

University of Stuttgart
Institute of Industrial Automation and
Software Engineering

Development of an AI-Based Test Case Generation Approach for the Validation of Variant-Rich Software- Defined Systems

Master Thesis 3646

Submitted at the University of Stuttgart by
John Pravin Arockiasamy

Information Technology (INFOTECH)

Examiner:	Prof. Dr.-Ing. Christof Ebert Prof. Dr.-Ing. Dr. h.c. Michael Weyrich
Supervisor:	Lennard Hettich, M.Sc.

18-06-2024



Table of Contents

TABLE OF CONTENTS.....	II
TABLE OF FIGURES	IV
TABLE OF TABLES	V
TABLE OF ABBREVIATIONS.....	VI
GLOSSARY	VII
ABSTRACT	IX
ABSTRACT	X
1 MOTIVATION AND INTRODUCTION	11
2 FUNDAMENTALS	13
2.1 Transformer Models	13
2.1.1 Training	14
2.1.2 Types and Impact of Transformer Models	14
2.1.3 Ways to Improve the performance of the Transformer Models	15
2.2 CARLA Simulation Environment	16
2.3 Automotive Safety-Critical Scenarios	17
2.4 ASAM OpenSCENARIO and OpenDRIVE	19
3 TECHNOLOGY ANALYSIS.....	21
3.1 Intended Test Infrastructure	21
3.1.1 Requirements Engine	22
3.1.2 Assistant System	24
3.1.3 Scenario Database/Generation	26
3.1.4 Scenario Engine	27
3.1.5 System Under Test (SUT) Evaluation	29
3.1.6 Key Performance Indicators (KPIs)	29
3.1.7 Scenario Comprehension	30
3.1.8 Research Intensity of the Components:	31
3.1.9 Survey.....	31
3.1.10 Decision on work of this thesis	35
3.2 Comparison on Large Language Models	39
3.2.1 Key Considerations for Choosing Large Language Models	39
3.2.2 Decision on LLMs for this thesis	41
4 CONCEPTUAL WORK.....	45
4.1 Scenario Generation in Carla	45

4.1.1	Scenario Definition Languages	46
4.1.2	Simulation of Scenarios in CARLA	47
4.1.3	Test Parameter Space Generation Process	47
4.2	Scenario Adaptation and Manipulation in CARLA	49
4.2.1	Test Case Generation Process	49
5	PROTOTYPE IMPLEMENTATION	52
5.1	Implementation of Scenario Generation in CARLA	52
5.2	Implementation of AI Technologies	55
6	EVALUATION AND RESULTS	56
6.1	Test Parameter Space Generation Process	56
6.1.1	Evaluation of Test Parameter Space Generation Process	56
6.1.2	Results and Discussion of the Test Parameter Space Generation Process	58
6.1.3	Ablation Study	60
6.2	Test Case Generation Process	61
6.2.1	Evaluation of the Test Case Generation Process	62
6.2.2	Results and Discussion of the Test Case Generation Process	64
7	CONCLUSION AND OUTLOOK	71
	BIBLIOGRAPHY	73
	DECLARATION OF COMPLIANCE	80

Table of Figures

Figure 1 Robo-Test Approach for Testing Autonomous Systems	11
Figure 2 Robo-Test Approach for Testing Autonomous Systems	12
Figure 3 The Schematic of The Encoder-Decoder Structure of The Transformer Architecture.....	13
Figure 4 The RGB Camera Output (left) and LiDAR Sensor Output from CARLA [67].....	16
Figure 5 Three Levels of Abstraction for Testing Automated Driving Systems Introduced by the PEGASUS Project.....	18
Figure 6 Five Layer Model for Description of Scenarios Introduced by The PEGASUS Project	19
Figure 7 Revised Intended Test Infrastructure	21
Figure 8 Number of Publications Across Different Processes/Components taken from arXiv	34
Figure 9 Overview of the Scenario Simulation in CARLA.....	47
Figure 10 Flow Diagram of the Test Parameter Space Generation in the Intended Test Infrastructure.....	49
Figure 11 Flow Diagram of the Test Case Generation from the Test Parameters..	50
Figure 12 Rough Illustration of a Generated Scenario	53
Figure 13 Example Picture of Generated Scenario Named “Hero_And_Bicycle_On_The_Same_Lane”	54
Figure 14 One Sample from the Test Parameter Space Generation Benchmark.	547
Figure 15 The Results of Different Sampling Techniques, Along with Their Class Metrics, For Each Iteration Across All 10 Test Scenarios	67
Figure 16 Collision Between the Hero Vehicle (SUT) And the Bicycle.	68
Figure 17 Collision Not Registered under “CollisionTest” Despite Hero Vehicle (SUT) Colliding with Bicycle.....	69
Figure 18 Bar Chart Comparing the Test Scenario 01 Experiment Results of Sampling Techniques	70

Table of Tables

Table 1: Survey results on different components of intended test infrastructure from Google Scholar.	32
Table 2: Survey results on papers from 1994 from 2024 from arXiv.	33
Table 3: Comparison of the Components of the Intended test infrastructure based on the survey.	35
Table 4: Comparison of different approaches for the Test Case Generation.	37
Table 5: Comparison on different popular LLMs along with their open-source evaluation metrics	44
Table 6: The different road configurations in Town01 & Karlsruhe Maps.....	52
Table 7: The number of available scenarios in different road configurations and the actors involved.....	53
Table 8: The different LLMs and their results on the test parameter generation benchmark.	59
Table 9: The ablation study results between human-designed prompt and AI-generated prompt	60
Table 10: The ablation study results between one-shot learning and one-shot learning with CoT prompting techniques.....	61
Table 11: The Key Performance Indicators (KPIs) used for the evaluation of SUT.....	62
Table 12: Details of different manually created class metrics.....	63
Table 13: The results of sampling techniques on 10 different test scenarios	65

Table of Abbreviations

CARLA	CAR Learning to Act
AV	A utonomous V ehicle
SUT	S ystem U nder T est
ADAS	A dvanced D riving A ssistance S ystems
AI	A rtificial I ntelligence
NLP	N atural L anguage P rocessing
LLM	L arge L anguage M odel
GPT	G enerative P re-trained T ransformer
CoT	C hain- of - T hought
DL	D eep L earning
ML	M achine L earning
BO	B ayesian O ptimization
JSON	J ava S cript O bject N otation
SDL	S cenario D escription L anguage
ODD	O perational D esign D omain
KPI	K ey P erformance I ndex
API	A pplication P rogramming I nterface
CPU	C entral P rocessing U nit
GPU	G raphics P rocessing U nit
RAM	R andom- A ccess M emory
XML	E xtensible M arkup L anguage

Glossary

CARLA Simulator	An open-source simulation platform used to develop, train, and validate autonomous driving systems. It provides a range of tools for testing in realistic urban environments, including various sensors and vehicles [67].
Scenario Based Testing	A method of testing that involves creating specific scenarios to evaluate the performance and reliability of a system, such as an autonomous vehicle. This approach helps identify how the system responds to different situations and challenges.
ASAM OpenSCENARIO	A standard file format and interface specification developed by the Association for Standardization of Automation and Measuring Systems (ASAM). It defines how to describe complex driving scenarios for use in driving simulators and automated driving development [71].
ASAM OpenDRIVE	A standard file developed by ASAM which specifies a common format for describing road networks. It includes details such as road geometry, lanes, and traffic signs, which are crucial for accurate simulation and testing of autonomous driving systems [74].
JSON	A lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is often used for transmitting data between a server and a web application.
API	A set of rules and definitions that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange information.
Large Language Models (LLMs)	AI systems trained on vast datasets to understand and generate human-like text.
Prompting Technique	A method used to elicit specific responses from LLMs by crafting input prompts in particular ways to guide the output.

Sampling Methods	Techniques in statistics to select representative data subsets from larger datasets, ensuring efficient and accurate model training, evaluation, and prediction by maintaining diversity and reducing bias.
Bayesian Optimization	An optimization technique that uses Bayes' theorem to find the minimum of a function efficiently by balancing exploration and exploitation [2].
PEGASUS method	A method employs a six-layer model for systematically describing scenarios for testing automated driving systems [96].

Abstract

The dynamic digital landscape necessitates personalized solutions, highlighting the significance of software-defined systems. These systems, characterized by over-the-air updates and innovative business models, face challenges such as frequent releases and increased customer expectations, resulting in an explosion of system variations [99]. Despite the adaptability of Continuous Integration/Continuous Deployment (CI/CD) practices, ensuring the safety of numerous software-defined system variants remains a significant concern [100]. Traditional testing methods, limited by shorter development cycles and increased resource demands, fall short in this context. This thesis contributes to addressing these challenges by establishing a robust test infrastructure that leverages advanced AI techniques to generate and manipulate test scenarios, aligning with the ongoing initiatives of the IAS department.

The planned test infrastructure involves a variant-rich Advanced Driver Assistance System (ADAS) or CARLA AutoPilot controlling a simulated car in the 3D CARLA simulation environment. This thesis primarily aims to develop a fully automated system using AI to induce failures in the System Under Test (SUT) by adapting and manipulating scenarios. A secondary goal is to commission a semi-automated Safety Pool database pipeline for running scenarios within the CARLA environment. Despite challenges with the Safety Pool database, a fully automated pipeline using a Python package was created. The process of scenario generation, adaptation, and manipulation involves generating test parameter spaces and test cases using various Large Language Models (LLMs) and sampling techniques. In the test parameter space generation, OpenAI GPT-4 Turbo showed superior performance on a benchmark designed for this process compared to other closed-source and open-source models. For test case generation, the Bayesian Optimization (BO) sampling technique demonstrated higher efficiency, with promising results also observed from the OpenAI GPT-4 Turbo LLM.

This thesis significantly advances the development of an automated test infrastructure for autonomous vehicle safety validation. The innovative methods for inducing failures and validating SUT performance provide valuable insights and tools for future research, contributing to the creation of safer and more reliable autonomous vehicles.

Key Words: Software-defined systems, CI/CD, Advanced Driver Assistance System (ADAS), CARLA simulation environment, AI-based test case generation, Large Language Models (LLMs).

Abstract

Die dynamische digitale Landschaft erfordert personalisierte Lösungen, was die Bedeutung von softwaredefinierten Systemen unterstreicht. Diese Systeme, die sich durch "Over-the-Air"-Updates und innovative Geschäftsmodelle auszeichnen, stehen vor Herausforderungen wie häufigen Releases und gestiegenen Kundenerwartungen, was zu einer explosionsartigen Zunahme von Systemvarianten führt. Trotz der Anpassungsfähigkeit von Continuous Integration/Continuous Deployment (CI/CD)-Praktiken bleibt die Gewährleistung der Sicherheit zahlreicher softwaredefinierter Systemvarianten ein großes Problem. Traditionelle Testmethoden, die durch kürzere Entwicklungszyklen und erhöhten Ressourcenbedarf eingeschränkt sind, greifen in diesem Zusammenhang zu kurz. Diese Arbeit trägt zur Bewältigung dieser Herausforderungen bei, indem sie eine robuste Testinfrastruktur einrichtet, die fortschrittliche KI-Techniken zur Generierung und Manipulation von Testszenarien einsetzt und sich an den laufenden Initiativen der IAS-Abteilung orientiert.

Die geplante Testinfrastruktur umfasst ein variantenreiches Fahrerassistenzsystem (ADAS) oder CARLA AutoPilot, das ein simuliertes Fahrzeug in der 3D-Simulationsumgebung CARLA steuert. Diese Arbeit zielt in erster Linie darauf ab, ein vollautomatisches System zu entwickeln, das KI nutzt, um durch Anpassung und Manipulation von Szenarien Fehler im zu testenden System (SUT) hervorzurufen. Ein sekundäres Ziel ist die Inbetriebnahme einer halbautomatischen Safety Pool-Datenbank-Pipeline für die Ausführung von Szenarien innerhalb der CARLA-Umgebung. Trotz der Probleme mit der Safety Pool-Datenbank wurde eine vollautomatische Pipeline unter Verwendung eines Python-Pakets erstellt. Der Prozess der Szenariengenerierung, -anpassung und -manipulation umfasst die Generierung von Testparameterräumen und Testfällen unter Verwendung verschiedener Large Language Models (LLMs) und Sampling-Techniken. Bei der Generierung von Testparameterräumen zeigte OpenAI GPT-4 Turbo im Vergleich zu anderen Closed-Source- und Open-Source-Modellen eine überlegene Leistung bei einem für diesen Prozess entwickelten Benchmark. Bei der Generierung von Testfällen zeigte die Sampling-Technik der Bayes'schen Optimierung (BO) eine höhere Effizienz, wobei auch mit dem OpenAI GPT-4 Turbo LLM vielversprechende Ergebnisse erzielt wurden.

Diese Arbeit bringt die Entwicklung einer automatisierten Testinfrastruktur für die Sicherheitsvalidierung autonomer Fahrzeuge erheblich voran. Die innovativen Methoden zur Induzierung von Fehlern und zur Validierung der SUT-Leistung liefern wertvolle Erkenntnisse und Werkzeuge für die zukünftige Forschung und tragen zur Entwicklung sicherer und zuverlässiger autonomer Fahrzeuge bei.

Key Words: Software-definierte Systeme, CI/CD, Advanced Driver Assistance System (ADAS), CARLA-Simulationsumgebung, KI-basierte Testfallerstellung, Large Language Models (LLMs).

1 Motivation and Introduction

The dynamic digital landscape demands personalized solutions, emphasizing the critical role of software-defined systems. These systems operate at the software level, enabling over-the-air updates and adopting innovative business models [99]. However, the rapid evolution of technology poses challenges, such as frequent releases and increased customer expectations, resulting in an explosion of system variations. Despite the adaptability of Continuous Integration/Continuous Development (CI/CD) practices, ensuring the safety of countless software-defined system variants remains a significant concern. Conventional testing methods for variant-rich software systems, relying on individual expertise or inadequate techniques in the context of CI/CD, face limitations due to shorter development cycles and increased resource demands associated with CI/CD practices [100].

The IAS focuses on providing product developers with advanced automated techniques to select variants for efficient testing, necessitating a dedicated test infrastructure. This master's thesis contributes to the establishment of such an infrastructure, aligning with IAS's ongoing initiatives. The test infrastructure involves a System Under Test (SUT) controlling a simulated car in a 3D simulation environment (CARLA) through a specified interface.

The IAS provides a Robo-Test platform [98], crucial for identifying and covering critical corner cases during testing of autonomous systems, particularly the autonomous excavator (refer to Figure 1). This platform serves as an inspiring foundation for implementing the test infrastructure outlined in this master's thesis.

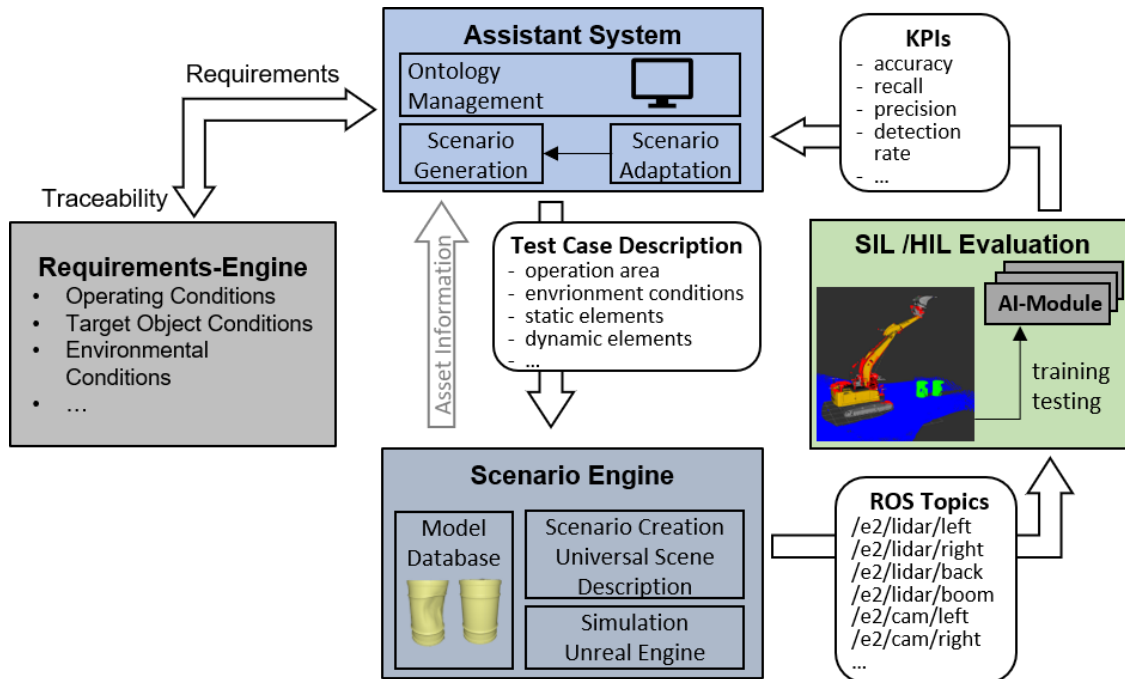


Figure 1 Robo-Test Approach for Testing Autonomous Systems

The development of the test infrastructure and its goals are briefly outlined in Figure 2, involving three projects. First, the Safety Pool scenario database compiles scenarios from expert knowledge, accident databases, and naturalistic data. A research project (RP3488) implemented a semi-automatic pipeline for executing selected scenarios from the Safety Pool on the CARLA simulation. Second, a master's thesis (MT3644) concurrently addresses the implementation of a variant-rich SUT. Simultaneously, this project focuses on AI-based generation of test cases and scenarios, aiming to challenge the safety of the SUT in the CARLA simulation environment by manipulating existing scenarios from the database.

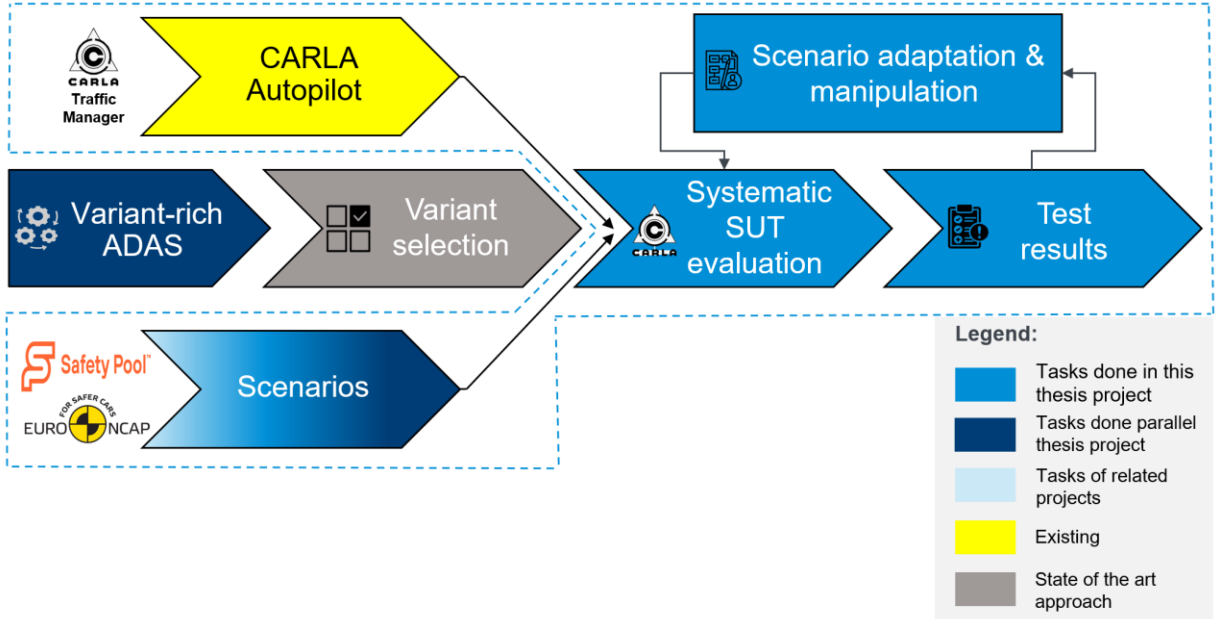


Figure 2 Robo-Test Approach for Testing Autonomous Systems

As mentioned, the primary goal of this thesis is to develop a fully automated system using advanced AI technology to induce failures in the system-under-test (SUT) controlled by the variant-rich ADAS from MT3644 or CARLA AutoPilot by adapting and manipulating scenarios. The secondary goal is to commission a semi-automated pipeline to run scenarios from the Safety Pool dataset within the CARLA simulation environment. If this approach proves unsuccessful, scenarios will be generated as part of this thesis.

