UNIVERSITY OF OSLO

Master's thesis

Can large language models make our roads safer?

Utilising large language models to decrease driveability in autonomous driving system simulator scenarios

Oliver Ruste Jahren

Informatics: Programming and Systems architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Fall 2025

Draft of October 1, 2025 (commit: 74c5780)

Oliver Ruste Jahren

Can large language models make our roads safer?

Utilising large language models to decrease driveability in autonomous driving system simulator scenarios

Supervisors:
Shaukat Ali
Karoline Nylænder

Simula Research Laboratory

Abstract

Autonomous driving systems (ADSs) rely on extensive testing in order to verify their operational safety. But due to their nature of being able to operate in any unseen environment with arbitrary external actors, the number of potential scenarios is infinite. It is therefore important to obtain the least driveable scenarios for simulator testing ahead of real world deployment. We therefore propose applying Large Language Models to Autonomous driving system scenario files to decrease their driveability, exposing potential underlying issues in the ADS being tested in advance of it happening during real world operation, avoiding causing severe damage to its operator and/or other external actors.

Sammendrag

For å kunne få selvkjørende biler ut på veiene, må vi være sikre på at de er trygge. Trygge både for seg selv, sjåføren, og andre traffikanter. Men det ligger i en bils natur at den skal kunne brukes overalt, med alle mulige folk inne i bildet. Derfor er det teoretisk umulig å forutse alle mulige situasjoner og teste disse i forkant. På bakgrunn av dette fremmer vi i dette arbeidet en metode for å ta ibruk KI til å gjøre dagens testscenarioer *mer utrygge* enn hva de allerede er. Å teste med disse forespeiles å ville kunne avdekke potensielle underliggende feil i bilens systemer, slik at de kan rettes før den volder skade ute i verden.

Contents

List of Figures

Listings

Preface

Here comes your preface, including acknowledgments and thanks.

Part I

Introduction

Chapter 1

Motivation

Conventional cars are ubiquitous in society. Whether for freight trafficking or for humans, cars have great flexibility with their ability to go wherever without requiring tailored infrastructure such as railway tracks. They do, however, have one major weak point — the human driver. For this reason, industry and academia have put forward efforts to enhancing cars with Autonomous driving system (ADS) capabilities. By **empowering humans** with autonomous vehicles, it is expected that traffic efficiency will increase and road fatalities will fall.

Due to the critical safety situation of manoeuvring a car In a public setting where other external actors are present, it is essential that Autonomous driving systems are thoroughly tested before they are deployed so that they are confirmed to be sufficiently safe and capable of handling the situations in which they may typically end up. But due to the complicated nature of the typical ADS operating environment, coming up with exhaustive system test solutions is near impossible. For this reason we want a way of testing the system that is capable of pushing the Autonomous driving system to its limits such that we can measure its performance and see if it is capable of handling complex scenarios.

Having an existing repository of Autonomous driving system test cases, such as DeepScenario we wish to improve them. **Large Language Models (LLMs)** have demonstrated great capabilities of context learning and emergent abilities, which begs the question of their applicability for ADS testing. There are various methods of testing Autonomous driving system. Can these existing test methods be improved by applying LLM technology to them?

Chapter 2

Background

2.1 Testing

First, we need to establish some basic testing concepts.

2.1.1 Pre- and post-conditions

When running test cases, the concept of *Pre-conditions* refers to certain properties that obtain *before* running a given test case. E.g that the ADS ego vehicle is stationary.

In many ways mirroring pre-conditions, *post-conditions* refers to the properties that obtain *after* having ran a test. E.g. that the ego vehicle will be moving after having performed the test.

2.1.2 Test coverage

Test coverage refers to the what degree the entire system is being tested. The concept can be used to describe both hardware and software test coverage [29, p. 187]. Malaiya et al. posit that hardware-based test coverage is measured in terms of the number of possible faults covered, whereas software-based test coverage is measured in terms of the amount of structural or data-flow units that have been exercised [29, p. 187]. A test case that exercised every single code line of the system would by definition have perfect test coverage.

2.2 Autonomous driving systems (ADSs)

Autonomous driving system (ADS) are systems that enable automotive vehicles to drive autonomously. Due to the typical operating scenarios of a car it is pivotal that the Autonomous driving system maintain a high safety standard. A common way to assert safety is to use simulator based testing [27, p. 1].

2.2.1 Autonomous driving system testing

Testing is essential for assuring Autonomous driving system operative safety [17, p. 163]. Several methods for testing exist, testing various aspects of the Autonomous driving system. An ADS typically exists of several modules, all working together and handling different aspect of the Autonomous driving system.

Huang et al. outline several typical architectures for ADS testing, drawing on traditional software testing traditions outlining how *software testing* can be used

alongside more specialized ADS testing techniques such as *simulation testing* and *X-in-the-loop testing* [17, pp. 163–164].

2.2.2 Autonomous driving system driveability

Driveability is a high-level estimator of the overall driving condition of an ADS, derived from several lower-level sources [16, p. 3140]. It can be used to refer to various aspects of a scene. Guo, Kurup, and Shah discuss the concept further, using the scene definition of Ulbrich et al. as outlined in Section 2.2.7 → p. 6, they describe how driveability can refer both to (1) road conditions, and (2) human driver performance. Guo, Kurup, and Shah go on to give an overview of how driveability can be used to refer to a (3) *driveability map* which divides a map into cells indicating where the ADS expects that it will be able to go, and (4) *object driveability*, which refers to the classification of physical objects in the environment that the ADS expects that it can run over without causing damage to the ego vehicle [16, pp. 3135–3136].

The main method for assessing the driveability of a scene comes form assessing the environment of the scene. Factors such as (1) weather, (2) traffic flow, (3) road condition, and (4) obstacles all play into this. The ADS infers information from observation [16, p. 3136].

They continue to give an overview of various *driveability factors* and their associated difficulties, using a a split between *explicit* and *implicit* factors.

Explicit driveability factors will typically include factors such as **Extreme weather** such as (1) fog, (2) heavy rain, (3) snow, all serving to impair road visibility and causing increased difficulties for vision-based tasks such as road detection and object tracking [16, pp. 3136–3137]. **Illumination** also poses various challenges for typical ADS tasks as a typical ADS will be required to operate in a plethora of scenes with varying degrees of illumination depending on factors such as time of day and location (e.g. if the ADS is operation in a dimly lit tunnel) [16, p. 3137]. The authors highlight how low illumination may serve as an advantage for the ADS as this allows for using the head lights of other vehicles as a feature for detecting them, whereas it make pedestrian detection significantly more challenging [16, p. 3137]. **Road geometry** is another external factor, satisfying our natural intuition that *intersections* and *roundabouts* are more difficult to drive through than straight highways [16, p. 3137].

Implicit driveability factors consist of behaviours and intent of other road users interacting with the autonomous car [16, p. 3138]. This includes the actions of other vehicles such as their (1) overtaking, (2) lane changing, (3) rear-ending, (4) speeding, and (5) failure to obey traffic laws . Guo, Kurup, and Shah call these factors **vehicle behaviours** [16, p. 3138]. Furthermore, **pedestrian behaviours** are also taken into account, noting how pedestrians can sometimes (6) cross the road, (7) be inattentive, or (8) fail to comply with the traffic law [16, p. 3138]. They go on to describe the **driver behaviour** of other drivers pointing out how (9) distraction, and (10) drowsiness can be factors that cause accidents even for ADS-enhanced vehicles due to the other, manual, cars interfering with their operation [16, pp. 3138–3139]. Lastly **motorcyclist/bicyclist behaviours** cause their own source of implicit driveability factors: The models and methods developed for analysing the group's behaviour are far more limited than other groups of road users [16, p. 3139]. Guo, Kurup, and Shah theorise that this comes down to the lack of available datasets that capture and label the trajectories and behaviours of motorcyclists and bicyclists [16, p. 3139], causing potential issues for any ADS that wishes to operate in a shared traffic environment with this group.

2.2.3 Autonomous driving system testing metrics

When evaluating ADS testing, several metrics can be used. What metric to use will depend on what the relevant test is measuring.

Building on what we have learnt about driveability (Section 2.2.2 → p. 4), we take after Guo, Kurup, and Shah and review three metrics for quantifying driveability: (1) scene driveability, (2) collision-based risk, and (3) behaviour-based risk.

Scene driveability refers to how easy a scene is for an ADS to navigate, and the *scene driveability score* refers to how likely the Autonomous driving system is to fail at traversing the scene [16, p. 3140]. It is typically found through and end-to-end approach. Note how this is a metric for *scenes*, without taking into account the performance of any specific ADS.

Collision-based risk comes in two kinds - (1) binary risk indicator, and (2) probabilistic risk indicator. Guo, Kurup, and Shah posit that the prior, binary metric, indicates whether a collision will happen in the near future in a binary 'either-or' sense, whereas the latter yields a probability calculated based on current states, event, choice of hypothesis, future states and damage [16, p. 3140].

Behaviour-based risk estimation also represents a binary classification problem wherein nominal behaviours are learnt from data, and then dangerous behaviours are detected on that. This requires a definition of 'nominal behaviour', which is typically defined on on acceptable speeds, traffic roles, location semantics, weather conditions and/or the level of fatigue of the driver [16, p. 3140]. Furthermore Guo, Kurup, and Shah describe how this metric also allows more than one ADS to be labelled as 'conflicting' or 'not conflicting' [16, p. 3140], representing a ruling on their compatibility. Finally, they note how behaviour-based risk assessment typically focuses on driver behaviours, not taking into account other actors in the scene such as pedestrians or cyclists.

2.2.4 The complexities of ADS testing

As we have seen, ADSs can perform several tasks, in several environments. As such, there are several relevant factors for testing them. It is not feasible to test all potential variations of all potential environments in the real world, meaning that the *test coverage*[1] typically will be low.

Some of the factors that complicate ADS operations are (1) timing, (2) sequence of events, and (3) parameter settings such as the different speeds of various vehicles and other actors.

Park, Yang, and Lim posit that *the concept of complexity exists everywhere, but there is no agreement on one for driving situations* [30, p. 1182]. Therefore they introduce their own concept of Driving situation complexity (DSC), which serves to give a metric of a the complexity of a given driving situation. Their DSC is defined as the output of a mathematical formula taking into account the perplexity and standard deviation of several control variables $\mathcal{M}$ representing the surrounding vehicle's behaviour [30, p. 1182]. Their formula also takes into account the ratio of *V2X*-capable vehicles [30, p. 1182], i.e. the vehicles that are connected and capable of communicating [39, p. 1].

2.2.5 Autonomous driving system simulation

Due to the complexity involved in testing Autonomous driving systems (Section 2.2.4 → p. 5), simulators are typically used for this purpose [27]. While the same

[1] See Test coverage → p. 3

points about not being able to test *all* possible scenarios do remain true for simulator based testing due to the sheer number of factors, using a simulator allows for far greater testing at far lower cost due to the minimal overhead of (1) generating, (2) running, and (3) evaluating the outcome of test cases.

Furthermore, simulators allow for greater flexibility in determining the test scenarios due to not being confined by the physical world that is available to the scientist that wishes to perform the testing. Using a simulator, a Europe-based scientist can test their ADS for North American conditions, or vice-versa.

2.2.6 The ADS simulator jungle

Due to the appeal of running ADS simulation, several contenders exist on the market.

Carla is a widely used ADS simulator [12]. It is implemented using the game engine UnrealEngine [13] and allows for running test cases under various scenarios and collecting their results. Carla is fully open source and is under active development. It has been applied in projects such as KITTI-Carla, which generated a KITTI dataset using Carla [10].

LGSVL is a deprecated simulator from LG [32]. It was used in projects such as DeepScenario [27]. It allowed for running various maps with various vehicles and tracking their data. It was also capable of generating HD maps [2]. DeepScenario is a project similar to this, concerned with testing Autonomous driving systems. Further details about it in are located in Related work → p. 16.

