Sch] Wolfram Schneider. Bbbike. `https://extract.bbbike.org/`. (Accessed on 01/12/2017).

[Sch11] Stephan Schmidl. Untersuchung des fahrverhaltens in unterschiedlichen kurvenradien bei trockener fahrbahn. Master's thesis, Universität für Bodenkultur Wien, Mar 2011.

[Spr] Spring. Spring boot project. `https://projects.spring.io/spring-boot/`. (Accessed on 01/12/2017).

[SPT15] Chalermpol Saiprasert, Thunyasit Pholprasit, and Suttipong Thajchayapong. Detection of driving events using sensory data on smartphone. *International Journal of Intelligent Transportation Systems Research*, 15(1):17–28, Jul 2015.

[SSX+15] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[Ste97] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[Str] Johannes Strommer. Haftreibungszahlen. `https://www.johannes-strommer.com/rechner/basics-mathe-mechanik/haftreibungszahlen/`. (Accessed on 26/04/2017).

[Tec] AIT Austrian Institute Of Technology. Roadstar. `https://www.ait.ac.at/themen/road-condition-monitoring-assessments/strassenzustandserfassung-mit-dem-roadstar/`.

[Typ] Movable Type. Calculate distance and bearing between two latitude/longitude points using haversine formula in javascript. `https://www.movable-type.co.uk/scripts/latlong.html`. (Accessed on 02/11/2017).

[Ung12] Dipl. Ing. Thomas Unger. Junge fahrer 2012. *Berichte der ADAC Unfallforschung*, 2012.

[VCÖ16] VCÖ. Österreichs autofahrer fahren im schnitt 34 kilometer pro tag. `https://www.vcoe.at/news/details/vcoe-oesterreichs-autofahrer-fahren-im-schnitt-34-kilometer-pro-`
Feb 2016.

[Vor15] William Vorhies. Stream processing – what is it and who needs it - data science central. `http://www.datasciencecentral.com/profiles/blogs/stream-processing-what-is-it-and-who-needs-it`, 10 2015. (Accessed on 04/22/2017).

[VSDTD12] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36. ACM, 2012.

[Wik] Wikipedia. Mobile edge computing - wikipedia. `https://en.wikipedia.org/wiki/Mobile_edge_computing`. (Accessed on 03/24/2017).

[WSH15] Johan Wahlstrom, Isaac Skog, and Peter Handel. Detection of dangerous cornering in gnss-data-driven insurance telematics. *IEEE Transactions on Intelligent Transportation Systems*, 16(6):3073–3083, Dec 2015.

[WZZ93] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(3):75–86, July 1993.

[YMdOB$^+$12] Chuang-Wen You, Martha Montes-de Oca, Thomas J. Bao, Nicholas D. Lane, Hong Lu, Giuseppe Cardone, Lorenzo Torresani, and Andrew T. Campbell. Carsafe. *Proceedings of the 2012 ACM Conference on Ubiquitous Computing - UbiComp '12*, 2012.

[Zan09] Paul A Zandbergen. Accuracy of iphone locations: A comparison of assisted gps, wifi and cellular positioning. *Transactions in GIS*, 13:5–25, 2009.

[ZB11] Paul A. Zandbergen and Sean J. Barbeau. Positional accuracy of assisted gps data from high-sensitivity gps-enabled mobile phones. *Journal of Navigation*, 64(3):381–399, 2011.