The structure of this thesis is as follows: Section 2 provides a brief explanation of the fundamental concepts necessary for understanding this work. Section 3 offers a technological analysis, detailing the application of AI across all components of the intended test infrastructure. Following that, Section 4 discusses the conceptual work of this thesis. Section 5 provides a brief implementation detail and Section 6 present a detailed evaluation and the results of various approaches used to achieve these goals. Finally, Section 8 presents the conclusions of this thesis, offering insights for future work.

2 Fundamentals

In this chapter, an overview of different concepts and state-of-the-art technologies needed to understand the thesis better will be examined.

2.1 Transformer Models

The transformer architecture (Figure 3), introduced in 2017 in a paper called “Attention is All You Need”, has emerged as a groundbreaking innovation in the field of natural language processing (NLP) [97]. The transformer model has since become a dominant architecture in various NLP tasks such as language translation, text generation, and question answering.

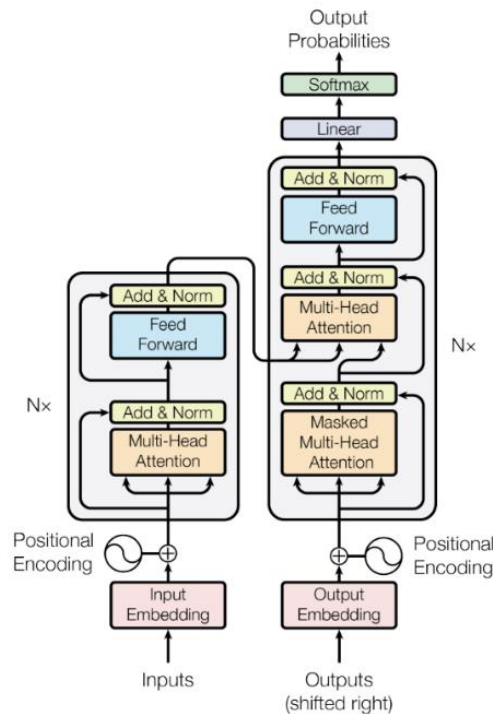


Figure 3 The Schematic of The Encoder-Decoder Structure of The Transformer Architecture - Sourced from [97]

Two key components that enable Transformer models to effectively process sequential data, such as text, are position encoding and the self-attention mechanism.

Position Encoding: In Transformer models, position encoding is a technique used to provide information about the position of each token (e.g., words or sub word pieces) within the input sequence. Instead of simply considering the order in which the words appear, a unique numerical value is assigned to each token. This position information allows the model to take into account the sequential nature of the input, enabling it to better understand and process the data [97].

Self-attention mechanism: The self-attention mechanism is another crucial component of Transformer models. This mechanism enables each token in the sequence to attend to and weigh the importance of other tokens within the same sequence. By capturing the dependencies and relationships between the tokens, the self-attention mechanism allows the model to develop a deeper understanding of the context and meaning of the input data [97].

2.1.1 Training

Transformer models are usually trained in two main phases:

2.1.1.1 Pre-training Phase

Transformer models are first trained on large amounts of unlabelled, general-domain data in a self-supervised manner. This pre-training allows the model to learn the underlying structure and patterns of the language or data it is trained on without being constrained to a specific task. Common pre-training approaches include masked language modelling (MLM), where the model predicts masked tokens in the input, and autoregressive modelling, where the model predicts the next token in a sequence. Pre-training is a computationally intensive and expensive process, often requiring millions of dollars of resources to train the largest models.

2.1.1.2 Fine-tuning Phase

After pre-training, the transformer model is then fine-tuned on a specific downstream task using labelled data. Fine-tuning involves further training the model on the target task, allowing it to specialize and perform well on that particular application. Fine-tuning typically requires less data and computational resources compared to the initial pre-training phase. Examples of downstream tasks include text classification, machine translation, question answering, and summarization.

2.1.2 Types and Impact of Transformer Models

Transformer-based models can be broadly categorized into three main types:

2.1.2.1 Auto-regressive Transformer Models or Decoder Only Models

These models, such as GPT (Generative Pretrained Transformer), are trained to predict the next token in a sequence, given the previous tokens. They are commonly used for tasks like text generation and language modelling [25].

2.1.2.2 Auto-encoding Transformer Models or Encoder Only Models

Models like BERT (Bidirectional Encoder Representations from Transformers) are trained to understand the context and meaning of text by predicting masked tokens in a sequence. They excel at tasks such as text classification, question answering, and natural language inference [25].

2.1.2.3 Sequence-to-Sequence Transformer Models or Encoder and Decoder Models

BART (Bidirectional and Auto-Regressive Transformers) and T5 (Text-to-Text Transfer Transformer) are examples of these models, which are trained to generate an output sequence given an input sequence. They are well-suited for tasks like machine translation, text summarization, and data-to-text generation [25].

The Transformer model's innovative attention-based architecture, combined with its impressive performance and versatility, has established it as a transformative breakthrough in the field of Natural Language Processing (NLP). The transformer model's scalability and modular design have been instrumental in the development of Large Language Models, such as GPT-3 [81] and Llama [84, 85]. These models have revolutionized the field of NLP by exhibiting remarkable capabilities in tasks like question answering, text generation, and even code generation. The potential of these large-scale language models to transform various industries is immense, with applications ranging from customer service and content creation to scientific research and software development.

2.1.3 Ways to Improve the performance of the Transformer Models

There are three techniques to improve the performance of transformer models specifically the Large Language Models (LLMs).

2.1.3.1 Prompt Engineering

This is a technique used to craft specifically designed prompts to guide a language model's outputs. By providing specific instructions and context within the prompt, developers can elicit more accurate, relevant, and coherent responses from the model. Techniques like prompt templates, zero-shot learning, one-shot learning, few-shot learning, tree-of-thought and chain-of-thought prompting can steer the LLMs towards more accurate and relevant outputs.

2.1.3.2 Retrieval-Augmented Generation (RAG)

This is a technique that combines language modelling with information retrieval to enhance the model's capabilities. It allows the model to access external knowledge sources and incorporate relevant information into the generated outputs. The retrieved information augments the LLM's knowledge, allowing it to generate more informed and factual responses, especially for open-ended queries.

2.1.3.3 Fine-tuning

This is a method where a pre-trained model is further trained on task-specific data to adapt it for particular applications. Techniques like LoRA (Low-Rank Adaptation) and PEFT enable efficient fine-tuning while requiring fewer computational resources compared to full model fine-tuning.

All these three methods can be used individually or in combination to enhance the performance of LLMs on various tasks. Prompt engineering is considered the easiest method, while RAG and PEFT are more advanced techniques that can further boost the LLM's capabilities.

2.2 CARLA Simulation Environment

CARLA (Car Learning to Act) is an open-source simulator designed for autonomous driving research. It provides a highly realistic 3D environment emulating real-world towns, cities, and highways, along with vehicles, pedestrians, and other objects that populate these driving spaces [67].

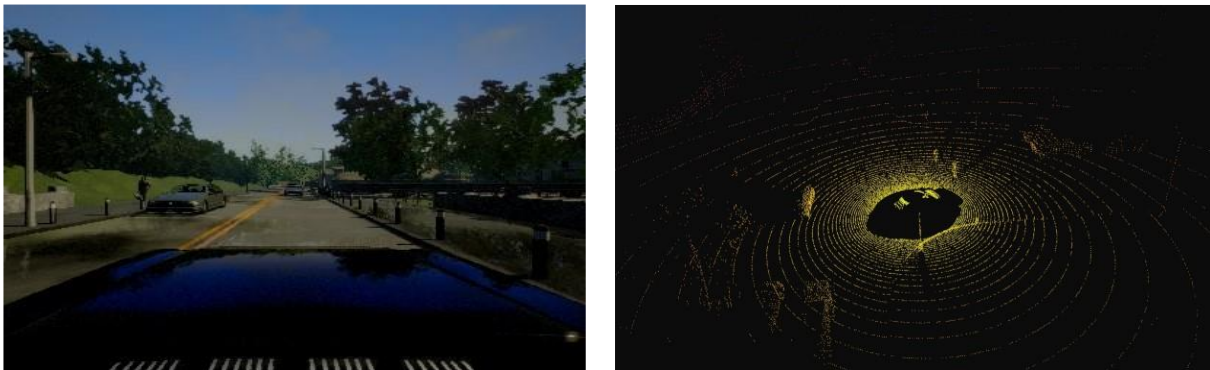


Figure 4 The RGB Camera Output (left) and LiDAR Sensor Output from CARLA - Sourced from [67]

Here is a summary of the key features and capabilities of the CARLA simulator:

Realistic Environments: CARLA offers 10 pre-built maps that mimic diverse urban, rural, and highway environments, allowing for training and testing autonomous driving agents in various scenarios.

Vehicle Models: The simulator provides a diverse array of high-fidelity vehicle models, including cars, trucks, and motorcycles, enabling realistic traffic simulation.

Pedestrian Models: CARLA includes a variety of pedestrian models to simulate foot traffic in the 3D environments.

Sensor Simulation: The simulator can simulate various sensors used in autonomous vehicles (AVs), such as cameras (with different projection models like perspective, fisheye, and equirectangular), LiDAR, radar, and GPS. The Figure 4 illustrates a sample RGB camera and LiDAR sensor output from CARLA.

Client-Server Architecture: CARLA follows a client-server architecture, where the server handles the simulation rendering and physics, while the client controls the logic and decision-making of the autonomous agents.

Python API: CARLA provides a Python API that allows users to control and interact with the simulation, enabling the development, training, and evaluation of autonomous driving algorithms.

Data Generation: The simulator can generate synthetic training data, including sensor data and ground truth information, which can be used for machine learning tasks related to autonomous driving.

Evaluation and Testing: CARLA serves as a safe and controlled environment for evaluating and testing the performance and safety of autonomous driving agents before deploying them in the real world.

The CARLA simulator is widely used in the autonomous driving research community, enabling researchers to develop, train, and validate their algorithms in a realistic yet risk-free virtual environment [63][68].

2.3 Automotive Safety-Critical Scenarios

A scenario depicts the temporal progression across multiple scenes in a sequence. Every scenario begins with an initial scene. Within a scenario, this temporal evolution may involve actions, events, goals, and values. Unlike a scene, a scenario unfolds over a specific duration.

A scenario space encompasses all the relevant situations that fit the same scenario description. The Operational Design Domain (ODD), within which the ego vehicle is expected to operate safely, constitutes a significant scenario space. ODD is defined as the operating conditions under which a given driving automation system or feature is specifically designed to function. This includes environmental factors like weather and lighting, geographical restrictions like urban or highway settings, time-of-day limitations, and the presence or absence of certain traffic or roadway characteristics. In essence, ODD delineates the operating environment boundaries within which an Automated Driving System (ADS) is intended to function safely and as designed.

The PEGASUS project, completed in June 2019, aimed to bridge significant gaps in testing up to Level 3 driving functions. The overall project goal was to develop a procedure for assuring and testing automated driving functions to enable rapid practical implementation [96].

The PEGASUS project introduced three levels of abstraction for describing scenarios for testing automated driving systems [96], as illustrated in Figure 5:

Functional Scenario represents the scenario space using natural language descriptions that are human-readable and understandable by experts, expresses scenarios in the terminology of the application domain and use case, and allows for creative formulation of scenarios by human experts without strict formalization.

Logical Scenario provides a formal representation of the scenario space by specifying parameter types and ranges of state values, describes the scenarios at the state-space

level, potentially including probability distributions for parameters, and captures relations between parameters, constraints, and dependencies.

Concrete Scenario represents a specific instance of a logical scenario by assigning concrete parameter values, serves as a parameterized representation that can be executed in simulations or real tests, and provides a distinct, reproducible definition of a scenario without room for interpretation.

The level of abstraction decreases from functional to concrete scenarios, while the number of scenarios increases. Functional scenarios allow human-friendly expression, logical scenarios enable formal modelling of the parameter space, and concrete scenarios represent executable instances for testing and validation.

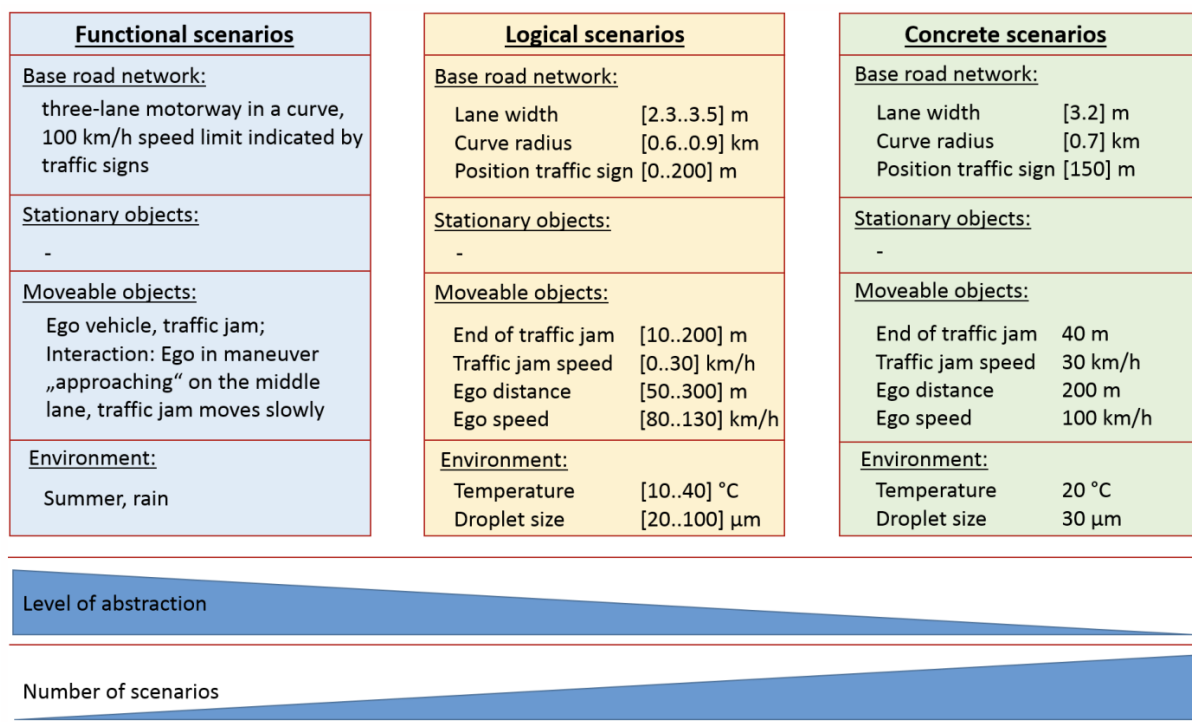


Figure 5 Three Levels of Abstraction for Testing Automated Driving Systems Introduced by the PEGASUS Project - Sourced from [96]

According to the PEGASUS project, scenarios for testing automated driving systems are described using a five-layer model [96], as depicted in Figure 6:

Layer 1: Street Level describes the geometry and topology of the road network and includes details like lane markings, road conditions, and boundaries.

Layer 2: Traffic Infrastructure covers static infrastructure elements like traffic signs, construction barriers, and traffic guidance systems.

Layer 3: Temporal Modifications represents temporary overlays or changes to the road layout and infrastructure (e.g., construction zones) and captures time-dependent changes lasting more than a day.

Layer 4: Movable Objects includes dynamic elements like vehicles, pedestrians, and animals and defines interactions and maneuvers between movable objects.

Layer 5: Environmental Conditions covers factors like weather (rain, fog, etc.), lighting, and time of day and models how environmental conditions influence properties across other layers.

This multi-layer approach provides a structured way to comprehensively describe scenarios by breaking them down into different components like static infrastructure, temporary changes, dynamic traffic participants and their behaviours, as well as environmental context. The layers build upon each other to create a complete representation of the scenario for testing automated driving functions.

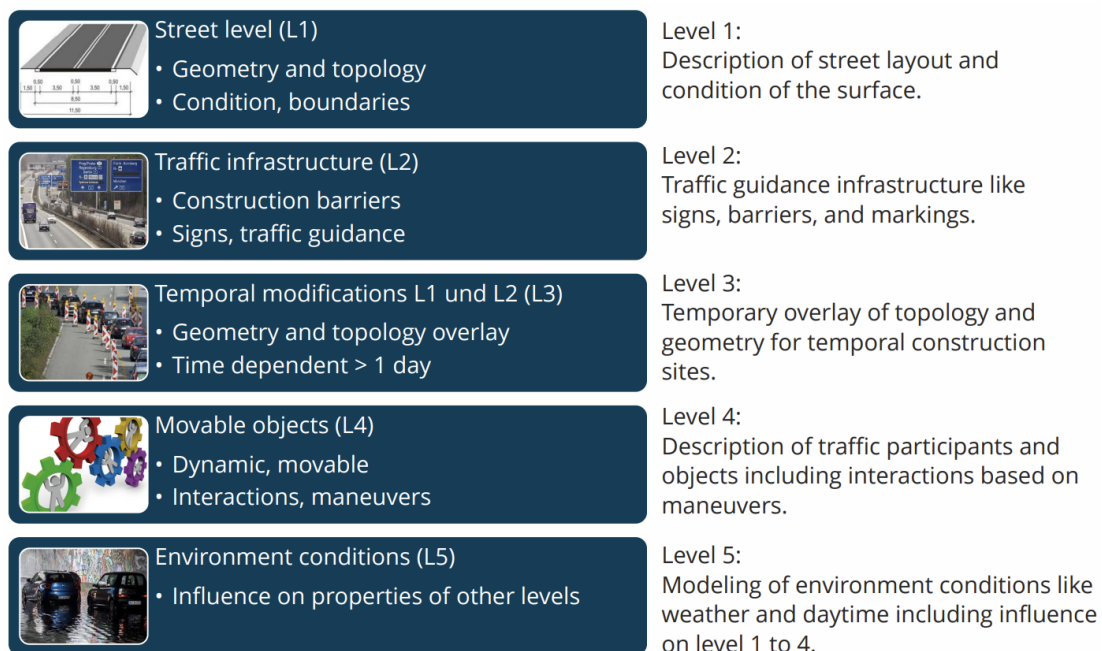


Figure 6 Five Layer Model for Description of Scenarios Introduced by The PEGASUS Project - Sourced from [96]

2.4 ASAM OpenSCENARIO and OpenDRIVE

ASAM OpenSCENARIO [71] and ASAM OpenDRIVE [74] are two complementary standards developed by ASAM (Association for Standardization of Automation and Measuring Systems) for describing scenarios and road networks, respectively, in the context of automated driving simulations and testing.