AirSim is Microsoft's offering [34]. It has, like LGSVL, been deprecated. It is also built using UnrealEngine. Unlike the other simulators we have seen, this also focused on autonomous vehicles outside of only cars, such as drones.

2.2.7 Concepts of ADS simulation

Ulbrich et al. draw up an outline for the terms *scene*, *situation*, and *scenario*, that are all concepts widely used in ADS simulation testing.

scene is a term that is used in different manners in various articles [37, p. 982], but Ulbrich et al. propose standardising the definition on *a scene describing a snapshot of the environment including the scenery and dynamic elements, as well as as all actors' and observers' self-representations, and the relationships among those entities* [37, p. 983].

situation is, like *scene*, employed in various fashions. Ulbrich et al. give a background detailing its usage ranging from *"the entirety of circumstances, which are to be considered by a robot for its selection of an appropriate behaviour pattern in a particular moment'*[3], in Wershofen and Graefe [41, p. 3] to Schmidt, Hofmann, and Bouzouraa introducing a distinction between *the true world* in a formal sense, and that being the ground truth upon which a situation is described [33, p. 892].

Ulbrich et al. propose to standardise on the definition of a situation being *the entirety of circumstances, which are to be considered for the selection of an appropriate behaviour pattern at a particular point of time* [37, p. 985].

scenario refers to *'the temporal development between several scenes in a sequence of scenes'*[37, p. 986]. We note how the definition a a scenario utilises that of a scene. Furthermore, Ulbrich et al. hold it to be the case that *'every scenario starts with an initial scene. Actions & events as well as goals & values may be specified to characterize*

[2] https://github.com/lgsvl/simulator?tab=readme-ov-file#introduction

[3] The translation from German is borrowed from Ulbrich et al., [37, p. 984]

this temporal development in a scenario' [37, p. 986], clarifying the distinction between a scenario and a scene.

Lastly they posit that a scenario spans a certain amount of time, whereas a scene has no such temporal aspect to it.

When running a simulation, we refer to the autonomous vehicle that is being simulated as the *ego vehicle* [15].

ADS scenario formats

OpenSCENARIO is a standard developed by the Association for Automation and Measurement Systems (ASAM), which is dedicated to the description of dynamic scenarios [7, p. 651]. Under this format, only the *dynamic* content of the scenario is recorded in the file. The static content is kept in other formats such as OpenDRIVER and OpenCRG [7, p. 652]. The simulator Carla (outlined in Section 2.2.6 → p. 6) supports this standard [7, p. 652].

Another widely popular scenario format is **CommonRoad** [25, p. 4941], first proposed in 2017 [2]. There are tools such as those proposed by Lin, Ratzel, and Althoff that allows for converting OpenSCENARIO scenarios to the CommonRoad format [25, p. 4941].

2.3 Large Language Models (LLMs)

Large Language Models (LLMs) are transformer-based language models that typically contain several hundred billion parameters and are trained on massive text data [43, p. 4]. Base language models, as the name implies, *model language*. They are typically statistical models and an example of Machine learning (ML).

2.3.1 Large Language Model (LLM) architecture

A Large Language Model is a neural network trained on big data [43, p. 3]. They expand on the older statistical language models by training on more data. This gives rise to *emerging abilities* such as in context learning [43, p. 3] (Emergent abilities → p. 8). These older statistical models are also neural networks, but they were impractical to train on large amounts of data. It was not until the seminal paper Attention is all you need [38] that a Google team headed by Vaswani et al. showed how neural networks can be trained in parallel using their new *attention* mechanism. This allowed for using amounts of data that was not technologically practical up until that point, opening the door for later advancements such as ChatGPT [43, p. 9]

Jurafsky and Martin describe how LLMs rely on *pretraining*.

The importance of training data

As a consequence of LLMs being statistical models of a certain input data [43, p. 1], what data the model is trained on is of great importance for the capabilities of the model [43, p. 6]. Zhao et al. give an overview of various LLMs and what kinds of corpora[4] they have been trained on [43, pp. 11–14].

The training data will provide the model with its base understanding of the world, and as such it will dictate (1) what it 'knows', and (2) how we should interact with

[4] A corpus (pl. corpora) refers to a document collection.

it. E.g., if we want to solve problems related to software code, we should employ a model that has been *trained* on software code related topics so that the probability of it predicting correct tokens will be higher. If it has not seen any code during its training it would not have any base 'knowledge' for solving our problem, and its output would be bad. The LLM would however have no way of knowing if its output would be right or wrong, and we could say that it would have *hallucinated*. See General challenges with LLMs → p. 10 for further information about hallucination.

2.3.2 Emergent abilities

Wei et al. outline how *emergent abilities* appear when scaling up language models [40, p. 1]. They define *emergent ability* to refer to abilities that are not present in smaller models, but present in the larger ones[40, p. 1], building on physicist Anderson stating that *Emergence is when quantitative changes in a system result in qualitative changes in behaviour.* [40, p. 2].

Furthermore, they discuss how *few-shot prompting* typically can achieve far superior results for harvesting LLM emergent abilities, whereas one-shot prompting can perform worse than randomized results [40, pp. 3–4].

They continue outlining several approaches for achieving augmented prompting strategies, underlining how (1) multi-step reasoning (2) instruction following (3) program execution, and (4) model calibration all serve as possible ways of increasing LLM performance [40, p. 5].

2.3.3 Intelligence in LLMs

There are three theories on machine intelligence, each serving to explain how they '*think*': (1) stochastic parrot (2) Sapir-Whorf hypothesis, and (3) conceptual blending.

Stochastic parrot

Bender et al. outline how LLMs can *fool* humans as they are trained on ever larger amounts of parameters and data, appearing to be in possession of an intelligence [4, pp. 610–611].

This anticipates the phenomenon of hallucination (Section 2.3.5 → p. 10).

Sapir-Whorf hypothesis

The Sapir-Whorf hypothesis posits that *The structure of anyone's native language strongly influences or fully determines the world-view he will acquire as he learns the language.* [5, p. 128].

We note how this maps to our LLMs, indicating that they will only ever be able to 'know' the data on which they have come into contact with.

Or: **Language** defines the possible room for **thought**.

Conceptual blending

Conceptual blending is a theory on intelligence. It refers to the basic mental operation that leads to new meaning or insight that occurs when one identifies a match between to input mental spaces, to project selectively from those inputs into a new 'blended' mental space [14, pp. 57–58].

This phenomenon explains how we are able to imagine phenomena that logically should not exist such as *land yacht* (Land yacht conceptual blend → p. 9)

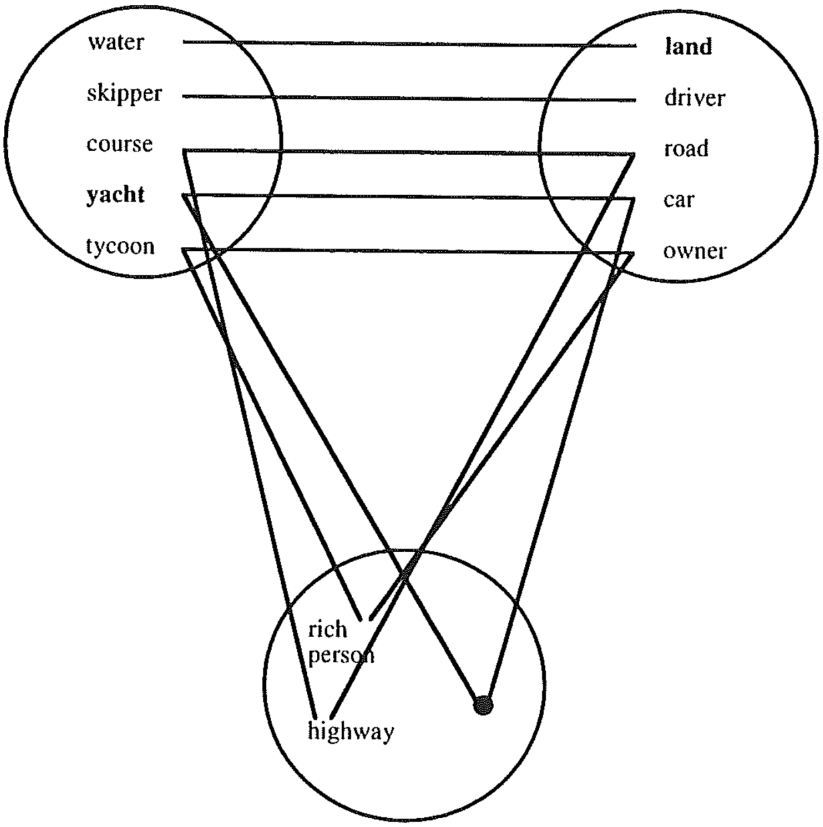

Figure 2.1: The conceptual blend of a *land yacht*[5]

We note how this is how LLMs operate when processing vectorized linguistic data.

2.3.4 Utilising LLMs - Prompt engineering

A typical way of interacting with LLMs is *prompting* [43, p. 44]. You prompt the model to solve various tasks. As we saw in Emergent abilities → p. 8, the level of performance you are able to extract from your Large Language Model can depend a great deal on how you interact with it. The process of manually creating a suitable prompt is called **prompt engineering** [43, p. 44]. Zhao et al. outline three principal prompting approaches:

In-context learning (ICL) is a representative prompting method that formulates the task description and/or demonstrations in natural language text [43, p. 44]. It is based on *tuning-free prompting* and it, as the name implies, never tunes the parameters of the LLM [26, p. 15]. One the one hand, this allows for efficiency, but on the other hand, heavy engineering is typically required to achieve high accuracy, meaning you must provide the LLM with several answered prompts [26, p. 16]. In layman's terms, ICL entails including examples of the process you want the model to perform when prompting it.

Chain-of-Thought (CoT) prompting is proposed to enhance In-context learning by involving a series of intermediate reasoning steps in prompts [43, pp. 44, 52]. The

[5]Diagram borrowed from Fauconnier and Turner, [14, p. 67].

basic concept of CoT prompting, is including an actual Chain-of-Thought inside the prompt that shows the way form the input to the output [43, p. 52]. Zhao et al. note that the same effect can be achieved by including simple instructions like '*Let's think step by step*' and other similar 'magic prompts' in the prompt to the LLM, making CoT prompting easy to use [43, p. 52].

Planning is proposed for solving complex tasks, which first breaks them down into smaller sub-tasks and then generates a plan of action to solve the sub-tasks one by one [43, pp. 44, 54]. The plans are being generated by the LLM itself upon prompting it, and there is a distinction between text-based and code-based approaches. Text-based approaches utilise natural language, whereas code-based approaches utilise executable computer code [43, pp. 54–55].

2.3.5 General challenges with LLMs

We have seen that LLMs demonstrate promising abilities (Emergent abilities →p.8) But they have nevertheless certain issues attached to them that we need to be aware of.

Hallucination

As we saw in Section 2.3.3 →p.8, LLMs are prone to *bullshitting*. They have no intuition of, or concern with *the truth*. They only ever yield whatever response is the most probable under their BEAM SEARCH algorithm being applied on their training data.

Environmental concerns

A University of Rhode Island study on the environmental impact of LLMs have shown that they require wast amount of energy and water [18]. They also found that the different LLMs may differ greatly in their energy consumption, highlighting that that certain LLMs may consume more than 70 times more energy than others [18].

Another study by Tomlinson et al. focusing specifically on *carbon emissions* did however find that these emissions significantly lower for LLMs than humans for specific tasks such as text and image generation, ranging from 130 to 2900 times less Co2 emitted depending on the task [36, p. 1].

Li et al. surveyed the water consumption of LLMs, finding that training the LLM GPT-3 could evaporate as much as 700 000 litres of clean freshwater [23, p. 1]. Furthermore they review the trends of current AI adoption and project that the water consumption of AI could reach levels as high as 4.2 - 6.6 billion cubic metres by 2027, which is comparable to 4 - 6 Denmarks, or half of the United Kingdom [23, p. 1]. Recent research indicates that *serving* LLMs currently account for more emissions than training them [11, p. 37].

Efforts to achieve greener LLMs have been proposed by Li et al., while recognizing the trade-off between ecological sustainability and high-quality outputs [22, p. 21799].

2.3.6 The different kinds of LLMs

There are several available LLMs, some of which are open source, and some proprietary. Open source LLMs afford greater insight into their composition and underlying training data, whereas proprietary models appear more like black boxes. Some popular model families include the GPTs, Gemini, Llama, Claude, Mistral, and DeepSeek.

The LLMs differ primarily in their (1) parameters, and (2) training data. As we saw in Section 2.3.1 →p.7, all typical LLMs utilise a transformer-based neural network.

But due to their various different properties, different models can behave differently for different tasks regardless of their similar architecture.

What they all share is their ability to perform *inference*, meaning that they predict output tokens given some input tokens (see Section 2.3.3 → p. 8).

2.3.7 Existing LLM applications for ADSs

Cui et al. give a broad overview of some of the ways LLMs have been applied for ADSs, highlighting some of the opportunities and potential weaknesses of LLM applications for ADS purposes. One of the ways LLMs can be applied, is for adjusting the driving mode, or aiding in the decision-making process [9, p. 1]. Cui et al. delve further into these aspects in their other work "Drive As You Speak: Enabling Human-Like Interaction With Large Language Models in Autonomous Vehicles", providing a framework for integrating Large Language Model's (1) natural language capabilities, (2) contextual understanding, (3) specialized tool usage, (4) synergizing reasoning, and (5) acting with various modules of the ADS [8, p. 1].

Chapter 3

Problem description

> A problem well stated is a problem half solved.
>
> *Charles F. Kettering*

3.1 Cost

Traditional techniques for obtaining ADS scenarios rely on high skilled manual labour. This incurs a significant cost, and is a major limitation in obtaining a large number of good scenarios, free from the bias of the author

3.2 Impossible to test all scenarios

Furthermore, even if we were to imagine a world in which we had infinite (1) time and (2) money , we would not be able to successfully account for every possible scenario. This is a reality we need to deal with. One possible measure of remedying with this, could be to *decrease* the driveability of our existing scenarios. Decreasing the driveability is not the same as suddenly having access to the infinite set of possible scenarios, but it is reasonable to infer that begin able to *test* the ADS (in a simulator) on these enhanced low-driveability scenarios will leave it better fit for encountering other low-driveability scenarios in the wild during operation.

3.3 Edge cases

Edge cases can be a major issue for ADS adoptation. The *tail problem* as it is known in the ML field posits that ML tasks are faced with a long tail of unseen cases. We can map these unseen cases, to our unseen ADS scenarios. Because of this, an ADS can be at risk of encountering an unseen edge case scenario during operation – something for which it might never have been tested. Arguing that the ADS would probably crash simply due to it finding itself in an unseen scenario is not logical. But it is important to keep in mind that the end we are pursing in the broader adaption of Autonomous driving systems (ADSs), is increased saftey and efficiency on our roads. Not sufficiently testing the ADS before deploying it would not serve our goal of increasing road safety – it would be a gamble with human lives.

Chapter 4

Literature review

TODO: Write literature review

Can move some things from related work such as LLM4AD?

4.1 Graz University of Technology survey on LLM applications for Autonomous driving systems

Zhao et al. give an extensive overview of some of the various ways that LLMs have been applied to scenario based testing of Autonomous driving systems. The authors classify the various research efforts based on (1) how they have employed the LLM, and (2) to what end [44]. Their survey is continually updated, the last update having been made 2 months before the time of writing[1]. This entails a certain overlap with some of the works we review in Related work → p. 16.

Not deterred by this, let us delve into the survey: They start by highlighting the trend between the number of LLM surveys, and ADS surveys – while the trend was increasing from 2020-23, there was an explision in 2024, with about 200 works concering applying LLMs for Autonomous driving system purposes being published [44, p. 1, figure (b)]. Furthermore, the number of ADS studies has remained steady over the last 4 years, wheras the number of LLM studies has exploded in popularity [44, p. 1, figure (a)]. This indicates that a significant amount of the scientific effort around ADSs the last year, has been concerned with utilising LLMs.

4.1.1 Meta survey review

The article summarizes the field, pulling together various surveys of the related subfields. Those being (1) LLM surveys, (2) surveys of scenario-based testing, (3) general cases of LLMs for ADSs, and finally (4) a broader review of surveys of LLMs being applied for *miscellaneous domains* , for each highlighting their specialized foci [44, p. 2].

4.1.2 The categories of ways of applying LLMs for ADS testing

The authors posit that there are 0 major categories of works of LLMs being applied to Autonomous driving systems. They are.

[1] I.e. as of September 17th 2025, the last update to their Github repo was on July 23rd, 2025. The paper on Arxiv was last updated May 22nd 2025.

4.1.3 The 5 key challenges when applying LLMs for ADS testing

Furthermore

Part II

The project

Chapter 5

Related work

> Learn from the mistakes of others.
> You can't live long enough to make
> them all yourself.
>
> E. Roosevelt

5.1 DeepScenario

DeepScenario is both a dataset and a toolset aimed at Autonomous driving system testing [27]. The principal value proposition of this work lies in recognizing the fact that (1) there are an infinite number of possible driving scenarios, and (2) generating critical driving scenarios is very costly with regard to time costs and computational resources [27, p. 52]. The authors therefore propose an open driving scenario of more than 30 000 driving scenarios focusing on ADS testing [27, p. 52]. The project utilises traditional machine learning methodologies, having been performed prior to the broad adaptation of LLMs.

Its scenarios are intended for the simulator SVL by LG (Section 2.2.6 → p. 6).

5.2 RTCM

RTCM is a ADS testing framework that allows the user to utilise natural language for synthesizing test cases. The authors propose a domain-specific language — called RTCM, after Restricted Test Case Modelling — for specifying test cases. It is based on natural language and composed of (1) an easy-to-use template, (2) a set of restriction rules, and (3) keywords [42, p. 397]. Furthermore, they also propose a tool to take this RTCM source code as input and generating either (1) manual, or (2) automatically executable test cases [42, p. 397]. The proposed tools were evaluated in experiments with industry partners, successfully generating executable test cases [42, p. 397].

5.3 DeepCollision

Lu et al. utilise Reinforcement learning (RL) for ADS testing, with the goal of getting the ADS to *collide*. They used *collision probability* for the loss function of the Reinforcement learning algorithm [28, p. 384]. Their experiments included training 4 DeepCollision models, then using (1) random, and (2) greedy models for generating a baseline to

compare their models with. The results showed that DeepCollision demonstrated significantly better effectiveness in obtaining collisions than the baselines. While not specifically focused on *testing*, we recognize that their work is thematically similar to our envisioned project.

5.4 AutoSceneGen

AutoSceneGen is a framework for ADS testing using LLMs, focusing on the motion planning of Autonomous driving system [1, p. 14539]. Aiersilan highlights how LLMs provide opportunities for efficiently evaluating ADS in a cost-effective manner [1, pp. 14539–14540]. They generate a substantial set of synthetic scenarios and experiment with using (1) only synthetic data, (2) only real-world data, and (3) a combination of the 2 as training data. They find that motion planners trained with their synthetic data significantly outperforms those trained solely on real-world data [1, p. 14539].

5.5 LLM4AD

LLM4AD is a paper that gives a broad overview of LLMs for Autonomous driving system. It touches on several of the various ADS applications where LLMs are relevant such as (1) language interaction, (2) contextual understanding, (3) zero-shot and few shot planning allowing LLMs to perform tasks they weren't trained on, helping with handling edge cases (4) continuous learning and personalization, and finally (5) interpretability and trust [9, p. 2]. Furthermore, the authors also propose a comprehensive benchmark for evaluating the instruction-following abilities of an LLM based system in ADS simulation [9, p. 1].

5.6 LLM-Driven testing of ADS

Petrovic et al. worked on using LLMs to for automated test generation based on free-form textual descriptions in the area of automotive [31, p. 173]. They propose a prototype for this purpose and evaluate their proposal for ADS driving feature scenarios in Carla. They used the LLMs GPT-4 and Llama3, finding GPT-4 to outperform Llama3 for the stated purpose. Their findings include this LLM-powered test methodology to be more than 10 times faster than traditional methodologies while reducing cognitive load [31, p. 173].

5.7 Requirements All You Need?

Lebioda et al. provide an overview of LLMs for ADS in their recent preprint *Are requirements really all you need? A case study of LLM-driven configuration code generation for automotive simulations*[1], focusing on LLM's abilities for translating abstract requirements extracted from automotive standards and documents into configuration for Carla (Section 2.2.6 $^{\rightarrow p.6}$) simulations [21]. Their experiments include employing the *autonomous emergency braking* system and the sensors of the ADS. Furthermore, they split the requirements into 3 categories: (1) vehicle descriptions, (2) test case pre-conditions, and (3) test case post-conditions (Pre- and post-conditions $^{\rightarrow p.3}$) [21]. The preconditions they used included (1) agent placement,

[1] This was submitted to Arxiv on 2025-05-19.

(2) desired agent behaviour, and (3) weather conditions amongst others, whereas their postconditions reflected the desired outcomes of the tests, primarily related to the vehicle's telemetry [21].

5.8 Language Conditioned Traffic Generation

Tan et al. look into using LLMs to generate specific traffic scenarios. They identify the importance of being able to use simulators to test ADSs, and highlight how test scenarios are expensieve to obtain [35, p. 1]. To this end, they propose a tool – LTCGen which employs the strengths of LLMs to match a natural language query with a fitting underlying map[2], and populates this with a (1) initial traffic distribiution, and (2) the dynamics of all the vehicles involved in the scene. Something to note is that they generate their scenarios, without initially taking the *ego vehicle* into account. The ego vehicle of the scene is simply determined as the vehicle that is in the *center* of the first *frame* [35, p. 3].

5.9 Scenario engineer GPT

Li et al. outline a framework for utilising the LLM-backed ChatGPT in order to generate scenarios. They propose SeGPT – a scenario generation framework that they found to yield *significant progress in the domain of scenario generation* [24, p. 4422]. They posit that their prompt engineering ensures that the generated scenarios are authentically diverse and challenging [24, p. 4423]. The focus is primarily on *trajectory scenarios* [24, pp. 4422–4423].

Note how they explicitly mention scenario *generation*. Our approach for this project has a different angle, with the focus being on modifying *existing* scenarios. More on this in Proposed solution → p. 20. The difference between generating a 'brand new' scenario with a model trained on exisiting scenarios, and modifying an existing scenario seems like a matter of granularity. These are very similar concepts, only that the enhanced scenario will have more common DNA whereas the other 'new' scenario will consist of a broader range of DNA from its various underlying scenario corpora.

5.10 LLM driven scenario generation

Chang et al. also look into using Large Language Models to generate ADS scenarios. They recognize several of the challenges we outline in Chapter 3 → p. 12. In their 2024 paper, they propose LLMScenario, which is an LLM-backed framework for both (1) scenario generation, and (2) evaluation feedback tuning [6, p. 6581].

They analyze scenarios in order to provide the LLM with a minimum baseline scenario description, and propose score functions based on both (1) reality and (2) rarity. Their prompting is based on Chain-of-Thought (CoT) and a posteriori emperical experience. Lastly, they tested several Large Language Models for their experiments. Their results were positive, indicating effectiveness for scenaro engineering in Industry 5.0 [6, p. 6581].

[2] Map as in a *world* in which a scenario can take place.