ASAM OpenSCENARIO defines a standardized format for describing dynamic driving scenarios involving various entities (vehicles, pedestrians, etc.) and their behaviors over time using an XML-based format (.xosc files), allows specifying scenarios at different levels of abstraction: functional (high-level description), logical (parameter ranges), and concrete (specific parameter values), enables the creation of complex scenarios through composition and parametrization of component scenarios, and aims

to facilitate massive testing and verification of automated driving systems by providing a standardized scenario description language.

ASAM OpenDRIVE provides a common base for describing the geometry of road networks, lanes, road markings, and roadside objects using an XML-based format (.xodr files) and describes either synthetic or real-world road networks to be used in simulations for development and validation of automated driving and ADAS features.

Together, these two standards enable a standardized and interoperable way of defining the static road environment (OpenDRIVE) and the dynamic scenarios (OpenSCENARIO) required for testing and validating automated driving systems through simulations [71, 74]. They are part of ASAM's broader efforts to establish standardized processes for the validation of automated driving.

3 Technology Analysis

In the next subsections, a detailed overview of the technological analysis required for this thesis will be discussed, providing comprehensive insights into the necessary technology.

3.1 Intended Test Infrastructure

As the automotive industry moves towards autonomous driving, incorporating Artificial Intelligence (AI) into software engineering and testing for these advanced vehicles has become crucial [64, 65]. The integration of AI across the software development lifecycle, from requirements management to deployment, is reshaping the approach to engineering autonomous vehicles (AVs). AI-based systems form the foundation of AV capabilities, encompassing perception, decision-making, control, and navigation. However, the unpredictable nature of AI-based systems presents unique validation and testing challenges [65]. Traditional software testing methods may prove inadequate, given the propensity of AI algorithms to exhibit unpredictable behaviours stemming from the extensive data they process. To address this issue, software engineers are adopting advanced techniques such as scenario-based testing, which involves simulating various real-world driving conditions to evaluate the vehicle's responses [66].

To facilitate testing of autonomous systems, the IAS research focuses on constructing a dedicated scenario-based test infrastructure, enabling product developers to select variants for testing. This necessitates the development of an intended test infrastructure, as depicted in Figure 7, wherein a System Under Test (SUT) controls a simulated car within a 3D environment (CARLA [67]) through a specified interface.

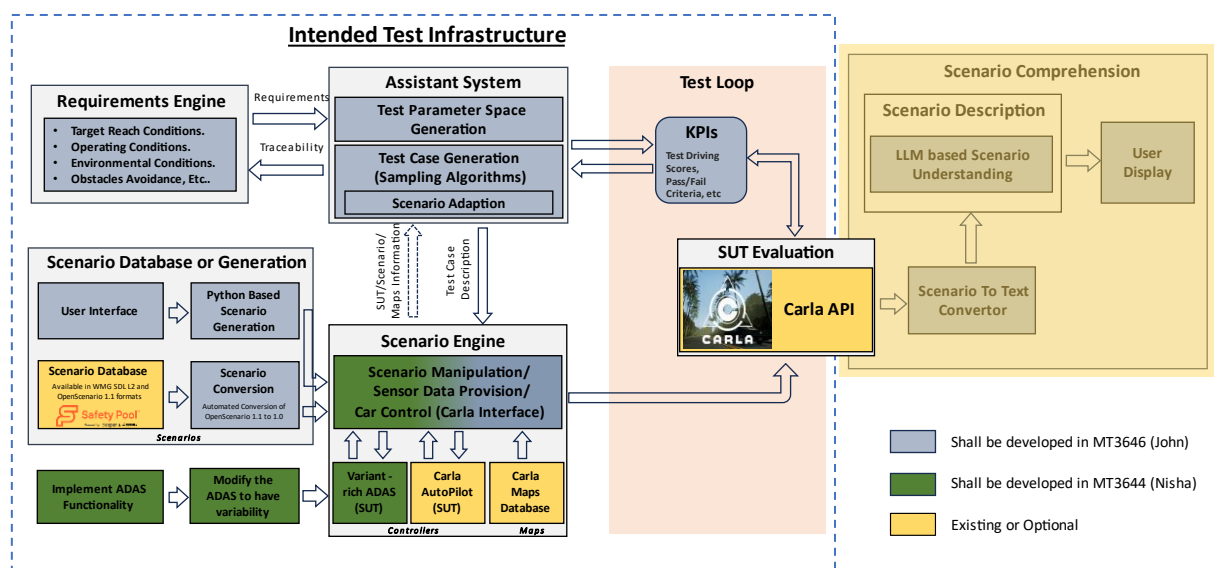


Figure 7 Revised Intended Test Infrastructure

The intended test infrastructure comprises seven components, which are explained briefly below:

3.1.1 Requirements Engine

This component defines the user requirements and functional scenarios for the evaluation of the SUT. Such include details such as map location, weather conditions, road actors involved (e.g., pedestrians, bicycles, and other vehicles), and the control methods of the SUT.

There are two key areas in requirement engine where traditional Machine Learning (ML), Deep Learning (DL), and Large Language Models (LLMs) techniques can be utilized: requirement engineering and requirement traceability. The below subsections define these two processes/areas in details.

3.1.1.1 Requirement Engineering - Related Works

Researchers have applied machine learning and deep learning techniques to enhance requirement engineering processes, focusing on two key areas: requirement extraction and requirement validation.

Requirement Extraction: In one study [52], the authors employed a deep learning technique, particularly Long Short-Term Memory (LSTM) with a Conditional Random Field (CRF) layer, utilizing an encoder-decoder architecture to generate a sequence for individual requirements. Subsequently, the relevant requirement entities were extracted from the sequence.

Requirement Validation: In another paper [53], the authors proposed an automated approach to identify and access methods for requirement validation. This approach aims to ensure that the developers have comprehensive understanding of the requirements and that these requirements conform to the company's standards.

By leveraging the power of ML and DL, these studies demonstrate how AI-based techniques can streamline and improve the critical tasks of requirement extraction and validation within the requirement engineering process.

In the field of requirement engineering, LLMs have demonstrated their potential to enhance various aspects of the process. This section explores the impact of LLMs on requirement retrieval, classification, ambiguity treatment, coreference detection, and completeness assessment.

Requirement Retrieval: A study [30] has shown that LLMs like ChatGPT exhibit strong capabilities in retrieving relevant information for requirements, but their performance in retrieving more specific information remains limited. Researchers have leveraged ChatGPT to enhance requirement retrieval methods, highlighting the potential of LLMs in this domain.

Requirement Classification: In a paper [28], the authors developed PRCBERT, a pretrained BERT model for accurate classification of software requirements into

functional and non-functional categories. PRCBERT achieved better classification performance on large-scale requirement datasets, demonstrating the potential of LLMs in this task.

Anaphoric Ambiguity Treatment: A study [31] introduced TABASCO, a tool designed to detect and identify ambiguities present in software requirements and other project-related documents. This tool leverages LLMs to address anaphoric ambiguities, a common challenge in requirement engineering.

Coreference Detection: In another paper [32], the authors developed DeepCoref, a fine-tuned BERT model capable of determining whether two entities are coreferent. This capability is crucial for ensuring consistency and clarity in requirement specifications.

Requirement Completeness: Researchers have explored using BERT to improve the detection of incomplete natural language (NL) requirements by predicting absent terminology [29]. This approach leverages the language understanding capabilities of LLMs to enhance the completeness of requirement specifications.

Comprehensive Requirement Engineering Approach: In a paper [33], the authors explained how to utilize LLMs in four essential sections of requirement engineering: requirement elicitation, specification, analysis, and validation. Meanwhile, another paper describes Prompt engineering guidelines for LLMs in Requirements Engineering [34].

Despite the improvements made in recent years to enhance requirement engineering, the impact of LLMs appears to be less noticeable compared to their profound influence on software development [26, 27].

3.1.1.2 Requirement Traceability – Related Works

Requirement traceability, the ability to link and track requirements across the entire development lifecycle, is a critical aspect of software engineering. Classical information retrieval (IR) methods, such as Vector Space Model (VSM) [14], Latent Dirichlet Analysis (LDA) [15] [16], and Latent Semantic Indexing (LSI) [17], have been traditionally used as an automated approaches for creating and changing links automatically between development lifecycle. These methods rely on word matching techniques. The highlight of relevant previous works on the use of various ML/DL and LLMs in requirement traceability can be found below.

The utilization of support vector machine (SVM) and decision trees for incorporating temporal dependencies and other process-related information into the tracing task yielded low accuracy in the generated trace links [54]. Consequently, the industry has been hesitant to adopt automated tracing solutions within their development life-cycles.

In response to this challenge, the author introduced an architecture based on the Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs), aimed at generating links between subsystem

requirements and design definitions through supervised learning technique [18]. While the RNNs resulted in good accuracy compared to the ML approaches, these models required large amounts of dataset for better performance and struggles to generalize on across diverse datasets.

The introduction of language models has led to significant advancements in requirement traceability. Fine-tuned language models have been employed, achieving higher performance with smaller datasets compared to traditional methods [19, 20]. However, this approach requires extensive computation and domain-specific pretraining.

To overcome this drawback, recently, a study [21] demonstrated the potential of using Claude, an LLM from Anthropic, for requirement traceability through prompt engineering. While not achieving the same level of performance as fine-tuned language models, this approach does not require extensive computation or domain-specific pretraining. The advantages of prompt engineering-based requirement traceability include reduced computational demands and the ability to adapt to various domains without extensive pretraining.

3.1.2 Assistant System

This component manages two processes, the test parameter space generation process and test case generation. It starts by analysing the requirements and creating logical scenarios from functional scenarios by incorporating relevant parameters to form a text-based scenario description. The component then generates test cases by selecting specific values from the parameter ranges using various approaches.

The test parameter space generation process converts the user requirements (functional scenarios) into test parameter space (logical scenarios), which are then used for test case generation to transform into concrete scenarios. For each iteration, the Assistant System provides a new test case for the SUT, which may or may not depend on the feedback from previous test results, depending on the techniques used.

3.1.2.1 Test Parameter Space Generation – Related Works

The test parameter generation process can be understood as transforming the requirements, written in an abstract or linguistic way, into a state space representation. In a recent paper, an automatic way to transform a keyword-based scenario description (functional scenario) modelled using the Web Ontology Language into a parameter space representation (logical scenario) is proposed [95]. However, this transformation does not involve any ML/DL or LLMs models; instead, it uses a programming language for the transformation. Apart from this, there is no predominant evidence of papers involving the conversion of functional scenarios to logical scenarios for testing and validation of autonomous systems using ML/DL or LLMs.

3.1.2.2 Test Case Generation – Related Works

The test case generation from the SUT test parameters can be understood as finding the optimal values from the range of values. Conventional models such as grid search, uniform search, and random search are initially used as optimizers for various use cases due to their simplicity, no tuning required, and the ability to run in parallel [1]. However, these models struggle to balance the exploration versus exploitation trade-off. Random searches prioritize uniform coverage, emphasizing exploration, while grid search exhaustively covers a given grid, emphasizing exploitation. Additionally, these models do not guarantee finding a local minimum to a specific precision unless the search space is thoroughly sampled, and they do not utilize the information gained from previous samples [1]. The highlight of relevant previous works on the use of various ML/DL and LLMs in the process of finding an optimal value from the range of values can be found below. These studies have explored the potential of advanced AI techniques to enhance the test case generation process.

Due to the above-mentioned limitations of conventional models, ML/DL techniques have gained popularity in various fields for optimization tasks. One popular ML/DL technique is the surrogate models [2]. Surrogate models (e.g., linear regression, neural networks, Bayesian optimization) have been used to solve problems that are difficult to optimize directly. These surrogate models have been applied in areas such as optimization of machine learning algorithms [3, 5], neural architecture search (NAS) [91], and hyperparameter optimization (HPO) [92]. While these models are effective in balancing exploration and exploitation and using feedback from previous results, they have a limitation in that they cannot be parallelized in their naive form, unlike random or grid searches. To address this, successive halving algorithms, such as Hyperband [6] and BOHB [7], have been introduced specific to HPO, which iteratively train machine learning models with different hyperparameters and resource budgets, removing poorly performing configurations and allowing the best ones to train further.

To further improve the trade-off between exploration and exploitation and address the limitations of surrogate and conventional models, researchers have introduced enhanced surrogate and conventional models, such as Guided Bayesian Optimization (GBO) and Random Neighbourhood Search (RNS), which have been used as scene generation samplers in automotive use cases [8]. Additionally, reinforcement learning-based methods have been explored for HPO, which have shown promising results in large search spaces with extensive training and unseen data [9].

In recent times, researchers have also focused on utilizing LLMs as optimizers. One study has compared the performance of OpenAI GPT-4, GPT-3.5, and Bayesian optimization algorithms for HPO on various ML/DL algorithms, with GPT-4 turbo outperforming the other algorithms [10]. Other papers have explored the use of LLMs for optimizing NAS [11, 12] and AutoML [13] cases. The main advantages of using LLMs for optimization tasks are their ability to adapt to various types of optimization problems, including continuous, discrete, and combinatorial optimization, and their potential to discover globally optimal solutions by exploring the entire solution space

more efficiently compared to local search methods. However, the limited context length and challenges in reproducibility remain significant limitations [10, 11, 13].

3.1.3 Scenario Database/Generation

This component enables the creation of diverse driving scenarios, encompassing real-world conditions such as traffic, weather, and other road actors' behaviour, within the CARLA simulation environment. This facilitates comprehensive scenario-based testing and validation of the SUT systems in a virtual environment.

Previous research has explored various approaches for generating scenarios within the CARLA simulation environment. These studies have examined different techniques and methodologies used to create diverse scenarios for testing and evaluating autonomous driving systems. The key findings and insights from the relevant prior work in this area are summarized below.

3.1.3.1 Scenario Database/Generation in CARLA – Approaches

The evaluation of autonomous vehicles (AVs) in CARLA simulation environment necessitates the generation of diverse and realistic test scenarios. Several approaches are available for this case but we have explained about the three main strategies that we find situation for running scenarios in CARLA simulation environment.

Safety Pool Scenario Database: The Safety Pool database is the world's largest public store of scenarios for testing automated vehicles, developed in the UK with global relevance [22]. It is based on the philosophy that the automated driving ecosystem should not compete on safety. The database features over 250,000 scenarios, covering diverse operational design domains (ODDs), and is underpinned by a credit-based scenario exchange mechanism to incentivize organizations to share test scenarios with each other. The Safety Pool Scenario Database is unique in its compliance with international standards such as ASAM OpenLABEL, ISO 34503, and ASAM OpenSCENARIO [22].

The numerous real-world scenarios are written in the WMG Scenario Description Language (SDL) Level 2 Language, which is also available in OpenSCENARIO 1.1 formats. These formats capture the dynamic behaviours of the scenarios, such as vehicle behaviours, spawning positions, and actions, as well as the static elements, including road details and traffic information. However, it is important to note that the Scenario Runner package in the CARLA simulation environment cannot accept the OpenSCENARIO 1.1 format directly. Thus, it has to be converted to OpenSCENARIO 1.0, but this means losing some of the detailed information that was in the original format.

Domain-Specific Modeling Languages (DSMLs): Domain-specific modeling languages (DSMLs), such as Scenic [23] and Paracosm [24], offer an alternative approach to scenario generation. These languages provide a simple and intuitive way to create scenarios that can be directly integrated into the CARLA simulation environment for

execution. By leveraging DSMLs, researchers and developers can define scenarios using a high-level, domain-specific language, abstracting away low-level implementation details.

Python-based Scenario Generation Package (pyoscx): The pyoscx package is a Python-based scenario generation tool that enables the creation of diverse scenarios using Python scripting [35]. This package generates OpenSCENARIO 1.0 formats, which capture the dynamic behaviours and static elements of the scenarios, respectively. The generated files can be used to run scenarios in the CARLA simulation environment through the CARLA ScenarioRunner package. In addition to scenario generation, pyoscx provides functionalities for modifying and visualizing OpenSCENARIO files. This versatility allows researchers and developers to iteratively refine and analyze their scenarios, facilitating the development and testing of AVs. The detailed view on creation of scenario will be discussed later in the Section 4 and 5.

3.1.3.2 Scenario Generation - Related Works

Recent advancements in natural language processing (NLP) and Large Language Models (LLMs) have opened up new avenues for scenario generation. The paper introduces ADEPT, a testing platform for simulated ADS built on the CARLA simulation environment [25]. The authors employ an approach that leverages NLP techniques and the question-and-answer capabilities of GPT-3 to create driving scenarios based on real-life accident reports. ADEPT can generate realistic and diverse test scenarios by processing these reports using the Scenic modeling language.

3.1.4 Scenario Engine

This component is responsible for the smooth linkage to execute the SUT for the simulation in the CARLA simulation environment. It utilizes the scenario file from the Scenario Database/Generation component, the test case description from the Assistant System, the map file, and the file needed to control the SUT. To control the SUT, the component explores two distinct approaches. The first approach utilizes the CARLA Autopilot, which the CARLA team has developed using imitation learning and reinforcement learning techniques [67]. The second approach involves the ADAS, which the parallel thesis (MT3644) is currently developing concurrently. The Simulation Integration component employs a Python-based interface to facilitate this seamless linkage, ensuring a smooth integration of the simulation environment with the SUT.

Prior research has investigated the use of various ML/DL and LLMs in the software development process. These studies have explored the application of such techniques for tasks like code generation, code completion, and other software engineering activities. The key findings and insights from the relevant previous work in this area are summarized below.

3.1.4.1 Software Development (CARLA Interface Development) - Related Works with ML/DL Techniques

Deep learning exhibits potential in automating code generation and summarization tasks. Regarding code generation, research has introduced deep learning-based approaches aimed at deriving source code from natural language requirements [60]. These techniques aim to automatically translate natural language descriptions into executable code. Additionally, other study has investigated the utilization of deep learning for program synthesis, wherein models generate code based on input-output examples [61]. As for code summarization, deep learning has been applied to automatically generate natural language summaries of source code [62]. However, considerable challenges persist deep learning approaches in generating high-quality, executable code and accurately capturing the complete semantics of source code.

3.1.4.2 Software Development (CARLA Interface Development) - Related Works with LLMs

LLMs have shown significant potential compared to the ML/DL techniques in enhancing various aspects of software development, some of the areas includes code generation, code completion, and code summarization. Here's a brief summary of how LLMs can be leveraged in these areas.

Code Generation: LLMs can be trained on vast amounts of code data to generate new code snippets or even entire programs based on natural language prompts or specifications. This capability can greatly accelerate the development process by automating repetitive coding tasks and enabling developers to focus on higher-level design and architecture decisions [26, 27]. Some recent works in this area are GPT4 [41], Copilot [42], and CodeGen [43].

Code Completion: LLMs can help developers by suggesting relevant code completions based on the context of the code they are writing. This can improve productivity by reducing the need for manual typing and remembering syntax details, especially in complex codebases. LLMs can also suggest alternative implementations or optimizations, enhancing code quality. Recent papers here include GPT4 [41], Copilot [42], and CodeGPT [44].

Code Summarization: LLMs can generate concise and readable summaries of code snippets or entire files, helping developers quickly understand the purpose and functionality of existing code. This can be particularly useful when working with unfamiliar codebases or when reviewing code written by others. Recent work in this area includes Codex [47], CodeBERT [46], and T5 [45].