5.11 Chat2Scenario

Zhao et al. propose a method for utilising LLMs to retrieve ADS scenarios given a natural language query. Their framework synthesizes scenarios from naturalistic[3] driving datasets, based on observation real world human driving [45, p. 55], that it then uses as a database for retrieveing the scenario that best matches the user's natural language query. Furthermore, they employ traditional techniques for asserting the relevance of the retrieved scenarios, allowing the user to specify a set of *criticality metrics*, of which a certain threshold must be reached amongst the scenarios that are initalliy retried by the LLM, pruning false positives. As a measure to increase the usability of their framework, they also provide a webapp with an intuitive GUI for both (1) operating the tool, and (2) visualizing the scenarios [45, p. 560].

In order to allow the LLM to determine whether a scenario is relevant under the provided query, they put forward a method for classifying the various scenarios using traditional ML techniques. This classification focuses primarily on highway scenarios and the activities of other actors in relation to the ego vehicle [45, pp. 561–562].

Prompt engineering

The project's prompts are 'informed' by the 6 OpenAI guidelines from their prompt engineering guide[4], ending up with a structured prompt of 5 segments. These segments serve to guide the LLM, delineating its role as an 'advanced AI tool for scenario analysis, specifically tasked with interpreting driving scenario following a pre-established classification model' [45, p. 562]. They then input the user-provided description of the scenario they wish to retrieve. Following this, a third segment declares the format for the LLM response, followed by a prime example of In-context learning, demonstrating what a satisfactory fulfillment of the desired format could look like. Lastly they instruct the LLM to *Remember to analyze carefully and provide the classification as per the structure given above* [45, p. 563].

[3] Their term. The intended meaning of *naturalistic* is not all clar to me.

[4] https://platform.openai.com/docs/guides/prompt-engineering (URL from the paper.)

Chapter 6

Proposed solution

We have seen that ADS testing is *complex* and that it is difficult to get a good test coverage (Section 2.2.4 → p. 5). Furthermore, we have seen that LLMs have *emergent abilities* (Section 2.3.2 → p. 8). We therefore propose a tool for (1) running a base ADS test case, (2) enhancing the test case using LLMs, (3) running the enhanced test case, and (4) comparing the results of the two runs.

This will allow us to learn the extent to which LLMs can be applied for enhancing Autonomous driving system test cases. We will survey several LLMs and evaluate their applicability for the problem at hand, in light of what we know about LLMs (The different kinds of LLMs → p. 10). We want to have a pipeline that is able to process several test cases in succession, in order to get a substantial dataset.

Let the pipeline tool be known as HEFE. The tool follows a natural pipeline structure. We have some base test cases that need to be ran in order to get a baseline for the results, we then have to improve these, and run the improved versions and compare them to their original versions. The architecture of the tool is visualised in Figure 6.1 → p. 20.

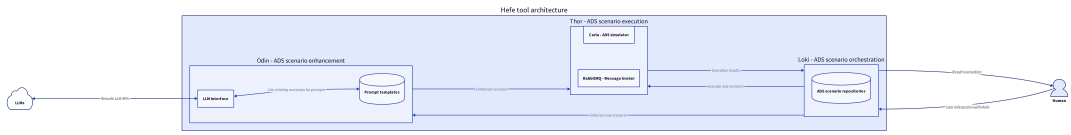

Figure 6.1: HEFE pipeline architecture

We need to define what requirement we will use for determining the *result* of a test case run. Without this, we cannot compare it to other test cases.

Furthermore, as outlined in Cui et al., Large Language Models can be applied to several aspects of Autonomous driving system. It is not feasible that we focus on *all* these aspects, and as such we should narrow down our scope. Let us review some of the relevant aspects.

The applicability of LLMs in ADS testing

Autonomous driving system are typically modular, as we have seen in Section 2.2.4 → p. 5. LLMs are applicable to the different modules in different ways as we saw in Related work → p. 16.

User history of using HEFE

I have a set of Autonomous driving system (ADS) test cases. I provide this set to HEFE. It will run the entire set, and generate a baseline of my ADS performance. HEFE will then improve my test cases using Large Language Models and run them again.
Lastly HEFE will report how the results differ from running the base and enhanced version of a test case.
This will give me insight into what caused my ADS to fail so that I can look into the cause of the error state and uncover underlying faults in the Autonomous driving system.

6.1 Implementation language

The programming language PYTHON is widely used for Autonomous driving system (ADS) simulation. It is a high level language, allowing the user great flexibility and developer experience. For this reason, I will implement HEFE using Python.

Python can be optimized using Just-In-Time (JIT) compilers such as Numba [20], which can speed up our execution times. Libraries such as Joblib provide Python with plug-and-play meomization, which will allow us to re-use values that have already been computed, saving time and energy.

6.1.1 The room for concurrency

When evaluating ADS test cases, the test cases are independent of each other. This means that our problem is *embarrassingly parallelizable* [1] and we can trivially process several test cases in parallel. Due to practical limitations in Carla, *running* the test cases should however probably be done sequentially. But (1) prompting, (2) enhancing, and (3) validating, can all be done concurrently. While Python lacks support of traditional threads, it has some support for multiprocessing [2].

6.2 Overview of the components of the HEFE pipeline

The pipeline architecture is visualised in HEFE pipeline architecture → p. 20. Here we present the major components and their responsibilities

6.2.1 Test case enhancement

Test case repositories

We have seen in Related work → p. 16 that there are existing repositories of ADS test cases. These will provide us with (1) a baseline, and (2) data onto which we can apply our LLM enhancements.

[1] https://en.wikipedia.org/wiki/Embarrassingly_parallel
[2] https://docs.python.org/3/library/multiprocessing.html

LLM enhancement

The base test cases will individually be enhanced by prompting the LLM. We will experiment with several LLMs.

For performing the actual improvement, it is essential that we (1) test several LLM, (2) give clear prompts and (3) verify that the returned test case adheres to the strictly necessary syntax rules. This last point is important due to our knowledge of LLMs hallucinating (see General challenges with LLMs →p.10).

In order to facilitate testing various Large Language Models, we should employ LLM agnostic software as a translation layer. This will allow us to write code for a common interface and test several LLMs that may all have different internal Application programming interfaces (APIs) without having to modify our test code for specific APIs. This (1) saves time and (2) makes for more even test conditions . Some pieces of software providing this type of functionality include AISUITE[3], RamaLama from RedHat[4], and the MIT licensed Ollama[5], both supporting a plethora of Large Language Models.

GUIDANCE[6] is a framework for limiting the room in which LLMs may operate, which might be useful if we run into issues with excessive hallucination.

Enhanced test case validation

We must expect the LLM to hallucinate to some extent (Section 2.3.5 →p.10). We therefore propose to verify the format of the enhanced file before running it.

As we saw in the section for ADS scenario formats →p.7, there exists several formats for ADS scenarios. In order to verify that the syntax of our enhanced test case is valid, we simply need to apply the syntax rules of our format.

The CommonRoad format is XML-based [2, p. 720] and as such we can to some extent assess the degree of hallucination by parsing the XML structure. Furthermore, it has an exhaustive Python library with several utilities[7].

OpenSCENARIO exists both as XML and a domain-specific language (DSL). If we utilise the XML version, we can apply the same methodology as for the CommonRoad format. If using the DSL version, one way the OpenSCENARIO format can be verified is by using free online cloud services such as this offering from AVL [8]. We should however strive for running a local verification service to (1) save time and compute, and (2) preserve data privacy. Besides, it is generally a good idea to limit the number of external dependencies[9].

6.2.2 Test case running and evaluation

Test case runner

The system will automatically run all our base test cases using an ADS simulator, and collect data points to get a baseline. It will later also run the mutated LLM-enhanced versions of the base cases.

[3] https://github.com/andrewyng/aisuite
[4] https://github.com/containers/ramalama
[5] https://github.com/ollama/ollama
[6] https://github.com/guidance-ai/guidance
[7] https://pypi.org/user/commonroad/
[8] https://smc.app.avl.com/validation
[9] Note for example how LGSVL[32] was shut down, preventing projects such as DeepScenario of Lu, Yue, and Ali to be further developed on the original platform.

We have already ran the test cases in their base form. We will now run their improved versions in order to compare them to see what effect the LLM enhancement (see Section 6.2.1 → p. 22) has had.

For the reasons we have seen in Section 2.2.6 → p. 6, we want to run our test cases on Carla. It is the best offering as it is open source, under active development and has a feature rich Python API.

Test case improvement evaluation

We saw in Section 2.2.3 → p. 5 that there are several metrics for assessing ADS. We will use these metrics when evaluating our improvements.

Test case result reporting

We will compare the results from running the baseline unmodified test case and comparing it with the results from running the LLM-enhanced version and returning to the user. Ideally with some automatic analysis of the results.

Having ran both the base test case and its enhanced counterpart, we have results. The results will be stored in Comma separated values (CSV) files, allowing (1) further analysis in Python/Jupyter, and (2) easy translation to LaTeXtables for the final report.

This is the final step of the envisioned pipeline. Where we have our result, and need to analyse them.

This last step has great opportunities for being scoped up to a fully integrated test suite which allows for both running test cases and analysing the results in a Graphical user interface (GUI). But we should focus on the prior steps for now, only creating a GUI if there is sufficient time towards the end of the project to focus on such non-LLM related topics.

Initially, the results will consist of numerical comparison of the CSVs with regard to the relevant metrics outlined in Test case improvement evaluation → p. 23.

Chapter 7

Implementation details

The implementation is what facilitates doing the actual experiments. For the most part, it follows what is outlined in the Proposed solution → p. 20, with some minor practical differences. What follows will analyze the impelementation of the components of the HEFE pipeline and explain more closely in detail not only *what* they do, as that is already covered in the solution proposal, but *how* they do it, with hands-on code examples.

All code is available on the Github repo `master-hefe`.

7.1 Carla interface and scenario utilities – Thor

The Thor module is responsible for all thigs related to the Carla ADS simulator. It provides the client with several scenario-related utilities, and is capable of executing the desired scenarios.

Certain of its utilities are simple tools for asserting the liveness of Carla, such as the `get_carla_is_up` function, shown in listing 7.1. This function will use the Caral standard Python library and attempt to connect to the server on its default port[1]. Note that we refer to the host as simpyly `carla` – this is possible due to the entire project running containerised with Docker Compose. Instead of refering to the speicifc IP address of the Carla server (typically localhost, if not running it externally), the Docker system will facilitate this name translation for us.

```
1  import carla
2
3  CARLA_HOST = "carla"
4  CARLA_PORT = 2000
5
6
7  def get_carla_is_up() -> bool:
8      """
9      Check if the CARLA simulator is up and running.
10     This function attempts to connect to the CARLA server and returns
       True if successful, otherwise False.
11     """
12     print("Running carla integration check to see if it is up...")
13     try:
14         client = carla.Client(CARLA_HOST, CARLA_PORT)
15         client.get_world()
16         return True
17     except Exception as e:
18         print(f"CARLA connection failed: {e}")
```

[1] I.e. 2000, line #4 in listing 7.1.

```
19          return False
```

Listing 7.1: Exerpt from carla_interface.py, demonstrating the implementation of a Carla health check.

This is used both to assert the general liveness of the HEFEpipeline, and to verify that the simulator is available before performing experiments. It is better to detect this illegal state *before* running experiments rather than during their execution.

Furthermore, it shall also be equipped with funcitonality for *executing* ADS scenarios on Carla. This is trivial when using Carla's existing Scenario Runner module's funcitonality. As of now, this has not yet been implemented due to greater challenges in the LLM module – Odin.

7.2 LLM interface and prompt applications – Odin

The Odin module handles all things LLM. It provides a unified API for applying various prompts to scenarios and returning the enhanced output resulting from having applied the prompt. We hve implemented support for the LLMs that are available on (1) Ollama, and (2) Gemini . This allows for testing with LLMs such as (3) Mistral 7.2B, and (4) gemini-2.5-flash .

7.2.1 LLM interface implementations

Gemini integration

The Gemini integration is quite straightforward, relying on Google's own `genai` Python module. Listing 7.2 renders the *entire* interface, again highlighting how straightforward this really is. The one piece of complexity to not is that it requires that the user provides their own Gemini API key and has this set as an environment variable with the proper name. Without this being as it should, the script will crash, as it would not possible for it to complete the desired LLM enhancement regardless as long as the API key is not present.

```
1  import os
2
3
4  from google import genai
5
6
7  def get_api_key() -> str:
8      api_key = os.getenv("GEMINI_API_KEY")
9      if not api_key:
10         raise EnvironmentError("GEMINI_API_KEY environment variable not
       set.")
11     return api_key
12
13
14 client = genai.Client(api_key=get_api_key())
15
16
17 def api_is_up():
18     return True  # Assume Google never dies...
19
20
21 # TODO: Use decorator for asserting API liveness? Or standard assertion??
22 def execute_gemini_model(model_name: str, prompt: str) -> str:
```

```
23
24      response = client.models.generate_content(
25          model=model_name or "gemini-2.5-flash",
26          contents=prompt
27      )
28
29      return response.text
```

Listing 7.2: llm_api_interfaces/gemini_interface.py, The implementation of a Gemini interface for executing prompts.

Ollama integration

The Ollama integration is a bit more cumbersome. This motly comes down to it not using any exisitng library modules for this specific purpose, instead relying on using the `json` and `requests` modules to implement the desired funcitonality from scratch, making it so that we need to handle network IO and marshalling the Large Language Model (LLM) response into a fitting return buffer.

Listing 7.3 renders the *entire* interface. As we can see, it is not too bad allthough nowhere near as clean as the Gemini implementation (7.2).

Its complexity arises principally from 2 major factors – (1) the already mentioned manual networking, and (2) having to parse the streamed response Furthermore, this code expects that the user already *has* an Ollama installation running on their host machine. The code provides no means of setup for this – that is an entirely external endeavour that is left up to the end user.

Similarly to how the Gemini implementation does it, this will crash if Ollama is not funcitoning properly as it would not possible for it to complete the desired LLM enhancement regardless if Ollama is unreachable.

```
1  import json
2  import requests
3
4
5  OLLAMA_API_URL = "http://localhost:11434"
6
7
8  def api_is_up():
9      try:
10         response = requests.get(OLLAMA_API_URL)
11         return response.status_code == 200
12     except requests.ConnectionError:
13         return False
14
15
16 # ollama models
17 def get_ollama_models():
18     try:
19         response = requests.get(f"{OLLAMA_API_URL}/api/tags")
20         if response.status_code == 200:
21             return response.json()["models"]
22         else:
23             print(f"Failed to get models: {response.status_code}")
24             return []
25     except requests.ConnectionError:
26         print("Failed to connect to the API.")
27         return []
28
```

```python

# TODO: Use decorator for asserting API liveness? Or standard assertion??
def execute_ollama_model(model_name: str, prompt: str):
    try:
        payload = {
            "model": model_name,
            "prompt": prompt
        }
        print(f"Executing model {model_name} with prompt: {prompt}")
        response = requests.post(
            f"{OLLAMA_API_URL}/api/generate", json=payload)
        if response.status_code == 200:
            result = ""
            for line in response.iter_lines():
                if line:
                    data = line.decode('utf-8')
                    try:
                        json_obj = json.loads(data)
                        result += json_obj.get("response", "")
                    except Exception as e:
                        print(f"Failed to parse line: {e}")
            return {"text": result}
        else:
            print(f"Failed to execute model: {response.status_code}")
            return None
    except requests.ConnectionError:
        print("Failed to connect to the API.")
        return None
```

Listing 7.3: llm_api_interfaces/ollama.py, The implementation of an Ollama interface for executing prompts.

7.2.2 Prompts and their associated code

In this project, the prompts are the instruction to the Large Language Model (LLM) for applying the enhancement to the scenario. Quite possibly the most critical piece of code related to the experiments. They need to take the base scenario as an input and integrate it into the LLM context, such that it knows what it shall use as its base to apply enhancements that will decrease the driveability. For this reason, it also provides certain scenario utilities[2].

Scenario utilities

These are essentially quite trivial helpers. Lisitng 7.4 render the core functionality – hopefully this is quite self-explaining.

```python
import os


# TODO: Implementer denne
# TODO: Fastslaa hvilket format vi bruker (OpenSCENARIO/CommonRoad/andre)
def file_format_is_valid(file_format: str) -> bool:
    """
    Check if the file format is valid.

    Args:
        file_format (str): The file format to check.
```

[2]That architectually might as well have been integrated in the Thor module...

```
12
13      Returns:
14          bool: True if the file format is valid, False otherwise.
15      """
16      return file_format in ["json", "yaml", "yml", "csv", "txt"]
17
18
19  def enumerate_enhanced_scenarios(scenario_repository_path: str,
    scenario_name: str) -> int:
20      """
21      Enumerate enhanced scenarios in a given scenario path.
22
23      Args:
24          scenario_repository_path (str): The path to the scenario
    directory.
25          scenario_name (str): The name of the scenario.
26
27      Returns:
28          int: The number of enhanced scenarios found.
29      """
30      # TODO: Can probably refactor this
31      acc = 0
32      for scenario in os.listdir(scenario_repository_path):
33          print(f"Checking scenario: {scenario}")
34          if scenario_name in scenario and "enhanced" in scenario:
35              acc += 1
36      return acc
37
38
39  # TODO: Should use better names. Need a way of tracking enhanced scenario
40  # metadata
41  # - Timestamp
42  # - What prompt was used
43  # - What model was used
44  # - What the original scenario was
45  # - What changes were made?
46  def get_enhanced_scenario_name(scenario_repository_path: str,
    scenario_name: str) -> str:
47      """
48      Get the enhanced scenario name.
49
50      Args:
51          scenario_repository_path (str): The path to the scenario
    directory.
52          scenario_name (str): The base name of the scenario.
53
54      Returns:
55          str: The enhanced scenario name.
56      """
57      num_enhanced_scenarios = enumerate_enhanced_scenarios(
58          scenario_repository_path, scenario_name)
59      if num_enhanced_scenarios == 0:
60          return f"{scenario_name}-enhanced"
61      else:
62          # Big brain time...who needs UUIDs when you can just count files?
63          return f"{scenario_name}-enhanced-{num_enhanced_scenarios + 1}"
64
65
66  def get_available_scenarios(scenario_repository_path: str) -> list:
67      def extension_is_ok(filename: str) -> bool:
68          # TODO: Verify which formats we want to support
```

```
69          return filename in ["xosc", "py"]
70
71      return [filename for filename in os.listdir(scenario_repository_path)
         if extension_is_ok(filename)]
72
73
74  # TODO: Should strip newlines??
75  def scenario_path_to_string(scenario_path: str) -> str:
76      with open(scenario_path, 'r') as file:
77          return file.read()
78
79
80  # TODO: Let this function determine output file name?
81  def save_enhanced_scenario(scenario_str: str, output_path: str):
82      with open(output_path, 'w') as file:
83          file.write(scenario_str)
```

Listing 7.4: scenario_utils.py, The implementation of an various scenaro helper functions for executing prompts.

Prompts – templating and usage

As mentioned, the prompts need to include the scenarios *in* them, so that they are accessible to the LLM. How this is done, is rendered in listing 7.5. The most interesting aspect is how the prompts are stored in the system as lambda functions. This makes it so that they can take an argument that represents the scenario – `python_carla_scenario_raw`(line #16) – and simply *execute* the function to insert the scenario into the prompt (line #42). This is then inserted into the output prompt at the location located at line #21 in the listing.

```
1  # We wish to decrease the driveability of the scenario by enhancing it
       with more
2  # details, increasing its complexity
3
4  # Prompt structure:
5  """
6  1 - Context: We are working with a driving simulation environment for the
        Carla simulator.
7  2 - Task: Decrease the driveability of the scenario by enhancing it with
       more details and complexity.
8  3 - Input: <scenario_description, in python carla scenario format>
9  4 - Output: An enhanced version of the scenario description with
       additional
10 details and complexity, still in Python carla scenario format. ONLY
       output the
11 code, without any additional text or explanation.
12 """
13
14 PROMPTS = [
15     [...] # NOTE: Removed most prompts from this listing for brevity.
16     lambda python_carla_scenario_raw: f"""
17     1 - Context: You are a tool for decreasing the driveability of
       scenarios in the driving simulator Carla.
18     2 - Task: Decrease the driveability of the scenario by enhancing it
       with
19     more details and complexity, using only methods that are part of the
20     official Carla API, version 0.9.15.
21     3 - Input, the Python specification for the scenario: {
       python_carla_scenario_raw}
```

```
    4 - Reasoning: Think step by step about how to make the scenario more complex and less driveable, considering possible obstacles, traffic, weather, and other factors using only the official Carla API.
    5 - Output: Only output the enhanced scenario code in Python Carla scenario format, with no additional text or explanation.
    """,
]


def name_to_prompt_idx(name: str) -> int:
    mapping = {
        "basic": 0,
        "no_explanation": 1,
        "no_explanation_strict": 2,
        "cot": 3,
        "cot_strict_methods_in_file": 4,
        "cot_strict_carla_api": 5,
    }
    return mapping.get(name, 0)


def get_prompt_for_python_scenario_enhancement(python_carla_scenario_raw: str, prompt_name: str) -> str:
    prompt_idx = name_to_prompt_idx(prompt_name)
    return PROMPTS[prompt_idx](python_carla_scenario_raw)
```

Listing 7.5: experiments/testbed/prompts.py, The implementation of a prompt testbed for executing prompts.

Lastly, note the comments in the top of the file, intended to give Github Copilot increased understanding of the context, so that it can provide better aid during programming.

7.3 Execution tool / user oriented frontend – Loki

The final module of the HEFEpipeline is Loki – it is simply a tool intended to be used by the user for operating the process. It (1) says what scenarios are available to it (i.e. those that are eligble for being enhanced), and (2) allows the user to select a prompt and (3) execute that prompt to the scenario of their choosing.

```
import pika
import requests

# from odin.server import ODIN_PORT
# from thor.server import THOR_PORT

ODIN_PORT = 4000
THOR_PORT = 6000


# TODO: Merge health checks into a single function?
def do_thor_health_check():
    try:
        res = requests.get(f"http://localhost:{THOR_PORT}/health")
    except requests.ConnectionError:
        print("Thor server is not running or unreachable.")
        exit(1)
    if res.status_code == 200:
        # parse json
```

```
        health_status = res.json().get("status", "unknown")
        if health_status == "healthy":
            print("Thor is healthy.")
        else:
            print(f"Thor health check failed.: {health_status}")
            exit(1)
    else:
        print("Thor health check failed.")
        exit(1)


def do_odin_health_check():
    try:
        res = requests.get(f"http://localhost:{ODIN_PORT}/health")
    except requests.ConnectionError:
        print("Odin server is not running or unreachable.")
        exit(1)
    if res.status_code == 200:
        health_status = res.json().get("status", "unknown")
        if health_status == "healthy":
            print("Odin is healthy.")
        else:
            print(f"Odin health check failed.: {health_status}")
            exit(1)
    else:
        print("Odin health check failed.")
        exit(1)


def run_test_case(test_case_id):
    print(f"Running test case: {test_case_id}")

    # Run test case on Loki
    result = requests.post(
        f"http://localhost:{THOR_PORT}/run_test_case",
        json={"test_case_id": test_case_id})
    if result.status_code != 200:
        print(f"Failed to run test case: {test_case_id}")
        return None
    print(f"Test case {test_case_id} executed successfully.")
    value = result.json().get("result", "No result found")
    return value


def get_enhanced_test_case(test_case_id):
    print(f"Enhancing test case: {test_case_id}")

    # Enhance test case on Odin
    result = requests.post(
        f"http://localhost:{ODIN_PORT}/enhance_test_case",
        json={"test_case_id": test_case_id})
    if result.status_code != 200:
        print(f"Failed to enhance test case: {test_case_id}")
        return None
    print(f"Test case {test_case_id} enhanced successfully.")
    value = result.json().get("result", "No result found")
    return value


def get_improvement(base_test_case, enhanced_test_case):
    # Simulate getting an improvement between two test cases
```

```
81      # TODO: Implement actual logic to compare test cases
82      return f"Improvement from {base_test_case} to {enhanced_test_case}"
83
84
85  def send_test_message(message):
86      # TODO: Implement proper credentail handling.
87      credentials = pika.PlainCredentials('user', 'pass')
88      connection = pika.BlockingConnection(
89          pika.ConnectionParameters('localhost', credentials=credentials))
90      channel = connection.channel()
91      channel.queue_declare(queue='test_queue')
92      channel.basic_publish(exchange='', routing_key='test_queue', body=
        message)
93      print(f"Sent message to RabbitMQ: {message}")
94      connection.close()
95
96
97  if __name__ == "__main__":
98      print("Loki is running...")
99
100     do_thor_health_check()
101     do_odin_health_check()
102
103     send_test_message("Hello from Loki!")
104
105     exit(0)
106     test_case_id = "test_case_123"
107
108     print("Starting test case execution...")
109     base_result = run_test_case(test_case_id)
110     print(f"Base result: {base_result}")
111
112     enhanced_test_case = get_enhanced_test_case(test_case_id)
113
114     enhanced_test_case_result = run_test_case(enhanced_test_case)
115     print(
116         f"Enhanced test case execution completed with result: {
        enhanced_test_case_result}")
117
118     improvement = get_improvement(base_result, enhanced_test_case_result)
119     print(f"Improvement: {improvement}")
120
121     print("Loki execution finished.")
```