While LLMs have shown promising results in these areas, it is important to note that sometimes they may make errors or show biases, so it is necessary for humans to validate their work, especially in critical applications. Researchers are still working on making LLMs better and more reliable for software tasks [26, 27].

3.1.5 System Under Test (SUT) Evaluation

This component encompasses the CARLA simulation environment and its corresponding APIs. Utilizing the Python-based interface of the Scenario Engine, simulations are conducted within this module. As it exclusively comprises the CARLA simulation environment and its APIs, no further discussion on ML/DL and LLMs is presented.

3.1.6 Key Performance Indicators (KPIs)

This component is responsible for evaluating the output of the simulation results from the SUT Evaluation component. It holds the metrics used to evaluate the SUT and provides a test result to the Assistant System to prepare for the next iteration.

The highlight of relevant previous works on the use of various ML/DL and LLMs in the mathematical calculations, a part of this component, can be found below.

3.1.6.1 KPIs (mathematical) calculations – Related Works with ML/DL Techniques

Amidst the deep learning era, remarkable progress has been achieved in the field of mathematical reasoning, due to its exceptional performance in transforming NLP tasks. A study showcased a deep neural solver that used RNNs to autonomously solve mathematical word problems [55]. Additionally, another study introduced ASTactic, a deep learning model capable of generating effective tactics and proving previously unprovable theorems through automated methods [56]. However, despite the impressive capabilities demonstrated of these models, there remains a lack of a clear taxonomy of the different types of mathematical reasoning tasks and specifying the essential capabilities required of deep learning models to solve them.

3.1.6.2 KPIs (mathematical) calculations – Related Works with LLMs

LLMs have demonstrated remarkable proficiency in various domains, including requirement engineering, requirement traceability, and software development as explained. However, their ability to perform mathematical calculations remains an area of active research and exploration. This section delves into some of papers the showcase the potential of LLMs in complex reasoning and mathematical problem-solving.

A study investigated the use of a “chain of thought” (CoT) approach to enhance the ability of large language models to perform complex reasoning tasks [59]. Their work argues that the CoT technique significantly improves the reasoning capabilities of LLMs by prompting them to break down complex problems into a series of intermediate steps, thereby facilitating more structured and interpretable reasoning processes.

Researchers have explored the potential of LLMs for solving mathematical problems. Recently, a study proposed a method called MathPrompter that utilizes the OpenAI Davinci model for solving math word problems [58]. Their approach involves

generating algebraic templates from the questions, providing multiple prompts to the LLM to solve the templates analytically, and verifying the solutions by evaluating them with random values.

While LLMs have shown promise in mathematical problem-solving, their limitations in tackling complex mathematical problems have also been highlighted. A study explored the challenges faced by current LLMs in solving intricate mathematical problems from the Math Stack Exchange website [57]. The authors identify areas where LLMs struggle, such as understanding complex mathematical notation, reasoning about abstract concepts, and providing step-by-step solutions.

3.1.7 Scenario Comprehension

This component is responsible for comprehending the scenario and providing reasoning to ensure accurate performance and responsiveness of the SUT.

Prior research has examined the use of various ML/DL and LLMs to enhance the understanding and comprehension of scenarios in order to improve the performance and efficiency of the SUT. The key findings and insights from the relevant previous work in this area are summarized below.

3.1.7.1 Scenario Comprehension - Related Works with ML/DL Techniques

In the context of the CARLA leaderboard [63], significant research has been conducted using deep learning approaches, such as ReasonNet [36] and Interfuser [37], to control the system-under-test under different scenarios and routes specified by the leaderboard. However, despite performing well, deep learning approaches encounter challenges in reasoning, interpreting, and memorizing complex scenarios akin to human capabilities.

3.1.7.2 Scenario Comprehension - Related Works with LLMs

Recently, LLMs have been explored for understanding scenarios and acting accordingly in the context of autonomous driving systems. This section highlights some notable works in this area. In a paper, the authors describe an interpretable end-to-end autonomous driving system that utilizes LLMs [39]. This approach aims to provide a transparent and explainable solution for autonomous driving by leveraging the language understanding capabilities of LLMs. Another study explores the potential of using an LLM to understand the driving environment in a human-like manner [40]. The researchers analyse the LLM's ability to reason, interpret, and memorize when facing complex scenarios. This work highlights the potential of LLMs in comprehending and reacting to dynamic driving environments, a crucial aspect of autonomous driving systems. By leveraging the language understanding and reasoning capabilities of LLMs, researchers aim to develop more robust and interpretable systems for controlling the SUT in various scenarios. These approaches have the potential to enhance the accuracy, responsiveness, and transparency of autonomous driving systems, ultimately contributing to safer and more reliable transportation solutions.

3.1.8 Research Intensity of the Components:

In the previous subsections, the related works on AI methods in each of the components in the intended test infrastructure were explained.

To find the amount of related works on each of the components and to find the novelty, four survey papers are referred. These survey papers examined the related works on the use of ML/DL and LLMs in software engineering, resulting in diverse conclusion across different areas of software engineering. The first two survey papers focused on ML/DL in software engineering, while the last two focused on LLMs in software engineering. The first survey paper, “A survey on deep learning for software engineering,” selected 250 DL papers related to DL for software engineering (SE) from 2015 to 2020, published in September 2022 [51]. The second survey paper, “Machine/deep learning for software engineering: A systematic literature review,” selected 1,209 ML and 35 DL papers related to ML/DL for SE from 2009 to 2020, published in May 2022 [50]. The third survey paper, “Large language models for software engineering: Survey and open problems,” selected 244 papers related to LLMs for SE and programming language (PL) from 2007 to July 2023, published in November 2023 [27]. The fourth survey paper, “Large language models for software engineering: A systematic literature review,” selected 229 papers related to LLMs for SE from 2017 to 2023, published in September 2023 [26]. It is important to that this table was generated based on the information available till March 20, 2024.

Although the survey papers provide a comparison of the potential improvements, research intensity on each component, it has two key limitations. Firstly, the latest LLM survey paper covered related works only until July 2023 [27]. However, there has been a significant amount of new research on the application of LLMs in software engineering since then. As a result, the conclusions drawn from the table regarding the intensity of LLM-related research and the performance of LLMs compared to ML/DL models may be outdated. Secondly, the table's conclusions are not specific to the components used in this thesis.

To address this limitation, a survey was conducted until March 20, 2024, with a specific focus on gathering related works related to the components in the intended test infrastructure. This approach aims to provide more current and component-specific insights into the comparison of ML/DL and LLMs-related research within each component.

In the subsequent subsection, a comprehensive overview of the survey methodology and its findings is provided. Towards the conclusion, a brief explanation on the primary focus of this thesis is underlined.

3.1.9 Survey

In this section, the research delves into the utilization of LLMs across eight distinct processes/components within the intended test infrastructure. It is noteworthy that requirement engineering and requirement traceability processes fall under the

requirement engine component, while test parameter space and test case generation are managed by the Assistant System. The SUT evaluation component, comprising solely the CARLA simulation environment, is excluded, resulting in a total of eight processes/components.

This analysis is crucial for two main reasons. Firstly, while previous survey papers provide an overview of LLM research in these components, they lack specific insights into each one. Secondly, these surveys only cover publications up to July 2023, excluding a significant portion of post-July 2023 literature. To overcome these limitations, a comprehensive survey of all publications up to March 20, 2024, was conducted, focusing specifically on these eight processes/components to understand LLMs research in detail.

To gather the data, a Python tool was developed using Google Scholar, adhering to its terms and conditions. Using specific search tags for each component, relevant information was extracted from Google Scholar. For example, for requirement generation, the search tag ‘The use of LLMs in requirement generation’ was used, and similarly for other processes/components. The publications were then filtered based on component-specific keywords in their titles or abstracts. The results are summarized in Table 1, showing the number of relevant publications retrieved for each component. Notably, due to Google Scholar's limitations, only the first 100 publications per search tag are considered.

Table 1 demonstrates differing research focuses across components. For instance, while the search tag for requirement generation yields 12 relevant publications out of 100 retrieved, the search tag for requirement traceability yields none. Similarly, search tags for test parameter generation and test case generation yield few relevant publications out of 100 retrieved. Adding specific keywords like “autonomous” or “vehicle” reduces relevant publications for test parameter and test case generation to zero, indicating a lack of literature in those contexts.

Table 1: Survey results on different components of intended test infrastructure from Google Scholar.

Component	G	S	S ∩ L ∩ C
Req. Gen (RG)	22500	100	12
Req. Traceability (RT)	17500	100	0
Test Parameter Gen. (TPG)	26200	100	7
Test Case Gen (TCG)	32700	100	9
KPI Calc. (KC)	19000	100	6
Code Gen. (CG)	30700	100	43
Scenario Gen. (SG)	23700	100	2
Scenario Compre. (SC)	17700	100	8

In the Table 1, G - total results from google scholar according to specific keyword. S - first 100 results scraped from the google scholar. $S \cap L \cap C$ - whose title or abstract included “LLM”, “Large language model”, “BERT”, “Claude”, “GPT” and Keywords from each component. Note that the year 2024 only included data up to 20th March 2024.

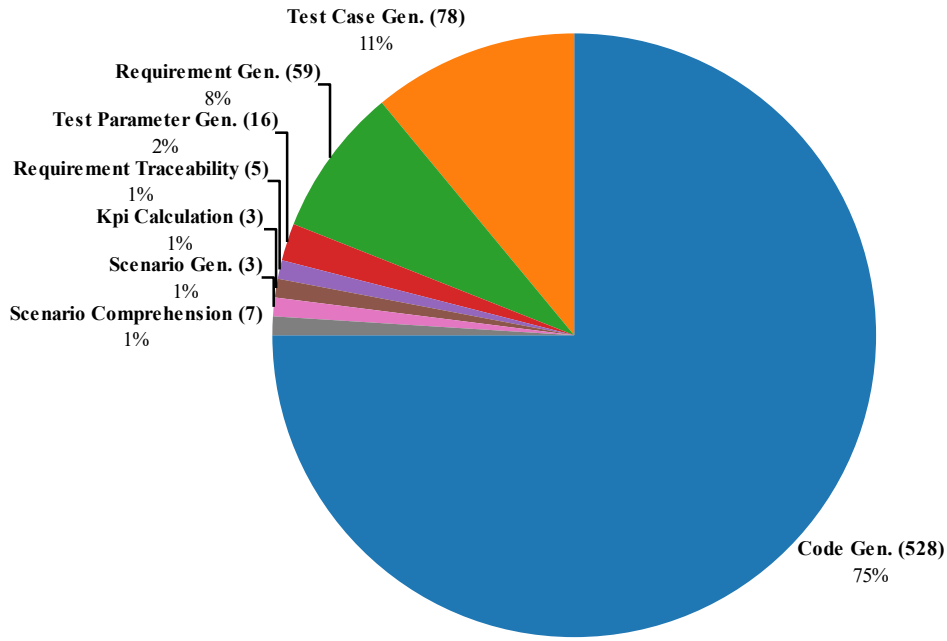
To overcome the limitations of Google Scholar in extracting large amount of publication details, arXiv is proposed as an alternative for precise counts, as it allows for the legal extraction of publication information and contains the most recent publications related to LLMs.

A Python tool was created using the arXiv Python API to extract relevant information related to this thesis. Initially, preprints categorized under cs.SE (software engineering) or cs.PL (programming language) from 1994 to 2017 were retrieved. These preprints were then filtered based on whether the abstract or title contained LLM-related keywords. The results are tabulated in the Table 2. Notably, 366 publications containing LLM-related keywords were found only in 2024, not considered in previous survey papers [26, 27].

Table 1: Survey results on papers from 1994 from 2024 from arXiv.

Year	S	$S \cap L$
1994 - 2017	6970	0
2018	1470	4
2019	1652	8
2020	2012	23
2021	2760	64
2022	2609	134
2023	3541	594
2024	1224	366
	22238	1193

In the Table 2, S - all preprints that are categorized under cs.SE or cs.PL. $S \cap L$ - preprints in cs.SE or cs.PL category whose title or abstract included “LLM”, “Large language model”, “BERT”, “Claude”, “GPT”. Note that the year 2024 only included data up to 20th march 2024.



*All components include papers from the preprints in the cs.SE or cs.PL categories from arXiv, whose title or abstract includes "LLM" and specific component keywords.

Figure 8 Number of Publications Across Different Processes/Components taken from arXiv

Further filtering of preprints is done based on specific keywords of the eight components in the intended test infrastructure, as shown in a pie chart shown in the Figure 8. The chart illustrates preprints under cs.SE and cs.PL with LLM keywords and keywords relevant to these eight components across different years. It is evident from the bar chart that the research intensity of the code generation component is relatively higher than the other seven components. In the case of LLMs in test case generation and test parameter generation, there are 16 (2%) and 78 (11%) relevant publications respectively. However, when keywords like "autonomous" or "vehicle" are added, relevant publications drop to zero, indicating a lack of literature on LLMs in test parameter or test case generation related to AVs.

Based on the survey focusing on the components discussed in this thesis, we have created a comparison table, shown in Table 3. This table provides a detailed comparison of the components within the intended test infrastructure, which we will use in the next subsection to determine the focus of this thesis: potential improvements (Low, Medium, or High), the intensity of ML/DL related research (Low, Medium, or High), the intensity of LLMs related research (Low, Medium, or High), and whether the LLM-related work has been proven to perform better than the ML/DL work in the relevant component (Low, meaning LLMs perform lower than ML/DL models, High, meaning LLMs perform higher than ML/DL models, or Unknown, meaning no evident exists).

Table 3: Comparison of the Components of the Intended test infrastructure based on the survey.

Components	Improvement potential	Research Intensity with other DL techniques	Research Intensity with LLMs	LLMs compared to other DL techniques' ability
Requirement Engineering	High	Low	Low	High [28][29][30][31][32][33][34]
Test parameter space generation (from RE)	High	Unknown	Low	Unknown
Test Case Description Generation (sampling methods)	High	High	Low	Low [10][11][12][13]
Requirement Traceability	High	Medium	Low	High [19][20][21]
Scenario Generation in Carla	Medium	No DL techniques (Scenic)	Low	Unknown
Carla interface Development	Medium	Medium	High	High [41][42][43][44][45][46][47]
Scenario Comprehension	High	High	Low	Low [39][40]

3.1.10 Decision on work of this thesis

In this thesis, the primary objective is to explore the use of AI-based techniques to generate test cases and scenarios for assessing the safety of a SUT in the CARLA simulation environment. The focus is on researching the utilization of ML/DL and LLMs in the Assistant System component.

Specifically, the thesis investigates the scenario generation, adaptation and manipulation which involves the generation of test parameter spaces from requirements and the generation of test case descriptions, which have seen limited to no research in the past. As mentioned in the previous subsections, the test parameter space generation or state space representation from the functional scenario is done automatically using a programming language. However, this transformation requires modelling the functional scenario in the Web Ontology Language, but no evidence of papers involving the conversion of functional scenarios modelled using simple

linguistic descriptions to the test parameter space with the help of ML/DL and LLMs has been found. Therefore, this thesis focuses on using LLMs rather than ML/DL algorithms for the test parameter space generation process, as LLMs have shown promising results in understanding text-based descriptions. For the test case generation (sampling methods), the thesis presents a comparison of different sampling methods, evaluating their performance, execution time, feedback utilization, proven use cases, parallelization capabilities and techniques useability on this thesis use case as illustrated in Table 4.

The search space in this thesis is relatively restricted, aiming to generate test cases that maximize SUT failures rather than solving a hyperparameter optimization (HPO) problem. The analysis reveals that conventional models often struggle to learn from previous results, resulting in subpar performance. Reinforcement learning-based optimization methods require extensive datasets and extensive time for optimal performance to learn the behaviour of the system, which are not feasible in our scenario [9]. Successive halving methods and hybrid approaches like hyperband and BOHB exhibit superior performance compared to other models, but they are tailored specifically for HPO tasks, rendering them unsuitable for our objectives [6, 7].

Surrogate models [2, 3, 4, 5], enhanced surrogate models [8], and large language models [10, 11, 12, 13] emerge as ideal candidates for our objectives. These models excel within limited spaces and possess the capability to learn from previous feedback, without being constrained to HPO tasks. To summarize, this thesis investigates the generation of test parameter spaces from requirements and the generation of test case descriptions mainly using LLMs. However, as a precautionary measure, surrogate models and conventional models such as Bayesian optimization and random search will be utilized as backups, considering that the effectiveness of LLMs in our specific use case has not been conclusively established. More details on this will be found in Section 4.

In requirement engine component, previous research shown above has demonstrated the potential of ML/DL and LLMs to support various aspects of requirement engineering. However, the focus of this thesis is not on the further utilization of ML/DL and LLMs in the requirement engineering process. Instead, the primary objective is to generate test cases using AI techniques. Since the thesis requires functional scenarios or descriptions, the user will provide these details, including information about scenarios, maps, vehicles, and weather. Consequently, the generation or extraction or validation of requirements using ML/DL and LLMs techniques are not further explored in this thesis, despite the potential improvements and promising results from previous related works [26, 27]. Also, even though LLMs have shown effectiveness in tracing requirements by linking and tracking them throughout the development lifecycle, this thesis does not delve deeper into this aspect.

Table 2: Comparison of different approaches for the Test Case Generation.

Approaches	Algorithms/ Models used	Perform ance in a limited search space	Fee dba ck	Proven Use Case	Execution time for a fixed iteration	Paralle- lization	Technique Usability Score
Conventional Models [1]	Random, Grid search	Low	No	NAS, HPO, Other	Low	Yes	High
Surrogate Model [2] [3] [4] [5]	Bayesian Optimization	High	Yes	NAS, HPO, Other	High	No	High
SOTA Hyper- parameter tuning algorithms [6][7]	Hyperband, BOHB	Excellent	Yes	HPO	Low (due to parallelizat ion)	Yes	Low
Enhanced surrogate and exhaustive models [8]	RNS, GBO	High	Yes	Other	High	No	High
Reinforcement Learning [9]	LSTM based RL Agent	Low	Yes	HPO	Unknown	No	Low
Large language Model (LLMs) [10][11][12][13]	OpenAI LLMs, PaLM	High	Yes	NAS, HPO	Depends on the LLM response time	NA	High

In the Scenario Database/Generation component, this thesis focuses of creation of scenarios which are required to execute the SUTs as the primary objective is to develop an AI-driven test case for validating the SUT within the CARLA simulation environment. Initially, the plan was to utilize scenarios from the Safety Pool database via a pipeline established by research project RP3488. However, since the pipeline details were not provided by the research project RP3488, a new pipeline was created within this thesis to execute the scenarios from the Safety Pool database.

The problem with using the Safety Pool database is that the numerous real-world scenarios are available only in OpenSCENARIO 1.1 formats. Unfortunately, the Scenario Runner package, a Python package developed by the CARLA community for running OpenSCENARIO in the CARLA Simulation Environment, cannot directly handle the OpenSCENARIO 1.1 format. Therefore, conversion to OpenSCENARIO 1.0 is necessary. However, this conversion using the new pipeline revealed the loss of some detailed information present in the original OpenSCENARIO 1.1 format.

As scenarios are indispensable, a small part of this thesis is on creating a Python - based scenario generation utilizing the pyoscx package which can then be executed in the CARLA simulation environment. Further elaboration on this topic can be found in Section 4 and 5.

In Scenario Engine component, the thesis does not focus on creating the CARLA interface (code generation) using LLMs, which could be a possible future work. Instead, this thesis uses the Python language to implement the interface between the CARLA APIs and the thesis codes. This approach aims to achieve the objective of exploring AI-based generation of test cases and scenarios to assess the safety of the SUT in the CARLA simulation environment.