Listing 7.6: loki/main.py, The implementation of the Loki script.

Listing 7.6 renders the implementation of the script. It relies on the Odin and Thor modules for all essential functionality, which is in line with what is to be expected as this is simply a frontend client to *reach* them.

It is relies on the `requests` module for doing Remote procedure call (RPC) to the other modules. There is also the outlines of a RabbitMQ implementation, which is why `pika` is being imported. As of now, this is in non-functioning alpha. Implementation of RabbitMQ message passing has not been prioritized as there were, as mentioned, more important issues to focus on that would yield better and more important results when resolved. This would maintain feature parity with the `requests`-based approach.

Chapter 8

Experiment methodology

> The torment of precautions often exceeds the dangers to be avoided. It is sometimes better to abandon one's self to destiny.
>
> Napoléon

8.1 Prompts

Prompting is our principal way of interfacing with the LLM. For this reason, our results rely on (1) good, and (2) fitting prompts . Without this all is lost.

We therefore propose several prompting strategies, taking after related research (Related work →p.16).

Prompts were determined by trial and error in an iterative manner, in conjunction with Github Copoilot. They are all descendant of listing 8.1, each subsequent iteration improving on the last based on what worked or did not worked when assessing the output. Due to a techical detail of the HEFE implementation (LLM interface and prompt applications – Odin →p.25), the datype of the prompt is a lambda function that takes the raw scenario represented as a string and then inserts it into the prompt in runtime. This is represented by the curly braces on line 3 in listing 8.1.

```
1 lambda python_carla_scenario_raw: f"""
2 1 - Context: We are working with a driving simulation environment for the
      Carla simulator.
3 2 - Task: Decrease the driveability of the scenario by enhancing it with
     more details and complexity.
4 3 - Input: {python_carla_scenario_raw}
5 4 - Output: An enhanced version of the scenario description with
     additional
6 details and complexity, still in Python carla scenario format.
7 """,
```

Listing 8.1: The first prompt.

8.2 Trying different LLMs

As we learnt in Section 2.3.6 →p.10, there are several LLMs extant. We should experiment with various different LLMs to maximize our chance of testing with a 'good' LLM that

goes well with our stated purpose.

The results were first carried out using a locally hosted 7.2B parameter Mistral model. Later, a Gemini model running on Google's infrastrucutre was used.

8.3 Metrics

The way the Carla simulator works, one simulator run can be analyzed post factum. The entire scenario execution is stored in a Carla-specific binary format. This binary file can then later be analyzed, extracting various metrics from one run. This saves time not having to run the simulator more than necessary, and allows for reproducing the metric calculations from the original underlying binary log file.

Due to the immense file size of these logs[1], publishing all our raw files is not feasible.

[1]Keep in mind that they track all actors in the scene over time.

Part III

Conclusion

Chapter 9

Results

> I have not failed. I've just found 10,000 ways that won't work.
>
> *Thomas A. Edison*

Our results show that the initially proposed solution of feeding bare ADS scenarios represented by Python code into LLMs, does not yield any meaningful results. This is caused by various reaons. The following discusses (1) why this is, and (2) ways by which it can be remedied in future work .

See listing A.1 in the Scenario file diffs → p. 48 appendix for a complete demonstration of what the Large Language Model is capable of doing.

9.1 Output of the LLM

Depending on the prompt, our results show that it *is* possible to get reasonable-looking Python out of the LLM. One somewhat annoying detail is their bent to mark the code as specific syntax, applying a Markdown-formatted code block indicating both that the output *is* code, and what language it is in.,to the first and last line of the output (Listing 9.1).

```
1 ```python
2
3 [ scenario code ]
4
5 ```
```

Listing 9.1: LLM-generated Python code with Markdown syntax. The bracketed part on line 3 has been added for demonstration purposes, removing the actual code for brevity.

Upon manually removing these syntactic artefacts, we can go ahead with executing the scenario. But as previously mentioned, we are unable get any meaningful results. This comes down to (1) halluciantion of Python code, and (2) Carla problems . Writing code to programatically these lines is naturally trivial, but we have not gone ahead with implementing this due to having the focus being on resolving the other issues that prevented the scenarios from being executed properly.

Something worth noting is that the LLM demonstrates a promising ability to explain back to the user *how* it ehnahced the scenario, e.g. in the fom of bullets in a docstring of the output code (see listing 9.2).

```
#!/usr/bin/env python

# Copyright (c) 2019-2020 Intel Corporation
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

"""
Cut in scenario:

The scenario realizes a driving behavior on the highway.
The user-controlled ego vehicle is driving straight and keeping its
    velocity at a constant level.
Another car is cutting just in front, coming from left or right lane.

The ego vehicle may need to brake to avoid a collision.

Enhanced scenario:
- Increased background traffic with varying speeds to create a more
    crowded environment.
- Challenging weather conditions (heavy rain, fog, strong winds) to
    reduce visibility and grip.
- Nighttime setting to further decrease visibility.
- Randomization of speeds and trigger distances for increased
    unpredictability.
"""
[...]
```

Listing 9.2: Head of an LLM-enhanced scenario, highlighting how the LLM can add an explenation of how it enhanced the scenario.

9.1.1 Hallucinations in the enhanced scenarios

The LLM typically seems to be on the right track, outlining something that *sounds* like a good approach to satisfying our prompt of decreasing the driveability of the scenario. But in practice, it will often hallucinate methods that don't exist, or use terms and phrasing that are not valid keywords in the Carla specificication. This is in line with what was found by e.g. Aiersilan [1, p. 14542] (See AutoSceneGen → p. 17 in Related work).

Non-existing methods

As mentioned, the LLM seems to have the right idea of what it can do to achieve the stated goal. But the way that it goes about obtaining it, does not always work. The enhanced scenario code will often call methods that don't exist. This leads to a runtime exception in the scenario runner when executing the enhanced scenario.

Non-existing arguments

In a similar vein to the non-existing methods, non-exisiting *arguments* were also shown to appear. The LLM could simply call methods that were already being used, with additional arguments that made semeantic sense, but that were not a part of the function definition. This also causes runtime exceptions in the scenario runner.

Illegal property keywords

Another trend we observed was the usage of various keywords that simply don't exist in the Carla repetoire. Where Carla would recognize the word 'snowstorm', the Large Language Model (LLM) proposed using the word 'blizzard'.

9.1.2 Carla crashes with certain scenarios

There appears to be a bug in Carla version 0.9.15[1] which causes the program to *hard crash* when executing certain scenarios with metric recording enabled. This has been reported to the project Github[2], but as of 2025-09-30 it has not been resolved. Testing shows that the same scenarios may be ran without crashing when **not recording**, but this naturally has severe implications for our opportunities of obtaining data from the simulation run. The 'record' function of the scenario runner is the crux of measuring the driveability of the scenario.

9.2 Metrics used for evaluation

We measure several metrics for evaluating the driveability of the scenario. The principal is *jerk*.

Due to the above resons with getting the enhanced scenarios to run, there is however minimal data to bases any qualitative analysis on.

[1] Which is the version employed for this project.

[2] By several members of the scientific community, see e.g.

- https://github.com/carla-simulator/carla/issues/9170 and
- https://github.com/carla-simulator/carla/issues/9152

Chapter 10

Discussion

10.1 Environmental concerns

Cost/benefit with using LLMs. Refer back to General challenges with LLMs → p. 10.

While we demonstrated promising results in Chapter 9 → p. 36, it is important to keep in mind the environmental cost of using the LLMs for this purpose. How good should the results need to be in order to justify using LLMs?

Perhaps future work can look into obtaining similar results using greener strategies.

10.2 Realism in the enhanced scenario

It is very easy to get bad driveability if your scene is bonkers. But there is no real world value/practical applicability in these scenarios?

https://www.simula.no/research/reality-bites-assessing-realism-driving-scenarios-large-language-models

Virker som at [6] har gjort et arbeid med å definere metrics for dette.

10.3 LLM context size

Hvis man har lange scenarios kan de overgå LLMens kontekst size og så mister man ting?

10.4 Python / OpenScenario / DSI

Con med Python: LLMen kan bruke utdatert syntax / bruke ting som ikke stemmer overens med den versjonen du vil bruke. De andre er mer "konstante" og mindre sårbare for dete

Chapter 11

Further work

11.1 LLM aspects

11.1.1 Different promtping strategies

Overdrivelser? Typ "Det er veldig viktig for meg at du gjør dette fordi da blir jeg glad"? Vise til litteratur som underbygger sånt.

11.1.2 Temperature

Hallucination.

11.1.3 Pretraining?

11.1.4 Retrieval-augmented generation (RAG)

Context, affordances.

11.1.5 More models

More models more good?

11.1.6 Tool calling

Can give the LLM access to tools, e.g. methods for adding objects etc.

11.2 GUI visualisations

Maybe: Frontend client - web GUI - Ivar If Loki does its job effectively, we can create a web based frontend for doing the process. It could do the same as Loki, but with greater ease of use. Having a GUI allows for making neat visualisations. Motivate why our enhanced test cases are better by showing it.

11.3 Instant validation of test case syntax

Compiler-stuff. Syntax. Parsing.

11.4 Other datasets

We used dataset x for our experiments. Scenario datasets y and z can also be used

Chapter 12

Conclusion

In this master's thesis, we propose a tool – HEFE– for using Large Language Models (LLMs) to decrease the driveability of Autonomous driving system (ADS) scenarios in order to expose underlying weaknesses in the ADS. We show this work is in line with other works in the field, and we show that our results are TODO.

Bibliography

[1] Aizierjiang Aiersilan. "Generating Traffic Scenarios via In-Context Learning to Learn Better Motion Planner". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 39. 14. 2025, pp. 14539–14547. DOI: 10.1609/aaai.v39i14.33593.

[2] Matthias Althoff, Markus Koschi, and Stefanie Manzinger. "CommonRoad: Composable benchmarks for motion planning on roads". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 719–726. DOI: 10.1109/IVS.2017.7995802.

[3] Philip W Anderson. "More Is Different: Broken symmetry and the nature of the hierarchical structure of science." In: *Science* 177.4047 (1972), pp. 393–396.

[4] Emily M. Bender et al. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 610–623. ISBN: 9781450383097. DOI: 10.1145/3442188.3445922. URL: https://doi.org/10.1145/3442188.3445922.

[5] Roger Brown. "Reference in memorial tribute to Eric Lenneberg". In: *Cognition* 4.2 (1976), pp. 125–153. ISSN: 0010-0277. DOI: https://doi.org/10.1016/0010-0277(76)90001-9. URL: https://www.sciencedirect.com/science/article/pii/0010027776900019.

[6] Cheng Chang et al. "LLMScenario: Large Language Model Driven Scenario Generation". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 54.11 (2024), pp. 6581–6594. DOI: 10.1109/TSMC.2024.3392930.

[7] He Chen et al. "Generating Autonomous Driving Test Scenarios based on OpenSCENARIO". In: *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. 2022, pp. 650–658. DOI: 10.1109/DSA56465.2022.00093.

[8] Can Cui et al. "Drive As You Speak: Enabling Human-Like Interaction With Large Language Models in Autonomous Vehicles". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops*. Jan. 2024, pp. 902–909.

[9] Can Cui et al. *Large Language Models for Autonomous Driving (LLM4AD): Concept, Benchmark, Experiments, and Challenges*. 2025. arXiv: 2410.15281 [cs.RO]. URL: https://arxiv.org/abs/2410.15281.

[10] Jean-Emmanuel Deschaud. *KITTI-CARLA: a KITTI-like dataset generated by CARLA Simulator*. 2021. arXiv: 2109.00892 [cs.CV]. URL: https://arxiv.org/abs/2109.00892.

[11] Yi Ding and Tianyao Shi. "Sustainable LLM Serving: Environmental Implications, Challenges, and Opportunities : Invited Paper". In: *2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)*. 2024, pp. 37–38. DOI: 10.1109/IGSC64514.2024.00016.

[12] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[13] Epic Games. *Unreal Engine*. Version 4.22.1. Apr. 25, 2019. URL: https://www.unrealengine.com.

[14] Gilles Fauconnier and Mark Turner. "Conceptual Blending, Form and Meaning". In: *Recherches en Communication; No 19: Sémiotique cognitive — Cognitive Semiotics; 57-86* 19 (Mar. 2003). DOI: 10.14428/rec.v19i19.48413.

[15] Erwin de Gelder, Maren Buermann, and Olaf Op Den Camp. "Coverage Metrics for a Scenario Database for the Scenario-Based Assessment of Automated Driving Systems". In: *2024 IEEE International Automated Vehicle Validation Conference (IAVVC)*. IEEE. 2024, pp. 1–8.

[16] Junyao Guo, Unmesh Kurup, and Mohak Shah. "Is it safe to drive? An overview of factors, metrics, and datasets for driveability assessment in autonomous driving". In: *IEEE Transactions on Intelligent Transportation Systems* 21.8 (2019), pp. 3135–3151.

[17] WuLing Huang et al. "Autonomous vehicles testing methods review". In: *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2016, pp. 163–168.

[18] Nidhal Jegham et al. *How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference*. 2025. arXiv: 2505.09598 [cs.CY]. URL: https://arxiv.org/abs/2505.09598.

[19] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released January 12, 2025. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/.

[20] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[21] Krzysztof Lebioda et al. *Are requirements really all you need? A case study of LLM-driven configuration code generation for automotive simulations*. 2025. arXiv: 2505.13263 [cs.SE]. URL: https://arxiv.org/abs/2505.13263.

[22] Baolin Li et al. "Sprout: Green Generative AI with Carbon-Efficient LLM Inference". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Ed. by Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 21799–21813. DOI: 10.18653/v1/2024.emnlp-main.1215. URL: https://aclanthology.org/2024.emnlp-main.1215/.

[23] Pengfei Li et al. *Making AI Less "Thirsty": Uncovering and Addressing the Secret Water Footprint of AI Models*. 2025. arXiv: 2304.03271 [cs.LG]. URL: https://arxiv.org/abs/2304.03271.

[24] Xuan Li et al. “ChatGPT-Based Scenario Engineer: A New Framework on Scenario Generation for Trajectory Prediction”. In: *IEEE Transactions on Intelligent Vehicles* 9.3 (2024), pp. 4422–4431. DOI: 10.1109/TIV.2024.3363232.

[25] Yuanfei Lin, Michael Ratzel, and Matthias Althoff. “Automatic Traffic Scenario Conversion from OpenSCENARIO to CommonRoad”. In: *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*. 2023, pp. 4941–4946. DOI: 10.1109/ITSC57777.2023.10422422.

[26] Pengfei Liu et al. “Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing”. In: *ACM Comput. Surv.* 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3560815. URL: https://doi.org/10.1145/3560815.

[27] Chengjie Lu, Tao Yue, and Shaukat Ali. “DeepScenario: An Open Driving Scenario Dataset for Autonomous Driving System Testing”. In: *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)* (2023), pp. 52–56.

[28] Chengjie Lu et al. “Learning Configurations of Operating Environment of Autonomous Vehicles to Maximize their Collisions”. In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 384–402. DOI: 10.1109/TSE.2022.3150788.

[29] Y.K. Malaiya et al. “The relationship between test coverage and reliability”. In: *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. 1994, pp. 186–195. DOI: 10.1109/ISSRE.1994.341373.

[30] Youngseok Park, Ji Hyun Yang, and Sejoon Lim. “Development of Complexity Index and Predictions of Accident Risks for Mixed Autonomous Driving Levels”. In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2018, pp. 1181–1188. DOI: 10.1109/SMC.2018.00208.

[31] Nenad Petrovic et al. “LLM-Driven Testing for Autonomous Driving Scenarios”. In: *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*. 2024, pp. 173–178. DOI: 10.1109/FLLM63129.2024.10852505.

[32] Guodong Rong et al. “LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving”. In: *arXiv preprint arXiv:2005.03778* (2020).

[33] Max Theo Schmidt, Ulrich Hofmann, and M. Essayed Bouzouraa. “A novel goal oriented concept for situation representation for ADAS and automated driving”. In: *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. 2014, pp. 886–893. DOI: 10.1109/ITSC.2014.6957801.

[34] Shital Shah et al. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[35] Shuhan Tan et al. *Language Conditioned Traffic Generation*. 2023. arXiv: 2307.07947 [cs.CV]. URL: https://arxiv.org/abs/2307.07947.

[36] Bill Tomlinson et al. “The carbon emissions of writing and illustrating are lower for AI than for humans”. In: *Scientific Reports* 14.1 (2024), p. 3732.

[37] Simon Ulbrich et al. “Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving”. In: *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. 2015, pp. 982–988. DOI: 10.1109/ITSC.2015.164.

[38] Ashish Vaswani et al. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[39] Jian Wang et al. "A Survey of Vehicle to Everything (V2X) Testing". In: *Sensors* 19.2 (2019). ISSN: 1424-8220. DOI: 10.3390/s19020334. URL: https://www.mdpi.com/1424-8220/19/2/334.

[40] Jason Wei et al. *Emergent Abilities of Large Language Models*. 2022. arXiv: 2206.07682 [cs.CL]. URL: https://arxiv.org/abs/2206.07682.

[41] Klaus Peter Wershofen and Volker Graefe. "Situationserkennung als Grundlage der Verhaltenssteuerung eines mobilen Roboters". In: *Autonome Mobile Systeme 1996*. Ed. by Günther Schmidt and Franz Freyberger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 170–179. ISBN: 978-3-642-80324-6.

[42] Tao Yue, Shaukat Ali, and Man Zhang. "RTCM: a natural language based, automated, and practical test case generation framework". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 397–408. ISBN: 9781450336208. DOI: 10.1145/2771783.2771799. URL: https://doi.org/10.1145/2771783.2771799.

[43] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2025. arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223.

[44] Yongqi Zhao et al. *A Survey on the Application of Large Language Models in Scenario-Based Testing of Automated Driving Systems*. 2025. arXiv: 2505.16587 [cs.SE]. URL: https://arxiv.org/abs/2505.16587.

[45] Yongqi Zhao et al. "Chat2Scenario: Scenario Extraction From Dataset Through Utilization of Large Language Model". In: *2024 IEEE Intelligent Vehicles Symposium (IV)*. 2024, pp. 559–566. DOI: 10.1109/IV55156.2024.10588843.

Appendix

Appendix A

Scenario file diffs

The diffs represent the *diff*erence between two files, highlighting what has changed. In this context – the red indicates something that was changed from the original scenario, and the green indicates something that was added by the LLM. The lines in black are unchanged.

A.1 Cut_in-enhanced-5.py

```
--- cut_in.py 2025-10-01 12:02:38
+++ cut_in-enhanced-5.py  2025-10-01 12:02:50
@@ -1,3 +1,4 @@
+```python
 #!/usr/bin/env python

 # Copyright (c) 2019-2020 Intel Corporation
@@ -23,17 +24,20 @@
 from srunner.scenariomanager.scenarioatomics.atomic_behaviors
   import (ActorTransformSetter,
         LaneChange,
         WaypointFollower,
-        AccelerateToCatchUp)
+        AccelerateToCatchUp,
+        KeepVelocity) # Added KeepVelocity for background traffic
 from srunner.scenariomanager.scenarioatomics.atomic_criteria
   import CollisionTest
 from srunner.scenariomanager.scenarioatomics.
   atomic_trigger_conditions import InTriggerDistanceToVehicle,
   DriveDistance
 from srunner.scenarios.basic_scenario import BasicScenario
+from srunner.tools.scenario_helper import
   get_waypoint_in_distance # Added for spawning background
   traffic


 class CutIn(BasicScenario):
```

```
22
23      """
24      The ego vehicle is driving on a highway and another car is cutting in just in front.
25 -    This is a single ego vehicle scenario
26 +    This is a single ego vehicle scenario, now enhanced with more complexity due to
27 +    adverse weather, challenging time of day, and additional background traffic.
28      """
29
30      timeout = 1200
31 @@ -45,15 +49,28 @@
32          self._map = CarlaDataProvider.get_map()
33          self._reference_waypoint = self._map.get_waypoint(config.trigger_points[0].location)
34
35 -        self._velocity = 40
36 -        self._delta_velocity = 10
37 -        self._trigger_distance = 30
38 +        self._velocity = 40  # Default velocity for cut-in vehicle
39 +        self._delta_velocity = 10 # Default delta velocity for cut-in acceleration
40 +        self._trigger_distance = 30 # Default distance for cut-in trigger
41
42 +        # --- ENHANCEMENT: More detailed randomization for key parameters ---
43 +        if randomize:
44 +            self._velocity = random.uniform(25, 55) # Slightly wider range for cut-in vehicle speed
45 +            self._trigger_distance = random.uniform(15, 45) # Slightly wider range for cut-in trigger distance
46 +            self._delta_velocity = random.uniform(8, 18) # Randomize acceleration aggressiveness of cut-in car
47 +            self._background_traffic_speed_variation = random.uniform(0.8, 1.2) # For background traffic speed
48 +        else:
49 +            self._background_traffic_speed_variation = 1.0
50 +
51 +
52          # get direction from config name
53          self._config = config
54          self._direction = None
55 -        self._transform_visible = None
56 +        self._transform_visible = None # This is for initial spawn location (above map)
57
58 +        self.number_of_background_vehicles = 3 # --- ENHANCEMENT: Number of additional background vehicles ---
59 +        self.background_vehicles = [] # List to store background vehicles
60 +
61          super(CutIn, self).__init__("CutIn",
62                                      ego_vehicles,
```