In KPI component, this thesis does not investigate improving the mathematical calculation of LLMs due to its complexity along with the short time constrain, which could be a possible future work. Instead, this thesis uses the Python language to calculate the necessary KPIs using the test feedback from the CARLA simulation environment.

In Scenario Comprehension component, this thesis considers scenario comprehension section as out of scope or as possible future work, even though this field shows higher potential for improvement and has seen less research work from the Table 3.

3.2 Comparison on Large Language Models

Large Language Models (LLMs) are advanced artificial intelligence systems designed for understanding and generating text resembling human language. With millions or billions of parameters, they utilize deep learning to excel in tasks such as text generation, translation, and summarization. LLMs employ architectures like transformers, allowing them to capture intricate patterns and relationships within text. Examples like GPT and BERT have brought about a revolution in various language-related applications, showcasing their adaptability and versatility across diverse domains. These models have transformed language-related tasks significantly.

3.2.1 Key Considerations for Choosing Large Language Models

The following are the key aspects that are considered when choosing large language models for any applications.

1. Categories of Large Language Models:

Encoder-only models, such as BERT, RoBERTa, and DeBERTa, which encode input sequences into lower-dimensional representations [25].

Encoder-decoder models, like T5 and BART, which generate output sequences based on context vectors and previously generated tokens, useful for tasks like machine translation [25].

Decoder-only models, such as the GPT series, Llama [84], Claude, and PaLM, which directly generate output sequences based on given context, typically token-by-token, without an encoder component [25].

2. **Number of Parameters (Size):** The size of a language model, indicating the number of parameters it encompasses, affects its understanding of complex language patterns. Larger models with millions or billions of parameters exhibit greater capacity but come with increased computational demands, influencing factors such as training time, memory requirements, and overall efficiency in deployment.
3. **Context Length:** Context length, representing the maximum number of words or tokens a language model can consider at once, is crucial for applications requiring an understanding of longer text passages. Selecting a model with an adequate context length ensures nuanced comprehension for tasks like document summarization or dialogue systems.
4. **License:** The open-source nature of a model determines accessibility for modification. Open-source models offer transparency and flexibility, allowing users to customize the model. Conversely, proprietary models may provide unique features but limit customization. The choice depends on the level of control and adaptability required for the application. Regarding the license, two distinct categories are outlined:

Permissive License: Models falling under this category possess licenses that are accommodating, permitting all forms of usage with the pretrained model, including both research and commercial applications.

Closed License: Models classified under this group are governed by highly restrictive licenses, allowing only for research or private use.

5. **Cloud or On-Premises:** Choosing between deploying a model in the cloud or on-premises depends on factors like scalability, flexibility, and ease of management. Cloud-based models are accessible only through an API, but they can raise concerns about data security and privacy. There is no assurance that the data inputted into the model won't be used for retraining, potentially compromising the privacy of sensitive information. In contrast, on-premises deployment offers more control and comprehensive data protection. However, it requires infrastructure management and may have scalability limitations.
6. **Fine-Tuned or not:** Fine-tuning, allowing adaptation of a pre-trained model to a specific task or domain, provides a middle ground between using a generic model and training from scratch. Models supporting fine-tuning are valuable for applications requiring domain-specific knowledge or customization, ensuring flexibility to tailor the model to specific requirements.
7. **Training Data Size:** The size of the training dataset directly influences the model's ability to generalize across scenarios. A larger and more diverse training dataset contributes to improved performance and robustness. Understanding the scale and diversity of the training data is vital for assessing how well the model handles a variety of inputs and scenarios.
8. **Core-Knowledge Metrics:** Large Language Models (LLMs) are essentially NLP models, and common NLP accuracy metrics like BLEU, METEOR, ROUGE, CIDEr, SPICE, and Perplexity serve as fundamental points of evaluation, particularly for pre-trained models. These metrics play a crucial role in assessing LLM performance across various NLP tasks, ensuring accuracy and effectiveness measurement.

Perplexity: A measurement of how well a probability distribution of a test sample matches with the corresponding LLM prediction. It is commonly used for well-defined LLM tasks such as Question-Answering.

BLEU (Bilingual Evaluation Understudy): A metric for machine translation, commonly used in NLP tasks. It measures the similarity of machine-generated text with human-produced references.

METEOR: An automatic metric for machine translation evaluation based on unigram matching between machine-produced and human-produced translations.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation): A set of metrics for evaluating automatic summarization of texts and machine translations.

CIDEr/SPICE: Metrics used for image captioning tasks, they are based on the concept of n-gram similarity between machine-generated and human-produced captions.

A comprehensive LLM is task-specific and context-dependent; thus, evaluation varies for tasks such as medical summarization and consumer support. Consequently, LLM evaluation necessitates alignment with task and domain-specific benchmarks.

The HuggingFace LLM Leaderboard assesses models using four key benchmarks from the Eleuther AI Language Model Evaluation Harness [93], a unified framework for testing generative language models across diverse evaluation tasks.

AI2 Reasoning Challenge (ARC 25-shot): Evaluates LLMs' reasoning ability using grade-school science questions, employing a few-shot learning approach with 25 examples provided before testing.

HellaSwag (10-shot): Tests common sense inference, challenging for models but easy for humans, utilizing a few-shot learning task with 10 examples before testing.

MMLU (5-shot): Measures a text model's multitask accuracy across 57 tasks, including various subjects. Utilizes a few-shot learning approach with 5 examples before testing.

TruthfulQA (0-shot): Assesses a model's tendency to reproduce falsehoods commonly found online, using a zero-shot learning task where the LLM is not given any examples before testing.

These benchmarks gauge LLMs' ability to reason, comprehend concepts, generate common sense inferences, and evaluate truthfulness in generated text, crucial for real-world performance measurement. Averaging these metrics provide a balanced overview of LLMs' capabilities across tasks and data settings. This averaged score synthesizes performance across dimensions, offering a consolidated quality rating that comprehensively assesses models' effectiveness in diverse linguistic challenges. Additional LLM benchmarks include TriviaQA, BoolQ, SIQA, OpenBookQA, GLUE, Big-Bench, among others.

9. **The LMSYS Chatbot Arena Leaderboard:** The LMSYS Chatbot Arena Leaderboard is a platform for ranking LLMs using crowdsourced human evaluation and the Elo rating system. It allows users to vote on which LLM they find most engaging, informative, or helpful in specific conversations, leading to dynamic adjustments of the LLMs' scores in Elo-Scale [73].

In the next subsection, a comparison of LLMs from popular providers such as OpenAI, Meta, Google, HuggingFace and Mistral etc are put together.

3.2.2 Decision on LLMs for this thesis

The Table 5 presents a comprehensive analysis of the performance of various Large Language Models (LLMs) offered by leading companies. The findings indicate the existence of high-performing models, but it is noteworthy that certain organizational developments remain inaccessible to the public. Using non-public (close-source) models might lead to cloud-based storage and potential data security issues. While

collaboration with organizations for high-performance models is an option, it often comes at a higher cost.

In this thesis, the intended use of LLMs involves converting the user requirements (functional scenario) into test parameter spaces (logical scenario), and addressing optimization in converting the test parameter spaces (logical scenario) into the test case description (concrete scenario), which are explained in detail later in Section 4. These applications do not involve proprietary information, making them suitable for use with non-public models without exposing sensitive data. Initially, GPT-3.5 [81] and GPT-4 [41] from OpenAI will be utilized as a proof of concept due to their significant performance, as shown in Table 5. Once successful, public (open-source) models will be preferred to avoid additional costs. Note, the Gemini models [82, 83] from Google are not considered as due to the time constraints.

Among public models, Meta's Llama-2 model, with an open-source-friendly license, has notably influenced the open-source community [84, 85]. Following this, several models with permissive licenses for research and commercial use have been released, as indicated in the Table 5. Some LLMs are the finetuned version of Llama-2, such as Orca-2. Orca-2 model's training data is a synthetic dataset that was created to enhance the small model's reasoning abilities [76]. Notably, Mistral AI's Mixtral-8x7B has emerged as a standout performer, surpassing even the renowned open-source Llama model and OpenAI's GPT-3.5 [87].

Mixtral-8x7B, similar to Mistral-7B [86], incorporates 8 “expert” models using a technique called mixture of experts (MoE). The MoE approach involves replacing certain Feed-Forward layers with a sparse MoE layer, which contains a router network efficiently selecting experts for processing tokens. In Mixtral's case, two experts are chosen for each timestep, enabling the model to decode at the speed of a 12B parameter-dense model during inference, despite containing 46.7B total parameters. Even though Mixtral-8x7B showcases overall strong performance and its open-source nature [87], the hardware required to run it on-premises is not available in IAS.

Recently, Meta introduced Llama-3 with 8B and 70B parameter models [90]. These model shows promising results in most benchmarks compared to competing models of comparable size [90]. While the Llama-3-8B model does not perform as well as the 70B version, it can run on the hardware available at IAS. However, to check the performance of Llama-3-70B model, a cloud-based GPU is rented solely due run the Llama-3-70B model, more details in Section 6. Therefore, along with the Llama-3-8B and 70B parameter models, other open-sourced models such as the Mistral-7B, and WizardLM-13B are used for better comparison, as they require less hardware than the Mixtral-8x7B model. However, these models are not expected to perform as well as the Mixtral-8x7B model. Additionally, it is important to note that open-source models such as Falcon [80], Zephyr-7B [88][89] and Gemma [49] are not considered due to time-constraints even though these models' performance is comparable to the once that are selected in this thesis.

To generate the desired output from the LLMs, several techniques are used, and some techniques adjust the model's behaviour to meet the needs, while others improve how we instruct LLMs to yield more refined and pertinent data. Widely used techniques include Retrieval Augmented Generation (RAG), Prompting, and fine-tuning.

Prompt engineering involves crafting carefully designed prompts to guide the LLM and influence its outputs. By providing specific instructions and context within the prompt, developers can elicit more accurate, relevant, and coherent responses from the model. In contrast, RAG combines language modelling with information retrieval to enhance the LLM's capabilities. It allows the model to access external knowledge sources and incorporate relevant information into the generated outputs. Finally, fine-tuning involves further training the pre-trained LLM on task-specific data to adapt it for particular applications. More details of these techniques are mentioned in the previous literature work section.

In this thesis, LLMs are not used for accessing to external knowledge sources. Therefore, RAG techniques are not considered. Fine-tuning a LLM for this specific purpose would require an excessive dataset, excessive hardware requirements, and a significant amount of time, even though it could result in considerably higher performance compared to prompting techniques. Due to time and hardware resource constraints, fine-tuning techniques to improve the performance of a LLM are not considered in this thesis. Instead, prompt engineering techniques, including advanced methods such as zero-shot, one-shot learning, and chain-of-thought, are used to generate the desired output from the LLMs.

To summarize the decision on LLMs, OpenAI models such as GPT-3.5 and GPT-4 will be considered first. Once proven efficient and if time permits, open-source models such as Llama-3-8B and Llama-3-70B [90], Mistral-7B [86], and WizardLM-13B [79] will be downloaded and run on-premises. More information on the models used can be found under Section 6. It is important to note that the instruct or fine-tuned versions of the open-source models are considered, not the base or pre-trained models. Unless otherwise mentioned explicitly elsewhere in this thesis, prompt engineering with advanced techniques such as zero-shot [78], few-shot learning [48, 77], and chain-of-thought [59] are used throughout.

It is important to acknowledge that these results in Table 5 are current as of the writing of this thesis. The rapidly evolving landscape of language models suggests that ongoing reconsideration and re-evaluation should be integral components of future research, given the anticipated development of newer and more advanced models in the coming months and years.

Table 3: Comparison on different popular LLMs along with their open-source evaluation metrics

Large Language Models	Company	Cloud / On-Premises	Metrics Average (Arc, HelloSwag, MMLU, TruthfulQA) [93]	Metrics (LMSYS Arena Elo Score) [73]	Refs
Llama-2 – 70B	Meta	On-Premises	80.7	1089	[84][85]
Gemma – 7B	Google	On-Premises	64.29	1084	[49]
Orca-2 – 13B	Microsoft	On-Premises	61.98	-	[76]
Llama-3 – 8B	Meta	On-Premises	66.87	1154	[90]
Llama-3 – 70B	Meta	On-Premises	77.88	1208	[90]
Mistral – 7B	Mistral AI	On-Premises	65.64	1012	[86][87]
WizardLM – 13B	WizardLM	On-Premises	61.25	1062	[79]
Mixtral-8x7B Instruct	Mistral AI	On-Premises	81.03	1114	[87]
Zephyr – 7B	HuggingFace	On-Premises	66.23	1054	[88][89]
Falcon – 180B	Tiiuae	On-Premises	68.57	1037	[80]
GPT-3.5	OpenAI	Cloud	80.23	1108	[81]
GPT-4 Turbo	OpenAI	Cloud	90.85	1258	[41]
Gemini Ultra	Google	Cloud	88.90	1249	[82][83]

4 Conceptual Work

In the technology analysis section, the thesis provides an overview of the seven components in the intended test infrastructure, along with the related work in terms of ML/DL and LLMs. Additionally, an overview of the thesis is briefly presented. In this section, the thesis delves into the details of the methodology. The main goal of this thesis is to generate critical automotive scenarios and adapt/manipulate those generated scenarios to induce failure in the SUT (ADAS or CARLA AutoPilot) by changing the weather conditions (e.g., sun, fog, or rain) and the actors' characteristics (e.g., actions, spawning position, control parameters, start condition, and stop conditions) related to that scenario.

In the following subsections, the thesis explains in detail the methods behind the scenario generation, and scenario adaptation and manipulation processes.

4.1 Scenario Generation in Carla

CARLA is an open-source autonomous driving simulator widely used by researchers and developers for scenario generation during the development and testing of self-driving car systems [67]. CARLA allows users to create diverse and realistic driving scenarios, which is a crucial aspect in the development and testing of self-driving car systems [68]. By simulating a wide range of driving conditions, including complex maneuvers, unexpected events, and diverse environmental factors, self-driving car systems can be thoroughly tested and refined before deployment on public roads. Effective scenario generation enables the identification of edge cases, the validation of safety-critical functionalities, and the iterative improvement of the autonomous driving capabilities. Ultimately, the quality and diversity of the generated scenarios directly impact the reliability and trustworthiness of the self-driving car systems, making scenario generation a fundamental component in the development and testing process [69].

There are various possibilities to generate scenarios in CARLA for the development or testing of autonomous systems, such as using probabilistic graphical models [68], scenario definition languages [70] etc. However, supported scenario definition languages play a key role in scenario generation within the CARLA autonomous driving simulation environment. The use of scenario definition languages like OpenSCENARIO [71] and Domain-Specific Modelling Languages (DSMLs) like Scenic [23] and Paracosm [24] can streamline the process of creating and managing driving scenarios in CARLA, compared to manually defining the scenarios from scratch.

4.1.1 Scenario Definition Languages

OpenSCENARIO is a standard for describing complex driving maneuvers involving multiple vehicles and other dynamic elements in a driving scenario, providing a structured way to define the various components of a driving scenario. In contrast, DSMLs involve the use of probabilistic programming languages to streamline the scenario generation process in CARLA.

The scenario definitions from both OpenSCENARIO standard and DSMLs can be integrated within the CARLA simulation environment. The scenarios are described using a five-layer model in the PEGASUS project [96]. The OpenSCENARIO standard facilitates the definition and execution of dynamic elements of the scenarios, namely layer 4 and layer 5, whereas the OpenDRIVE standard facilitates the definition and execution of static elements of the scenarios, namely layer 1, 2, and 3, in CARLA simulation environment. While DSMLs provide an interface for executing their own scenario definitions directly within the CARLA simulation environment. However, a limitation of DSMLs is the necessity to learn probabilistic programming languages, although they provide a concise syntax for specifying spatial and temporal relationships, making it easier to define complex driving situations. In contrast, the OpenSCENARIO standard format critical automotive scenarios are available from the Safety Pool database [22] or can be generated using the Python-based pyoscx package [35] based on the EURO NCAP specification, making it the preferred choice for the CARLA simulation environment.

The Safety Pool database [22] contains numerous real-world scenarios written in the WMG SDL Level 2 Language, which is also available in OpenSCENARIO 1.1 formats. However, the CARLA simulation environment cannot accept the OpenSCENARIO 1.1 format directly, and a conversion pipeline is required to execute these scenarios in CARLA. While a conversion pipeline was part of research project RP3488, lacking pipeline details prompted the creation of a new fully automated conversion pipeline within this thesis to execute scenarios from the Safety Pool database. Nonetheless, this conversion revealed the loss of detailed information present in the original OpenSCENARIO 1.1 format, rendering scenarios from the Safety Pool Database insufficient for the thesis due to information loss during the conversion process. Due to this issue in Safety Pool Database, the critical automotive scenarios used in this thesis are generated using pyoscx package [35] with the Euro-NCAP specifications.

The pyoscx package [35] is a Python-based scenario generation tool that enables the creation of diverse scenarios using Python scripting. This package has the capability to generate OpenSCENARIO 1.0, 1.1, and 1.2 formats, which capture the dynamic behaviours and static elements of the scenarios, respectively. Due to the package's diverse OpenSCENARIO format creation and the use of Python language to create scenarios, a part of this thesis focuses on creating a pyoscx package-based critical automotive scenario generation that can then be executed in the CARLA simulation environment.

4.1.2 Simulation of Scenarios in CARLA

Scenario-based simulation is a crucial approach for evaluating autonomous driving systems in the CARLA simulation environment. The main components enabling scenario-based simulation in CARLA are ASAM OpenDRIVE [74], ASAM OpenSCENARIO [71], and the Scenario Runner Tool [72], as illustrated in Figure 9. ASAM OpenDRIVE is a complementary standard that defines the map layout, including the static road network and infrastructure in CARLA, using XML syntax (file extension '. xodr'). The maps used are the Town01 (built-in CARLA map) and Karlsruhe maps (developed by a team at KIT). In contrast, ASAM OpenSCENARIO is a standard format for describing dynamic critical driving scenarios, including vehicle maneuvers, traffic behaviors, and environmental conditions, using XML syntax (file extension '. xosc'). The critical automotive scenarios are generated using the pyosxc Python package in an OpenSCENARIO 1.0 format. Upon providing the OpenSCENARIO and OpenDrive files, the Scenario Runner Tool [72], a separate repository from the main CARLA package, functions as an execution engine for executing simulations directly within the CARLA environment. It runs the simulation and tests the SUT controlled by CARLA AutoPilot (built-in CARLA simulation environment feature) or ADAS (being developed concurrently by MT3644) in that particular scenario.

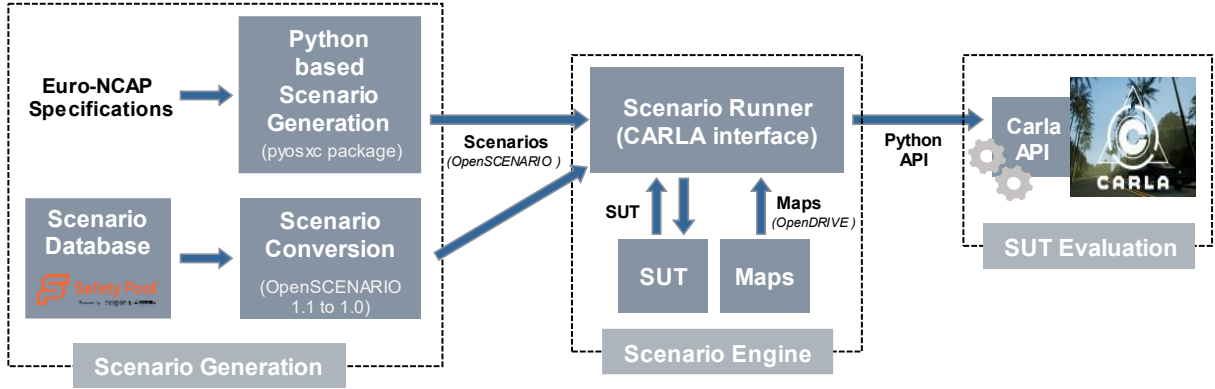


Figure 9 Overview of the Scenario Simulation in CARLA

With a comprehensive understanding of the necessary information for generating critical automotive scenarios and their simulation within the CARLA environment, we can now delve into the process of testing the SUT based on the scenarios determined by user requirements. To efficiently test the SUT under the conditions specified by the user, the first step involves generating the test parameter space from the user requirements, which will be briefly explained in the following subsection.

4.1.3 Test Parameter Space Generation Process

The test parameter space generation process in the Assistant System component takes the user requirement or functional scenario from the Requirements Engine component, representing the scenario space at a language level using linguistic scenario notations. The user requirement or functional scenarios provided by the user includes the scenario details, map details, controller details, and weather details for

the testing of SUT. It then converts the functional scenario into a logical scenario, a state-space level depiction of the scenario space with parameter ranges for moving objects and environmental conditions, by employing LLMs.

As illustrated in Figure 10, the conversion of the functional scenario to logical scenarios takes place in two parts: conversion of the functional scenario to an intermediate JSON file using the LLMs and conversion of the intermediate JSON file to the logical scenario using an automated Python script. In this case, the functional scenario is written in the English language within 1-2 sentences. Since LLMs have proven to be efficient in understanding the English language, as shown in the technology analysis section, the open-source LLMs as well as closed-source LLMs are selected for this task. The job of the LLMs is to retrieve the information such as scenario details, map and road details, weather details, controller details, and actors' details as a JSON file from the functional scenario, which is then used by the automated Python script to output the logical scenario.

In the example shown in Figure 10, the functional scenario provided by the user wants to test the SUT or hero vehicle or ego vehicle in the Karlsruhe map along with a slow-moving bicycle. The hero vehicle is mentioned to be controlled by the CARLA AutoPilot. The LLMs understand the functional scenario and retrieve the details of the map, road, scenario, controller, and actors involved in this case. The scenario has to be strictly within the scenario generated within the Scenario Database/Generation components, as mentioned in the scenario generation subsection previously.

In this case, the LLM understands from the functional scenario that the scenario the user wants to test is “hero_and_bicycle_on_the_same_lane,” and the actors involved are the hero vehicle, and bicycle, the map is Karlsruhe, the road is a single-lane road, and the controller is CARLA AutoPilot. The LLM provides this information as a JSON file. However, the user has not provided the weather details under which the SUT has to be tested, and thus the LLMs take “sun” weather as the weather details are important for running the scenario in the CARLA simulation environment.

The Automated Python script takes the intermediate JSON file and selects the test parameters associated with the actors involved and the weather from a logical scenario template. Since the actors involved in this scenario are the hero vehicle and bicycle, both “hero vehicle” and “bicycle” would have their own test parameters as a logical scenario, and “sun” weather would have its own test parameters as a logical scenario. Due to space constraints, only the hero test parameters are shown in Figure 10. However, the sample test parameters associated with hero, bicycle and weather. In the Figure 10, the example of the test parameters associated with the hero vehicle is provided, which explains the range of test parameter values. For example, the hero “maxSpeed” variable shows [20, 200], which means that the maximum speed of the hero vehicle has to be within 20 to 200 km/hr, and the “RoutingAction” shows [0, 50.60], which depicts the minimum and maximum length of the road in which this scenario takes place.

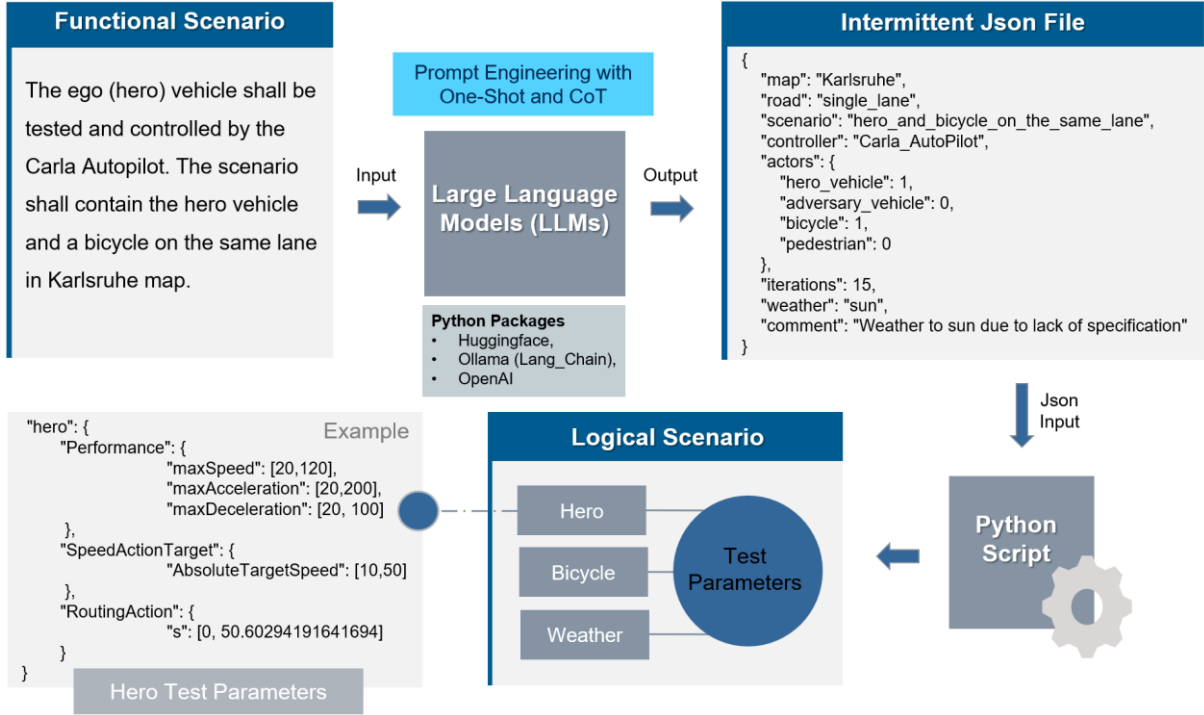


Figure 10 Flow Diagram of the Test Parameter Space Generation in the Intended Test Infrastructure.

It is important to note that the variable names such as “maxSpeed” and “RoutingAction” are named with the exact spelling as they are named in the OpenSCENARIO format, so that in the future, the existing values in the OpenSCENARIO format can be replaced once the logical scenario is converted into concrete scenario as the part of scenario adaptation and manipulation which is explained in the next section.

4.2 Scenario Adaptation and Manipulation in CARLA

The primary objective of this thesis is to induce failure in the system-under-test (SUT) within a scenario in the CARLA simulation environment. To achieve this goal, we focus on modifying layers 4 and 5, which correspond to the dynamic elements such as moving objects and environmental conditions, in the five layers describing scenarios as per the PEGASUS project [96].

The main process responsible for the scenario adaptation and manipulation is the test case generation, which are explained in the next subsection.

4.2.1 Test Case Generation Process

The test case generation process in the Assistant System component is responsible for converting the logical scenario generated as part of the test parameter generation process into a concrete, parameterized representation. The main purpose is to maximize the failure cases of the hero vehicle or ego vehicle controlled by ADAS or CARLA AutoPilot (SUT) by selecting an instance of the logical scenario by specifying a concrete value for the hero, actors, and weather parameters that are likely to cause

the hero vehicle to fail the scenario. To convert the logical scenario, which consists of parameter ranges, into a concrete scenario with specific parameter values, the system employs various sampling methods. In this thesis, three sampling methods namely random techniques, Bayesian optimization-based sampling, and Large Language Model (LLM)-based sampling are explored, as mentioned in the earlier sections.

The random sampling method randomly selects a concrete value from the parameter range for each hero vehicle's, other road actors', and weather's parameters, without considering the results of previous samples as feedback. In contrast, the Bayesian optimization-based sampling method builds a surrogate model from initial and previous points, and then optimizes using acquisition functions (such as expected improvement, or EI) to find the best concrete values that are likely to improve the failure rate of the hero vehicle. Similarly, the LLM-based sampling method uses a LLM to hand-pick the next concrete values that are likely to induce failure in the hero vehicle by learning from the previously sampled concrete values for all the parameters associated with the hero vehicle, other road actors, and weather.

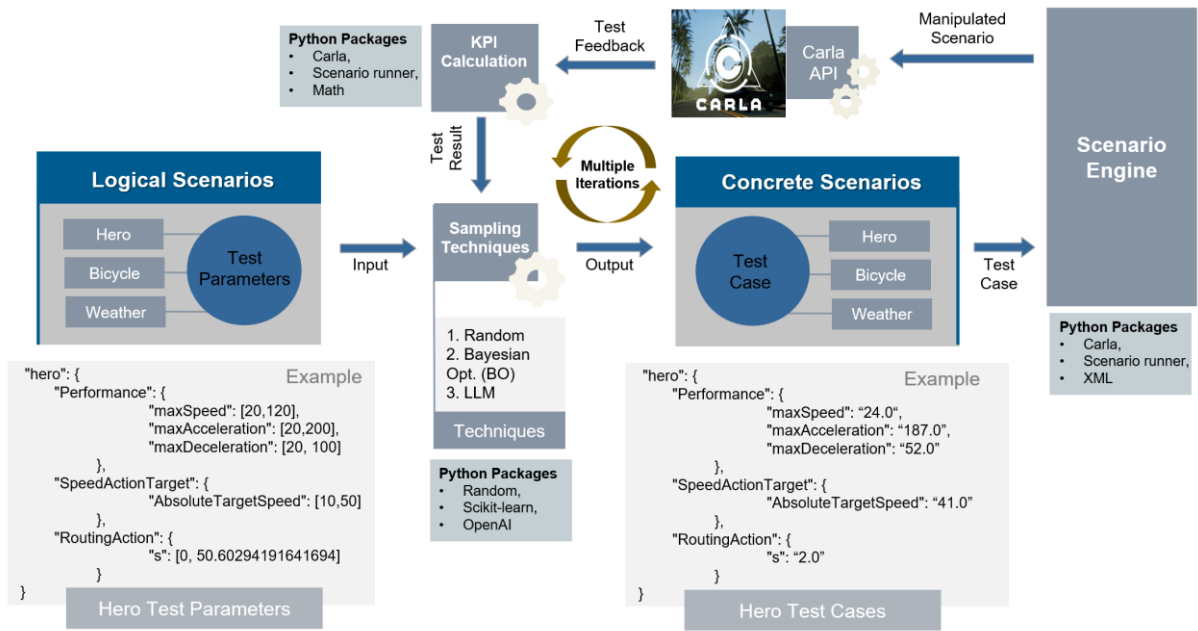


Figure 11 Flow Diagram of the Test Case Generation from the Test Parameters.

The test case generation process is illustrated in Figure 11, which is a continuation of the example shown in the test parameter generation process (Figure 10). The user provides a functional scenario, which is then converted into a logical scenario using a two-step process. The example scenario corresponds to a “hero_and_bicycle_on_the_same_lane” case, where the actors involved are the hero vehicle and a bicycle, the map is Karlsruhe, the road is a single-lane road, and the controller is CARLA AutoPilot or ADAS. Given the logical scenario, which depicts a range of values for each parameter associated with the hero vehicle, bicycle, and weather, the sampling methods are used to convert them into a concrete scenario by sampling a concrete value from the range of values for each parameter. These test

case description or concrete values are then given to the Scenario Engine component for further processes.

In the Scenario Engine component, the concrete parameter values (test case description) are then incorporated into the original scenario ("hero_and_bicycle_on_the_same_lane") file formatted in the OpenSCENARIO 1.0 standard. The simulation of the manipulated scenario file containing the concrete values from the sampling methods is performed in the CARLA simulation environment in the SUT evaluation component using the CARLA interface, which is built on top of the Scenario Runner Toolkit and CARLA APIs, as mentioned earlier. The necessary maps and SUT controller files are also selected according to the functional scenario provided by the user. The CARLA simulation environment outputs evaluation metrics with a "SUCCESS" or "FAILED" status on different tests (e.g., collision with the bicycle). Based on these evaluation metrics, a test score is induced based on 10-class evaluation metrics, more details in Section 6.2.1, in the KPI component, which is sent back to the test case generation process in the Assistant System to use this feedback to select the concrete values from the logical scenarios to maximize the failure rate of the hero vehicle in the next iteration. This process continues for a fixed number of iterations, and a comparison is made among the three sampling techniques to determine the best method for inducing failure in the SUT.

The details of how to evaluate the test parameter generation process, which converts the functional scenario to a logical scenario, and the test case generation process, which uses sampling methods to convert the logical scenario to concrete scenarios using evaluation metrics and various tests, will be discussed in the Section 6.

5 Prototype Implementation

In this section, a brief overview on the implementations of the scenario generation process and the AI methods used in this thesis are discussed.

5.1 Implementation of Scenario Generation in CARLA

As mentioned in the previous section, the scenarios conforming to the OpenSCENARIO 1.0 format can be generated using the pyoscx Python package [35]. However, for the completion of the scenario, the pyoscx package needs the OpenDRIVE format file, which defines the static elements.

The CARLA simulation environment includes 12 built-in maps described in OpenDRIVE format, starting from Town01 to Town12. Notably CARLA reserves Town08 and Town09 as secret for the Leaderboard challenge [63], thus not available for public use. Furthermore, users can create and integrate custom maps using the RoadRunner module [75]. Alongside the pre-existing CARLA maps, the Karlsruhe Institute for Technology (KIT) has developed a custom map depicting a segment of Karlsruhe city in OpenDRIVE format file, which is compatible with the CARLA simulation environment. In this thesis, the Town01 map from the CARLA built-in maps and the Karlsruhe maps from KIT are used.

The Town01 map, a small town featuring a river and several bridges, has road configurations such as single-lane roads, and 3-way intersections (T-way intersections) and Karlsruhe map feature single-lane, multi-lane, 3-way intersections, and 4-way intersections. The precise count of these road configuration types in Town01 and Karlsruhe maps is provided in Table 6.

Table 6: The different road configurations in Town01 & Karlsruhe Maps

Road Configurations	Maps		Total
	Town01	Karlsruhe	
Single Lane	13	46	59
Multi Lane	0	9	9
3-way Intersection	12	43	55
4-way Intersection	0	1	1
Total	25	99	124

In each scenario (OpenSCENARIO) file generated, the scenario involves a hero vehicle or ego vehicle controlled by ADAS (SUT) or CARLA AutoPilot (SUT) and one or more of the other road actors, such as adversary vehicles, other vehicles, pedestrians, or bicycles. The details of the scenario such as weather (e.g., sun, fog, or rain) and the actors' characteristics (e.g., actions, spawning position, control

parameters, start condition, and stop conditions) are also described within the OpenSCENARIO file. This OpenSCENARIO file can be executed directly in CARLA simulation environment using the Scenario Runner Tool as explained earlier. Table 7 illustrates the available scenarios in each road configuration and the associated other road actors.

Table 7: The number of available scenarios in different road configurations and the actors involved.

Road Configurations	Actors Involved in Scenarios			No. of available scenarios
	Adversary Vehicle	Pedestrian	Bicycle	
Single Lane	1	1	3	5
Multi Lane	1	1	3	5
3-way Intersection	2	5	3	10
4-way Intersection	2	5	3	10

For better understanding of the scenario generation process, Figure 12 illustrates a rough idea of one scenario from a 3-way intersection in Town01, involving a hero vehicle (H) and other road actors such as an adversary vehicle (A), a bicycle, and a pedestrian. In this scenario, the pedestrian (P) crosses the road. The bicycle (B) rides slowly. The adversary vehicle (A) crosses the hero vehicle's path. The hero vehicle starts from Road 6 (S) and ends at Road 19 (E). The scenario ends when the hero vehicle reaches the end, avoiding collisions.

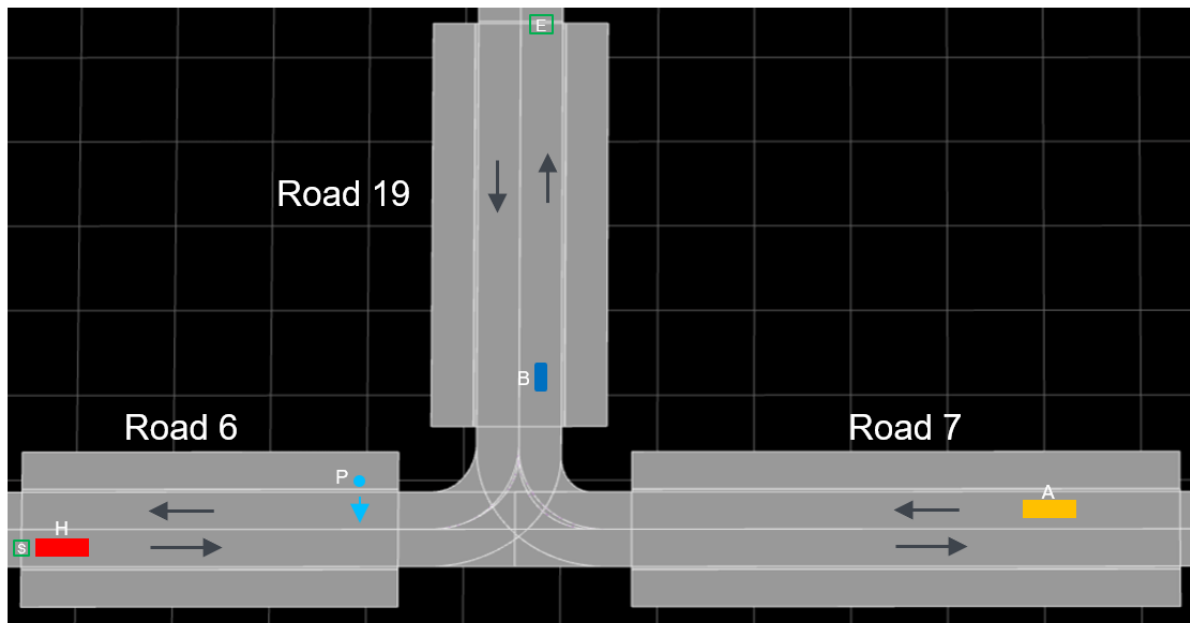


Figure 12 Rough Illustration of a Generated Scenario

It is important to note that, in the Figure 12 illustration, the hero vehicle could potentially start at Road 7 and end at Road 6 or 19, or start at Road 19 and end at Road 7 or 6, which are randomly selected. Regarding the control of the actors, the pedestrian and cyclist controls are explicitly defined in the OpenSCENARIO file, while the adversary actor is controlled by the CARLA “simple_control_vehicle” option, which controls the speed, acceleration, and throttle without any collision or lane invasion awareness. The hero vehicle can be controlled by the CARLA “AutoPilot” option or an ADAS system being developed concurrently.