Appendix A. Scenario file diffs

```
                                        config,
@@ -61,10 +78,48 @@
                                        debug_mode,
                                        criteria_enable=
    criteria_enable)

-       if randomize:
-           self._velocity = random.randint(20, 60)
-           self._trigger_distance = random.randint(10, 40)
+       # --- ENHANCEMENT: Add adverse weather and time of day
    ---
+       self._setup_environment()

+
+    def _setup_environment(self):
+        """
+        Sets up the environment with adverse weather and
    challenging time of day.
+        This significantly decreases driveability by reducing
    visibility and grip.
+        """
+        # Randomize weather parameters for more variations and
    challenge
+        weather_params = carla.WeatherParameters(
+            cloudiness=random.uniform(70, 100),
+            precipitation=random.uniform(50, 90), # Rain
+            precipitation_deposits=random.uniform(50, 90), #
    Puddles
+            wind_intensity=random.uniform(0.5, 1.5), # Stronger
    wind
+            fog_density=random.uniform(20, 50), # Moderate fog
+            wetness=random.uniform(50, 90), # Wet roads
+            sun_altitude_angle=random.uniform(-10, 10) # Dusk/
    Dawn or low sun
+        )
+        self.world.set_weather(weather_params)
+
+        # Force challenging time of day (night or very low sun
    angle)
+        if random.random() < 0.6: # 60% chance of night
+            self.world.set_weather(carla.WeatherParameters(
+                sun_altitude_angle=-90.0,
+                cloudiness=random.uniform(80, 100),
+                precipitation=random.uniform(60, 100),
+                precipitation_deposits=random.uniform(60, 100),
+                wetness=random.uniform(70, 100),
+                fog_density=random.uniform(30, 60),
+                moon_intensity=random.uniform(0.1, 0.5) # Dim
    moonlight
+            ))
+        else: # Dusk/dawn with challenging conditions
+            self.world.set_weather(carla.WeatherParameters(
+                sun_altitude_angle=random.uniform(-20, 20),
+                cloudiness=random.uniform(70, 100),
+                precipitation=random.uniform(30, 70),
+                precipitation_deposits=random.uniform(30, 70),
```

```
+                wetness=random.uniform(30, 70),
+                fog_density=random.uniform(10, 40)
+            ))
+
     def _initialize_actors(self, config):

         # direction of lane, on which other_actor is driving before lane change
@@ -74,69 +129,154 @@
         if 'RIGHT' in self._config.name.upper():
             self._direction = 'right'

-        # add actors from xml file
+        # add actors from xml file (this is the cutting-in vehicle)
         for actor in config.other_actors:
             vehicle = CarlaDataProvider.request_new_actor(actor.model, actor.transform)
             self.other_actors.append(vehicle)
+            # Initially disable physics. It will be enabled by ActorTransformSetter when it drops.
             vehicle.set_simulate_physics(enabled=False)

-        # transform visible
+        # transform visible: This places the cutting-in car high above the map initially
         other_actor_transform = self.other_actors[0].get_transform()
         self._transform_visible = carla.Transform(
             carla.Location(other_actor_transform.location.x,
                            other_actor_transform.location.y,
-                           other_actor_transform.location.z + 105),
+                           other_actor_transform.location.z + 105), # Spawn high above
             other_actor_transform.rotation)

+        # --- ENHANCEMENT: Spawn background traffic to increase complexity ---
+        self._spawn_background_traffic()
+
+    def _spawn_background_traffic(self):
+        """
+        Spawns additional background vehicles to increase traffic density and complexity.
+        These vehicles will follow simple driving behaviors.
+        """
+        ego_waypoint = self._map.get_waypoint(self.ego_vehicles[0].get_location())
+
+        available_vehicle_blueprints = CarlaDataProvider.get_filtered_traffic_actor_blueprints(
+            'vehicle.*',
+            rolename='background'
+        )
+        if not available_vehicle_blueprints:
```

Appendix A. Scenario file diffs

```
153 +            return # No blueprints available, skip spawning background traffic
154 +
155 +        spawn_points = []
156 +
157 +        # Background vehicle 1: On ego's lane, behind ego
158 +        wp_behind_ego = get_waypoint_in_distance(ego_waypoint, -random.uniform(25, 45), False)
159 +        if wp_behind_ego and wp_behind_ego.lane_id == ego_waypoint.lane_id:
160 +            spawn_points.append(wp_behind_ego.transform)
161 +
162 +        # Background vehicle 2: On ego's lane, ahead of ego (further away, possibly slower)
163 +        wp_ahead_ego = get_waypoint_in_distance(ego_waypoint, random.uniform(60, 100), False)
164 +        if wp_ahead_ego and wp_ahead_ego.lane_id == ego_waypoint.lane_id:
165 +            spawn_points.append(wp_ahead_ego.transform)
166 +
167 +        # Background vehicle 3: On an adjacent lane, adding general highway traffic
168 +        # This car will be on the lane opposite to where the cut-in is coming from,
169 +        # making the driving environment more dense and limiting escape routes.
170 +        if self._direction == 'left': # Cut-in from left, add traffic on right lane
171 +            adjacent_lane_wp = ego_waypoint.get_right_lane()
172 +        else: # Cut-in from right, add traffic on left lane
173 +            adjacent_lane_wp = ego_waypoint.get_left_lane()
174 +
175 +        if adjacent_lane_wp:
176 +            # Place it slightly behind or abreast of ego
177 +            wp_adj_lane = get_waypoint_in_distance(adjacent_lane_wp, random.uniform(-10, 20), False)
178 +            if wp_adj_lane:
179 +                spawn_points.append(wp_adj_lane.transform)
180 +
181 +        # Spawn the vehicles
182 +        for i in range(min(self.number_of_background_vehicles, len(spawn_points))):
183 +            bp = random.choice(available_vehicle_blueprints)
184 +            transform = spawn_points[i]
185 +            # Adjust Z to prevent spawning issues if ground isn't perfectly flat
186 +            transform.location.z += 0.5
187 +
188 +            vehicle = CarlaDataProvider.request_new_actor(bp.id, transform)
189 +            if vehicle:
190 +                self.background_vehicles.append(vehicle)
191 +                self.other_actors.append(vehicle) # Add to general other_actors for cleanup and criteria checks
192 +                vehicle.set_simulate_physics(enabled=True) # Background vehicles should have physics
```

```
+
+
     def _create_behavior(self):
         """
         Order of sequence:
-        - car_visible: spawn car at a visible transform
+        - car_visible: spawn cut-in car at a visible transform (above map)
         - just_drive: drive until in trigger distance to ego_vehicle
         - accelerate: accelerate to catch up distance to ego_vehicle
         - lane_change: change the lane
         - endcondition: drive for a defined distance
+
+        --- ENHANCEMENT: Integrate background traffic behaviors ---
         """

-        # car_visible
-        behaviour = py_trees.composites.Sequence("CarOn_{}_Lane".format(self._direction))
-        car_visible = ActorTransformSetter(self.other_actors[0], self._transform_visible)
-        behaviour.add_child(car_visible)
+        # --- Cut-in vehicle behavior (Original logic, now wrapped in a sequence) ---
+        cut_in_behavior_sequence = py_trees.composites.Sequence("CutInBehavior")

-        # just_drive
-        just_drive = py_trees.composites.Parallel(
-            "DrivingStraight", policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
+        # car_visible: Teleport cut-in car high above, then let it drop (physics enabled by setter)
+        car_visible = ActorTransformSetter(self.other_actors[0], self._transform_visible, physics_enabled=True)
+        cut_in_behavior_sequence.add_child(car_visible)

-        car_driving = WaypointFollower(self.other_actors[0], self._velocity)
-        just_drive.add_child(car_driving)
+        # just_drive: Wait until cut-in car is close enough to ego
+        just_drive_parallel = py_trees.composites.Parallel(
+            "DrivingStraightUntilTrigger", policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)

-        trigger_distance = InTriggerDistanceToVehicle(
+        # The actual driving behavior for the cutting-in car
+        cut_in_car_driving = WaypointFollower(self.other_actors[0], self._velocity)
+        just_drive_parallel.add_child(cut_in_car_driving)
+
```

Appendix A. Scenario file diffs

```
+        # Trigger condition: When the cut-in car is within trigger distance to ego
+        trigger_distance_condition = InTriggerDistanceToVehicle(
             self.other_actors[0], self.ego_vehicles[0], self._trigger_distance)
-        just_drive.add_child(trigger_distance)
-        behaviour.add_child(just_drive)
+        just_drive_parallel.add_child(trigger_distance_condition)
+        cut_in_behavior_sequence.add_child(just_drive_parallel)

-        # accelerate
+        # accelerate: Accelerate to match/catch up with ego
         accelerate = AccelerateToCatchUp(self.other_actors[0], self.ego_vehicles[0], throttle_value=1,
                                          delta_velocity=self._delta_velocity, trigger_distance=5, max_distance=500)
-        behaviour.add_child(accelerate)
+        cut_in_behavior_sequence.add_child(accelerate)

-        # lane_change
+        # lane_change: Perform the actual cut-in
         if self._direction == 'left':
             lane_change = LaneChange(
                 self.other_actors[0], speed=None, direction='right', distance_same_lane=5, distance_other_lane=300)
-            behaviour.add_child(lane_change)
-        else:
+            cut_in_behavior_sequence.add_child(lane_change)
+        else: # self._direction == 'right'
             lane_change = LaneChange(
                 self.other_actors[0], speed=None, direction='left', distance_same_lane=5, distance_other_lane=300)
-            behaviour.add_child(lane_change)
+            cut_in_behavior_sequence.add_child(lane_change)

-        # endcondition
+        # endcondition: Drive for a defined distance after cut-in
         endcondition = DriveDistance(self.other_actors[0], 200)
+        cut_in_behavior_sequence.add_child(endcondition)

-        # build tree
-        root = py_trees.composites.Sequence("Behavior", policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
-        root.add_child(behaviour)
-        root.add_child(endcondition)
+
+        # --- Background traffic behavior (Parallel to the main cut-in logic) ---
+        # All background vehicles will just keep driving straight.
+        background_traffic_behavior = py_trees.composites.Parallel(
+            "BackgroundTraffic", policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ALL
+        )
```

```
277 +        # Ensure we only add behavior for actually spawned background vehicles
278 +        if self.background_vehicles:
279 +            initial_ego_speed_kph = CarlaDataProvider.get_velocity(self.ego_vehicles[0]).length() * 3.6 # Get ego's initial speed in km/h
280 +            for i, b_vehicle in enumerate(self.background_vehicles):
281 +                # Calculate a slightly varied speed for each background vehicle
282 +                # Ensure a minimum reasonable speed for traffic flow
283 +                b_speed = max(initial_ego_speed_kph * self._background_traffic_speed_variation, 20)
284 +                background_traffic_behavior.add_child(
285 +                    KeepVelocity(b_vehicle, target_velocity=b_speed)
286 +                )
287 +
288 +        # --- Overall behavior tree: Cut-in behavior and background traffic run in parallel ---
289 +        # The scenario completes when the primary cut-in behavior finishes.
290 +        root = py_trees.composites.Parallel("OverallScenario", policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
291 +        root.add_child(cut_in_behavior_sequence)
292 +        if self.background_vehicles: # Only add if there are background vehicles
293 +            root.add_child(background_traffic_behavior)
294 +
295          return root
296
297      def _create_test_criteria(self):
298  @@ -145,8 +285,8 @@
299          """
300          criteria = []
301
302 +        # Collision criteria for the ego vehicle with *any* other actor (including new background traffic)
303          collision_criterion = CollisionTest(self.ego_vehicles[0])
304  -
305          criteria.append(collision_criterion)
306
307          return criteria
308  @@ -156,3 +296,4 @@
309          Remove all actors after deletion.
310          """
311          self.remove_all_actors()
312 +```
313  \ No newline at end of file
```

Listing A.1: The diff of an LLM-enhanced Cut_in scenario, highlighting *how* the LLM enhanced the scenario.