Within this thesis, a total of 5 scenarios occurring in single-lane and multi-lane road configurations, and 10 scenarios occurring in 3-way and 4-way intersections are generated. As depicted in Table 7, Town01 consists of single-lane roads and 3-way intersections, resulting in 15 corresponding scenarios. Conversely, the Karlsruhe map contains single-lane, multi-lane, 3-way intersections, and 4-way intersections, thus yielding a total of 30 scenarios associated with it. It is important to note that the scenarios can take place randomly on any corresponding road configuration in the Town01 and Karlsruhe maps, leading to diverse scenario environments. For example, a scenario featuring a single-lane road in Town01 map can occur on any of its 13 single-lane roads, chosen randomly.



Figure 13 Example Picture of Generated Scenario Named “Hero_And_Bicycle_On_The_Same_Lane”

Figure 13 illustrates one such scenario, namely “hero_and_bicycle_on_the_same_lane,” generated as part of the thesis taking place in a single-lane road in the Karlsruhe map. In this scenario, the dark blue Ford Mustang is the hero vehicle or ego vehicle controlled by ADAS (SUT) or CARLA AutoPilot (SUT) spawning at the start of one of the single-lane roads in the Karlsruhe map. In front of the hero vehicle, there is a cyclist (a person wearing a white t-shirt) spawning at some

distance gap on the same lane of the road. The action of the bicycle in this scenario is to ride at a slow pace for a moment and suddenly brake for some time before starting to ride again at a slower pace. The hero vehicle will have to act accordingly to avoid a collision with the cyclist throughout and pass other tests such as “WrongLaneTest”, “offRoadtest” etc. The details of the different tests on which the performance of the hero vehicle would be tested can be found in the Section 6. This scenario will come to an end when the hero vehicle completes the route by driving until the end of the road (towards the big tree). In this scenario, the hero vehicle can be controlled by the CARLA “AutoPilot” option or an ADAS system being developed concurrently, whereas the cyclist's controls are explicitly defined in the OpenSCENARIO file. Similar to this scenario, other scenarios have their own set of actors and take place in other road configurations in the Town01 and Karlsruhe maps, as mentioned above. The scenarios in which the hero vehicle is being tested are considered “SUCCESS” if the hero vehicle successfully passes all the different tests associated with the scenario.

5.2 Implementation of AI Technologies

In this thesis, various AI methods ranging from Bayesian optimization to LLMs are used in different processes. In this subsection, we discuss the implementation details of all the AI methods.

We implemented open-source Large Language Models (LLMs), such as Llama-3-8B, Llama-3-70B, Mistral-7B, and WizardLM-13B, using packages like Ollama and Gpt4all. These models are freely available for public use, and developers and researchers can modify or enhance them to suit their specific needs. The Gpt4all Python package facilitates easy implementation of these models, allowing for greater flexibility and control in various applications. In contrast, we implemented or called closed-source models, such as OpenAI GPT-3.5 and GPT-4 Turbo, using the openai Python package. The GPT-4 Turbo is a powerful language model capable of understanding and generating human-like text, making it suitable for a wide range of applications, including automated customer service, content creation, and complex problem-solving. These models are proprietary and provided as a service by OpenAI. Users can access them via API calls, allowing for integration into a wide range of applications. The closed-source nature of these models means that their underlying architecture and training data are not publicly available.

We implemented random techniques using the random package in Python. We implemented the Bayesian optimization technique using the "GaussianProcessRegressor" class from the scikit-learn package in Python. The "GaussianProcessRegressor" class provides a probabilistic model that estimates the function being optimized, guiding the search for the optimal parameters by balancing exploration and exploitation.

6 Evaluation and Results

The evaluation section will discuss in detail how well the processes of test parameter generation and test case generation induce failure in the SUT on a particular scenario. It is important to note that throughout the evaluation and result section, we have experimented only with the CARLA AutoPilot as the SUT, as the variant-rich ADAS system from MT3644 was not ready at the time of the experiments.

6.1 Test Parameter Space Generation Process

As mentioned in the previous sections and illustrated in Figure 10, the test parameter space generation process converts the user-provided short description of requirements (functional scenario) into a range of values of actors and weather parameters (logical scenario). This functional scenario includes details about the scenario, the map in which the SUT needs to be tested, the weather conditions under which the SUT will be tested, and the methods that control the SUT.

To perform the test parameter space generation, the process first converts the functional scenario provided by the user into an intermediate JSON file using Large Language Models (LLMs). Then, a Python script uses this intermediate JSON file to create the logical scenario. For each functional scenario, the LLMs must generate the corresponding intermediate JSON file. To evaluate the performance of the LLMs, it is necessary to assess how well they understand the functional scenario provided by the user and extract the necessary information into the intermediate JSON file. Ultimately, a ground truth intermediate JSON file can be created and used to compare with the LLMs-generated intermediate JSON file. This comparison ensures that the LLMs can properly understand the functional scenario and extract the required information.

6.1.1 Evaluation of Test Parameter Space Generation Process

Before delving into the evaluation of test parameter generation, it is crucial to understand the structure of the intermediate JSON file. This file includes several key elements: The “map” key defines the maps used in the study, which are Town01 and Karlsruhe. The “road” key describes the road configurations in each map. Specifically, the Town01 map has two road configurations: “single-lane” and “3-way intersection.” The Karlsruhe map has four road configurations: “single-lane,” “multi-lane,” “3-way intersection,” and “4-way intersection.” The “scenario” key outlines 15 different scenarios involving various road configurations. The “controller” key defines the available methods to control the SUT, which are “CARLA AutoPilot” and “ADAS” (the latter being developed in a concurrent thesis (MT3644)). The “actors” key specifies the actors involved in the scenario, including the “hero vehicle” (SUT), “adversary vehicle,” “bicycle,” and “pedestrian.” Finally, the “weather” key includes the available weather conditions of “sun,” “fog,” and “rain,” where “sun” denotes dry conditions, which could be day or night.

To evaluate the ability of the LLMs to retrieve the above-mentioned details as an intermediate JSON file from the user-provided functional scenarios, a test parameter space generation benchmark with 100 functional scenarios is manually created as part of this thesis. Each of these 100 functional scenarios has a corresponding intermediate JSON file as a ground truth. Although it is possible to generate more than 100 functional scenarios, we determined that 100 functional scenarios were sufficient for thorough evaluation.

By evaluating the LLMs on the benchmark, we can determine if they can properly understand the functional scenarios and retrieve the information needed for the intermediate JSON file. This evaluation involves providing the 100 functional scenarios, which are part of the benchmark, to the LLMs and comparing the generated intermediate JSON files with the corresponding ground truth JSON files.

These 100 functional scenarios were carefully selected to represent a broad range of conditions relevant to the evaluation of the SUT in CARLA. The selection criteria included diversity in the actors involved and their actions, various road configurations, different operational conditions, and scenarios that reflect real-world usage patterns. Each scenario was manually evaluated to ensure its equal relevance for the performance assessment, thereby maintaining uniform importance across all scenarios. This uniformity ensures that the metric is statistically valid, as all functional scenarios used in this benchmark contribute equally to the final performance measurement, making any scaling or comparison meaningful and accurate.

For a better understanding of this benchmark, let us consider one of the 100 functional scenarios and its corresponding ground truth JSON file, as illustrated in Figure 14. In this example, the functional scenario describes a situation where the hero vehicle (SUT) has to deal with a pedestrian crossing the path of the hero vehicle at a single lane road configuration. The corresponding ground truth intermediate JSON file shows the road configuration as a “single_lane,” the scenario as “hero_and_pedestrian_crossing_in_front,” the map as “Town01,” the actors involved as “one hero vehicle” and “one pedestrian,” and the weather as “Fog”.

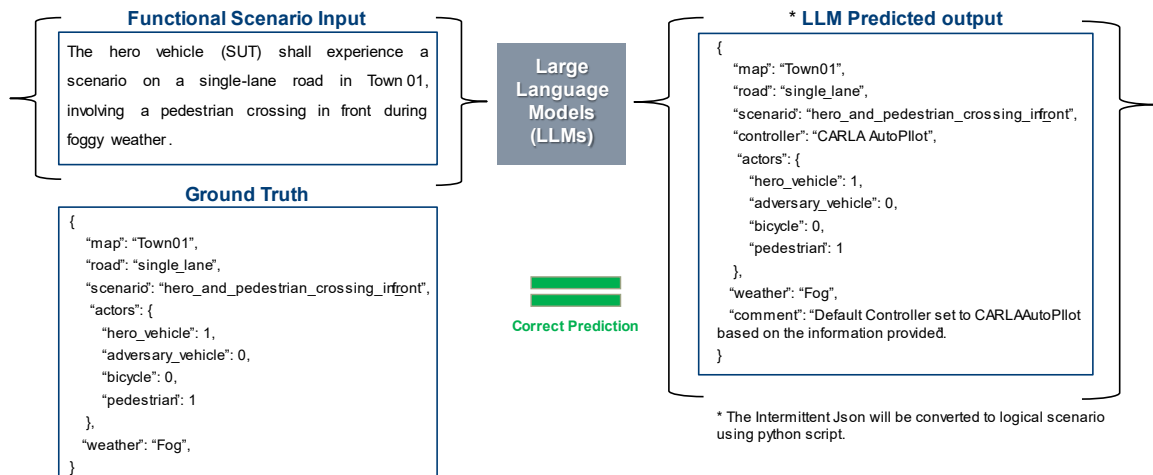


Figure 14 One Sample from the Test Parameter Space Generation Benchmark

This functional scenario is then provided as input to the LLMs, which then output JSON predictions that are compared against the ground truth JSON output for evaluation. The LLMs' JSON prediction is considered "correct" if the predicted JSON file exactly matches the ground truth JSON file, including spelling and details. These scenarios are used to evaluate the LLMs as part of this benchmark, and the results of different LLMs are discussed in Section 6.1.2. It is important to note that the user did not provide the controller details in the functional scenario shown in Figure 14. Therefore, the ground truth intermediate JSON file does not include the controller information. In this case, the LLMs can randomly choose from the available controller methods to control the ADAS (SUT) or CARLA Autopilot (SUT), as the controller details were not specified by the user.

6.1.2 Results and Discussion of the Test Parameter Space Generation Process

In this thesis, as mentioned in the earlier sections, prompt engineering techniques have been used to enhance the performance of the Large Language Models (LLMs). For the evaluation of test parameter generation, 2 closed-sourced LLMs, namely GPT-4 Turbo and GPT-3.5, as well as 4 open-sourced LLMs, namely Llama-3-8B, Llama-3-70B, Mistral-7B, and WizardLM-13B are selected based on the discussion in the previous sections. Notably, the thesis utilizes 4-bit quantization for all the open-source models, where each weight or bias is represented using only 4 bits instead of the typical 32 bits used in single-precision floating-point format (float32). This approach leads to reduced CPU or GPU memory requirements compared to the standard float32 representation.

The use of 4-bit quantization allows the models to be more memory-efficient, which is particularly important for deployment on resource-constrained devices or in scenarios where memory is limited. By reducing the number of bits used to represent the model parameters, the overall model size and memory footprint are significantly decreased, enabling faster inference and potentially improved performance in certain applications.

All prompts to these models consist of four key components, inspired from the paper [38]: "Role and Goal," where the role and goals of the LLMs are mentioned as per this use case; "Context," which provides information about the available maps, road configurations, scenarios, controllers, and weather details; "Instructions," which explain further instructions along with automatic CoT; and "Examples," which include one input-output example for the model, a sample prompt template. It is important to note one-shot learning along with automatic CoT techniques are used to enhance the performance of the models. Experimenting with zero-shot learning techniques in the prompts did not yield the exact JSON format required for the evaluation of these LLMs, hence the decision to use one-shot learning.

Table 8 provides a comprehensive overview of the models used, including the number of parameters, techniques, time taken to generate the intermediate JSON files for all 100 functional scenarios, the number of correct predictions out of those 100 scenarios,

model size and required RAM for the open-sourced models, and tokens/money spent for the closed-sourced models, as well as the hardware specifications for running the open-source models on-premises.

The data in Table 8 shows that OpenAI GPT-4 Turbo significantly outperformed other models, with 94 correct predictions out of 100 functional scenarios. OpenAI GPT-3.5 performed well with 76 correct predictions and the fastest generation time. Among open-source models, Llama-3-70B surpassed GPT-3.5 but not GPT-4 Turbo. The smaller Llama-3-8B outperformed Mistral-7B and WizardLM-13B. GPU usage reduced open-source model run times by approximately 7x. Mistral-7B was faster on GPU but slower than closed-source models, likely due to OpenAI's powerful GPUs. It is important to note that finetuning the smaller open-sourced, on-premises models on this benchmark to enhance their performance is an option, but it was not explored due to time and resource constraints. In the next subsections, two ablation studies related to the evaluation of test parameter space generation using LLMs are discussed.

Table 8: The different LLMs and their results on the test parameter generation benchmark.

LLM	Parameters	Techniques	Time Taken (secs)	Correct Predictions	Cloud / On-Premises	Model Size / Required RAM	Tokens / Money Spent (Dollars)	Hardware Specifications	
								CPU	GPU
OpenAI GPT-4 Turbo	Unknown	One-shot Prompting and 4-bit Quantization	774.88	94 / 100	Cloud	-	86749 / 1.14	-	
OpenAI GPT-3.5	Unknown		239.75	76 / 100	Cloud	-	84377 / 0.06	-	
Llama-3	70B		7856.46	79 / 100	Lambda Labs Cloud	Unknown / 40GB	- / 2.81	-	NVIDIA A100 Tensor (40GB)
Mistral-7B	7B		2019.44	47 / 100	On-Premises	3.83 GB/ 8GB	-	-	NVIDIA GeForce RTX 3080TI (12GB)
WizardLM	13B		3407.09	42 / 100	On-Premises	6.86 GB/ 16GB	-	-	
Llama-3	8B		3415.13	60 / 100	On-Premises	4 GB/ 8GB	-	-	
Mistral-7B	7B		16626.99	45 / 100	On-Premises	3.83 GB/ 8GB	-	AMD Ryzen-9 5900 HS	-
WizardLM	13B		26794.62	43 / 100	On-Premises	6.86 GB/ 16GB	-		-
Llama-3	8B		22517.23	60 / 100	On-Premises	4 GB/ 8GB	-		-

6.1.3 Ablation Study

As part of this thesis, two ablation studies, discussed in the next subsections, are conducted as part of the evaluation of test parameter space generation using the benchmark discussed in the previous section.

6.1.3.1 Human Generated Prompt vs AI Generated Prompt

In this subsection, we examine the performance and cost impact of the OpenAI GPT-4 Turbo model in test parameter space generation benchmark using both human-generated and AI-generated prompts. Initially, a human-designed prompt was optimized to enhance GPT-4 Turbo's performance in generating intermediate JSON files from user-provided functional scenarios in the test parameter generation benchmark. This prompt was then provided to the OpenAI GPT-3.5 model, which reduced the number of words without affecting performance, creating an AI-generated prompt.

Table 9 compares these prompts, showing that the AI-generated prompt slightly outperformed the human-designed one, with 94 correct predictions out of 100. Notably, the cost to run GPT-4 Turbo using the AI-generated prompt was \$1.14, approximately 12% cheaper than using the human-designed prompt. This demonstrates that AI can further optimize prompts, improving performance and reducing costs by eliminating unnecessary words. Additionally, GPT-4 Turbo ran faster with the human-designed prompt, possibly due to varying internet speeds during the experiment.

Table 9: The ablation study results between human-designed prompt and AI-generated prompt

Large Language Model used	Prompts used	Time Taken (secs)	Correct Prediction	Total Tokens	Total Cost
OpenAI GPT-4 Turbo	Human-designed Prompt	454.2	92 out of 100	105289	1.31 dollars
OpenAI GPT-4 Turbo	AI-generated Prompt	774.9	94 out of 100	86749	1.14 dollars

In conclusion, it is evident that after engineering a human-designed prompt for the purpose of test parameter generation, providing the human-designed prompt to a LLM for further reduction of unnecessary words and structuring can potentially improve the performance and result in a cheaper cost.

6.1.3.2 One-Shot learning vs One-shot learning with automatic chain-of-thought CoT techniques

As mentioned in the previous sections, in this thesis, the prompt engineering technique with one-shot learning is used for the process of test parameter generation instead of zero-shot learning. This is because the Large Language Models (LLMs) generate the

intermediate JSON file in a different structure than the ground truth intermediate JSON file of the benchmark.

Recent studies have argued that the automatic chain-of-thought can improve the performance of large language models in complex reasoning tasks by simply adding the sentence “Let's think step by step” to the prompt [94]. In this subsection, we will explore the performance of the OpenAI GPT-4 Turbo when using the one-shot learning prompting technique and one-shot learning with automatic chain-of-thoughts.

Table 10 compares the OpenAI GPT-4 Turbo model's performance on the test parameter generation benchmark using these techniques. Both methods achieved 94 correct predictions out of 100, showing no performance difference. However, one-shot learning without the chain-of-thought technique slightly reduced the total tokens and cost for generating 100 intermediate JSON files, as the phrase "let's think step by step" was omitted. Additionally, there was a minor difference in overall time taken, likely due to internet speed or OpenAI GPU delays during the experiment.

Table 10: The ablation study results between one-shot learning and one-shot learning with CoT prompting techniques

Large Language Model used	Techniques	Time Taken (secs)	Correct Prediction	Total Tokens	Total Cost
OpenAI GPT-4 Turbo	One-Shot Learning	638.64	94 out of 100	83367	1.09 dollars
OpenAI GPT-4 Turbo	One-Shot Learning with CoT	774.9	94 out of 100	86749	1.14 dollars

In conclusion, while chain-of-thought techniques can enhance LLM reasoning capabilities, they did not impact the GPT-4 Turbo model's performance in the test parameter generation benchmark.

6.2 Test Case Generation Process

As mentioned in the previous sections and illustrated in Figure 11, The test case generation process converts the logical scenarios (test parameter space), generated during the test parameter space generation phase, into concrete scenarios (test case descriptions) using three sampling techniques: random sampling, Bayesian optimization-based sampling, and Large Language Model (LLM)-based sampling. The adapted concrete scenarios from these sampling techniques are then incorporated by manipulating the original scenario file formatted in the OpenSCENARIO 1.0 standard. The manipulated scenario file is simulated in the CARLA simulation environment using the CARLA interface, built on top of the Scenario Runner Toolkit and CARLA APIs, along with the necessary maps and SUT controller files. To evaluate the manipulated scenario file, containing the test case description sampled from the sampling

techniques, in terms of its ability to make the hero vehicle (SUT) fail in that scenario, this thesis utilizes some evaluation metrics from the CARLA simulation environment as part of the CARLA Leaderboard [63] as Key Performance Indicators (KPIs), discussed in the next subsection.

During the simulation of the manipulated scenario file, if the hero vehicle violates any of the evaluation metrics mentioned in the next subsection, the CARLA simulation environment returns the scenario as FAILURE and provides the specific violated evaluation metric. If the hero vehicle does not violate any of the evaluation metrics, the CARLA simulation environment returns the scenario as SUCCESS. Based on this result, the sampling techniques generate the next set of concrete scenarios from the logical scenario that are more likely to induce failure in the hero vehicle, resulting in the FAILURE of the scenario. This thesis simulates a certain number of manipulated scenarios from the sampled test cases to check whether the hero vehicle fails in them. The details of this test case generation process are illustrated in Figure 11.

6.2.1 Evaluation of the Test Case Generation Process

In this thesis, some evaluation metrics are chosen from the CARLA Leaderboard, illustrated in Table 11.

Table 11: The Key Performance Indicators (KPIs) used for the evaluation of SUT.

KPIs	Description
Running a red-light evaluation	Checks whether the hero vehicle obeys the red-light signal throughout the scenario run time.
Running a stop sign evaluation	Checks whether the hero vehicle follows the stop sign symbol throughout the scenario run time.
Off-road driving for specific time evaluation	Checks how much time the hero vehicle is driving off-road in the scenario.
Collision evaluation	Checks whether the hero vehicle collides with other road actors or static road elements in the scenario.
Wrong Lane evaluation	Checks whether the hero vehicle is driving on the wrong lane of the road in the scenario.
On Sidewalk evaluation	Checks whether the hero vehicle is driving on the sidewalk of the road in the scenario.
Driven Distance evaluation	Checks whether the distance covered by the hero vehicle is above the given threshold distance.

When the hero vehicle violates any one of the evaluation metrics or KPIs, the scenario is considered a FAILURE. The Overall Failed Test Cases/Scenarios criterion defines how many failed test cases occur out of the total number of sampled test cases, and the Total Execution Time defines how much time in seconds is taken for the execution

of the total number of sampled test cases. For example, if the total number of sampled test cases (number of iterations of the test case generation process) is given as 15, the sampling techniques are allowed to choose 15 concrete scenarios from the logical scenarios based on the test results of the previous iterations to maximize the failure rate. If suppose the hero vehicle violates one of the evaluation metrics for 6 of the sampled test cases, the overall Failed Test Cases would be 6 out of 15. The Total Execution Time is the time taken for the generation and simulation of all 15 sampled test cases.

The evaluation criteria mentioned previously provide details on whether the hero vehicle fails or success in a particular scenario and how many failures of the hero vehicle occur from a certain number of iterations. However, these metrics fail to provide information on the severity of the FAILURE or SUCCESS of the hero vehicle in the scenario. For example, suppose the hero vehicle applies an extremely sudden brake to avoid a collision with a pedestrian, and it avoids the collision. In such a case, the result of the scenario would be a SUCCESS because it did not violate any of the evaluation metrics mentioned above, but the severity of the scenario becoming a FAILURE is not considered in the above evaluation criteria. Thus, to account for the severity of the FAILURE or SUCCESS of the hero vehicle in a scenario, different class metrics are introduced in this thesis. Table 12 describes the details of the different class metrics, the score range, and their descriptions.

Table 12: Details of different manually created class metrics

Class Metrics	Score range in %	Description
Class 1	(0 – 10)	No other vehicles are found, or the file is not running.
Class 2	(11 – 20)	No braking involved, but other road actors are in sight.
Class 3	(21 – 30)	Braking while maintaining a safe distance.
Class 4	(31 – 40)	Braking with an almost collision with other actors or static road elements, or almost violates other metrics
Class 5	(41 – 50)	Extreme braking with an almost collision with one other actor or static road element, or almost violates other metrics.
Class 6	(51 – 60)	Violation one of the metrics shown except collision.
Class 7	(61 – 70)	Unintentional collision with one other actor or static road element.
Class 8	(71 – 80)	Actual collision with one other actor or static road element, or violated two of the other metrics (except collision).
Class 9	(81 – 85)	Collision with more than one of the other actors or static road elements.
Class 10	(86 – 100)	Collision with one of the other actor or static road elements, and violated other metrics too.

For the result of every iteration or sampled test case after the simulation of the scenario in the CARLA simulation environment, the simulated videos and the results are manually checked, and the iteration or sampled test case simulation is manually categorized under the mentioned class metrics. Notably, Class metrics 1-5 are considered SUCCESS, and Class metrics 6-10 are considered FAILURE. To better understand, like in the previous example, if the hero vehicle applies extremely sudden braking and avoids a collision with a pedestrian, this scenario falls under Class 5, denoting that the hero vehicle is close to FAILURE. If, in another scenario, the hero vehicle does not apply any brakes, but other road actors are visible, this scenario is considered Class 2. If, in another scenario, the hero vehicle collides with another road actor twice, this scenario is considered Class 9, and so on. It is important to note that the categorization of the scenario is believed to help the sampling techniques in sampling the next test case better than just mentioning FAILURE (0) or SUCCESS (100).

In the next subsections, a detailed illustration of the results of various techniques in the test case generation process are presented.

6.2.2 Results and Discussion of the Test Case Generation Process

In the previous subsection, detailed evaluation criteria were presented to determine whether the adapted and manipulated scenario, based on the test case description sampled by one of the three sampling techniques used in this thesis, is a SUCCESS or FAILURE. However, to test the sampling techniques' ability to manipulate the original scenario to make the hero vehicle controlled by CARLA AutoPilot (SUT) fail, a total of 10 scenarios are considered in this thesis, each varying in terms of complexity. Out of these 10 scenarios, 6 take place on the Town01 map with different road configurations, weather conditions, and other road actors involved, while 4 take places on the Karlsruhe map with varying road configurations, weather conditions, and other road actors. Although different scenarios are created and available, as mentioned in Section 4.1, only 10 scenarios are selected due to time and resource constraints.

Table 13 shows the details of the 10 test scenarios used and the results of the three sampling techniques in the evaluation criteria, namely overall failed test cases/scenarios and total execution time. It is important to highlight again that throughout the evaluation and test case generation process, the CARLA AutoPilot (SUT) was used as the controller for the hero vehicle. Additionally, the Gaussian process was employed for Bayesian Optimization (BO), and the OpenAI GPT-4 Turbo model was utilized as the LLM due to its superior overall performance, as shown in Table 13, and its performance in the test parameter space generation process, as demonstrated in Table 8. Open-source LLMs such as Llama-3, Mistral-7B, and WizardLM-13B were not experimented with in this process due to time constraints.

Table 13: The results of different sampling techniques on 10 different test scenarios

Scenarios	Random Sampling		BO Sampling (Gaussian Process)		LLM Sampling (OpenAI GPT-4 Turbo)	
	Failed Test Cases	Execution Time (secs)	Failed Test Cases	Execution Time (secs)	Failed Test Cases	Execution Time (secs)
Test 01	1 out of 15	262.22	4 out of 15	423.03	0 out of 15	364.26
Test 02	0 out of 15	515.21	1 out of 15	893.04	1 out of 15	728.7
Test 03	8 out of 15	442.76	11 out of 15	597.29	10 out of 15	629.91
Test 04	1 out of 15	516.3	0 out of 15	912.68	0 out of 15	864.73
Test 05	0 out of 15	431.25	0 out of 15	1000.1	0 out of 15	665.99
Test 06	0 out of 15	270.97	0 out of 15	650.54	0 out of 15	586.5
Test 07	2 out of 15	241.84	5 out of 15	416.11	0 out of 15	602.49
Test 08	3 out of 15	499.93	7 out of 15	617.68	6 out of 15	928.73
Test 09	3 out of 15	227.49	3 out of 15	361.09	2 out of 15	416.72
Test 10	3 out of 15	777.97	5 out of 15	762.92	0 out of 15	1001.2
Total	21	4185.94	36	6634.48	19	6789.23

In terms of the overall failed test cases criteria, as illustrated in Table 13, it is observed that the BO-based sampling technique provided more FAILED scenarios across all the test scenarios when compared to the random technique or the LLM. In this case, the LLM does not perform better than BO, even though the LLM has been proven to perform better than BO in applications such as hyperparameter optimization [10]. The reason for the lower performance of the LLM can be attributed to the novelty of using a LLM as a sampling method in the field of autonomous driving. Notably, in some scenarios, such as scenarios 07 and 10, the LLM performs even poorer than the random sampling, even though the results of the random sampling are provided as a head start to the LLM. The performance of the LLM can be increased by providing comments for each of the sampled test cases across all the test scenarios. However, this would increase the time consumption and money spent on the LLM. Thus, this approach is not performed as part of this thesis, but it remains an untouched approach that could be a possible future work.

In terms of the total execution time, as illustrated in Table 13, the random sampling showcased a lesser time compared to the other two techniques to execute 15 iterations in all 10 test scenarios. The execution time can be completely attributed to the simulation of 15 iterations of the test scenarios, as the random sampling takes less than 5 seconds to sample 15 iterations for each test scenario. Whereas, the execution

time mentioned under the BO and LLM can be attributed to the sampling and simulation of 15 iterations of the test scenarios. The rough time taken for sampling the test cases for 15 iterations by BO and LLM can be the total execution time taken by BO and LLM minus the total execution time taken by the random sampling, assuming the simulation of 15 iterations takes the same time for both techniques. For example, the BO takes 423.03 seconds for the execution of sampling and simulating the 15 iterations of the Test 01 scenario. The test case sampling time taken by BO for the 15 iterations of the Test 01 scenario could be 160.81 seconds ($423.03 - 262.22$), as the execution time by random sampling showcases only the simulation time of the scenarios. Similarly, the test case sampling time taken by LLM for the 15 iterations of the Test 01 scenario could be 102.04, assuming the simulation time for 15 iterations are same for both LLM and random sampling.

In terms of the complexity of the scenarios shown in Table 13, test scenarios 05 and 06 resulted in 0 failed test cases by all three sampling techniques, showcasing the higher complexity of these scenarios. Both scenarios involved the hero vehicle with one other adversary vehicle, and the sampling techniques found it challenging to sample an optimal test case that could result in FAILED scenarios within 15 iterations. From this case, we can conclude that the CARLA AutoPilot showcased better understanding and prevented collisions or any other violations when it came to another vehicle rather than a pedestrian or bicycle. Surprisingly, in test scenario 04, the random sampling technique induced 1 failed case, whereas the BO and LLM could not produce any failed test cases. In contrast, test scenario 03 resulted in more than 7 failed test cases out of 15 by all three sampling techniques, with BO topping with 11 failed test cases, showcasing the least complexity of the scenarios. This scenario involves the hero vehicle with one other bicycle, in which the bicycle crosses in front of the hero vehicle on a multi-lane road and then moves on the same lane in front of the hero vehicle. This scenario shows a high propensity for failure of the hero vehicle controlled by the CARLA AutoPilot due to the unpredictable nature of the bicycle. The remaining test scenarios resulted in some failed test cases, at least by the BO technique, showcasing a decent amount of complexity.

The Table 13 provides insights into the complexity of the scenarios, overall failed test cases/scenarios, and total execution time. However, it does not clearly indicate the severity of the FAILED or SUCCESS scenarios for all 15 iterations of the 10 test scenarios. As mentioned in subsection 6.2.2.1, to understand the severity, we manually categorized all 15 iterations of the 10 test scenarios into 10 different classes by analyzing the simulation video and results. Figure 15 illustrates the results of sampling techniques in all 10 test scenarios in every iteration, categorized into different classes to understand the severity of the scenario results. In this figure, the x-axis represents all three techniques across 10 test scenarios, and the y-axis denotes the number of iterations. The dark green indicates Class 1 (extreme success), while the dark red denotes Class 10 (extreme failure) and blue represents the number of failures out of 15 iterations across all 10 test scenarios using all three sampling techniques.

Figure 15 revealed more information about the results of the test scenarios than Table 13 as it depicts every result of all 15 iterations of 10 test scenarios along with their class categories and some of the important observations about the results are described in the following.

Firstly, it is crucial to note the presence of instances labeled as "ERROR" in the figure. These instances indicate that BO and LLM spawned the hero vehicle and other road actors at precisely the same location. Ideally, such a scenario should induce failure in the hero vehicle due to the overlapping or extremely close proximity of the vehicles. However, the CARLA simulation environment does not allow for multiple vehicles to spawn on top of each other, resulting in the "ERROR" outcome. The "ERROR" category was introduced to prevent BO or LLM from spawning different road actors on top of or in very close proximity to each other, as this would violate the simulation constraints.

Secondly, the table showed that test scenarios 05 and 06 were complex scenarios, as none of the sampling techniques produced failed cases. To determine which scenario, 05 or 06, had the highest complexity, we analyzed Figure 15. The BO sampling technique in test scenario 06 produced categories of Class 4 and Class 5 in more than 7 out of 15 iterations, implying that BO was able to almost make the hero vehicle violate the evaluation metrics or Key Performance Indicators (KPIs). In contrast, in test scenario 05, all three sampling techniques produced more scenarios only in Class 2 and Class 3 categories, indicating that test scenario 05 was more complex than test scenario 06.

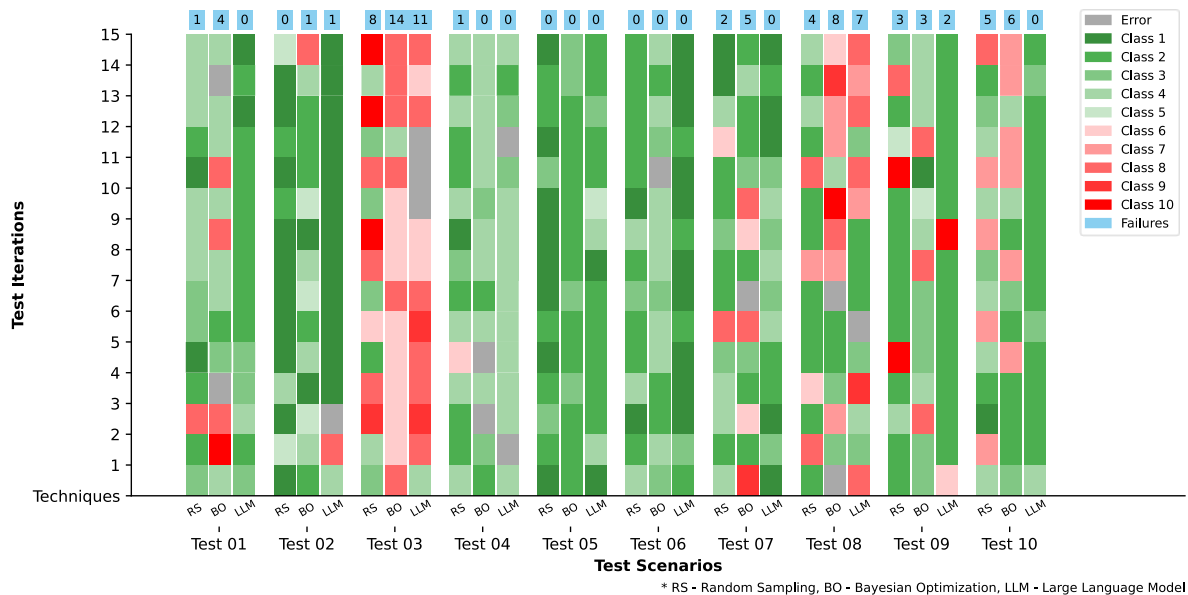


Figure 15 The Results of Different Sampling Techniques, Along with Their Class Metrics, For Each Iteration Across All 10 Test Scenarios

Thirdly, Table 13 showed that the random sampling technique performed better than BO and LLM techniques in test scenario 04, causing 1 failed test case out of 15. However, after analyzing Figure 15, we observed that BO and LLM produced almost 10 scenarios in Class 4 and Class 5 without any failed cases, whereas the random technique produced one failed case/scenario in Class 6, and the remaining scenarios mostly lie in Class 2 and Class 3. This analysis from the figure raises the question of whether the random technique is better than BO and LLM in test scenario 04. While BO and LLM techniques produced more almost collision cases/scenarios than the random technique without any failed cases, the random technique produced one failed case/scenario.

Finally, we observed that some violations of the hero vehicle in some test scenarios were not registered by the CARLA simulation environment. As per the test report generated by CARLA, there were a total of 8 failed cases/scenarios caused by random sampling, 11 failed cases/scenarios caused by BO, and 10 failed cases/scenarios caused by LLMs out of 15 iterations of test scenario 03. However, Figure 15 revealed that there was a total of 14 failed cases/scenarios caused by BO instead of 11, and 11 failed cases caused by LLM instead of 10 in test scenario 03. One of the violations not reported by CARLA using the BO sampling technique in iteration 13 of test scenario 03 is illustrated in Figure 16. From this figure, it is evident that there was a collision by the hero vehicle (red color) onto the bicycle (white color), however, this iteration is denoted as SUCCESS by CARLA with “CollisionTest” as 0, as shown in Figure 17.

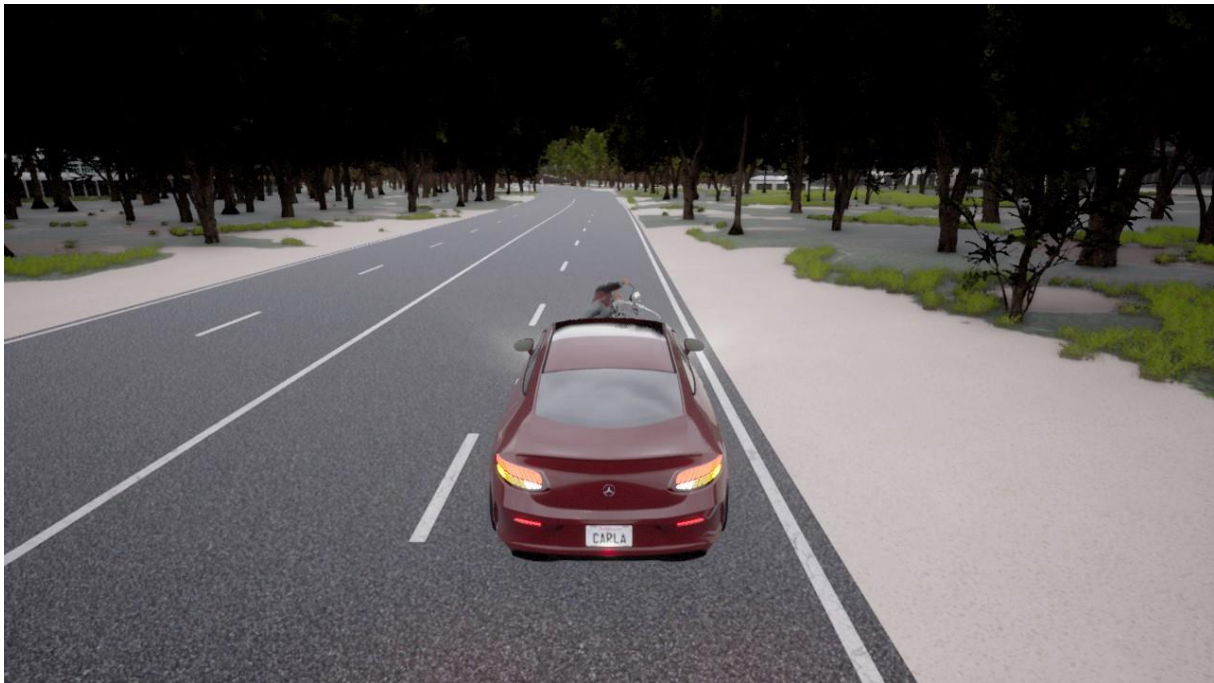


Figure 16 Collision Between the Hero Vehicle (SUT) And the Bicycle.

It is important to note that Figure 15 was generated manually by observing the simulation videos and results. Thus, CARLA did not register some violations caused by the hero vehicle, which could be attributed to sensor failures or other problems with the simulation environment. Similar to test scenario 03, in test scenario 08, the CARLA-

generated test result report (Table 13) mentioned 3, 7, and 6 failed cases/scenarios produced by random, BO, and LLM sampling techniques, respectively. However, the manually observed results (Figure 15) showed 4, 8, and 7 failed cases/scenarios produced by random, BO, and LLM sampling techniques. Similarly, there were violations in some other test scenarios not registered by CARLA, which can be understood by comparing the number of failures from Figure 15 with Table 13. However, Figure 15 also revealed that the BO sampling technique performed better than LLM and random techniques in sampling test cases from the test parameter space in this thesis.

```

===== Results of Scenario: hero_and_bicycle_crossing_in_front ---- SUCCESS =====
> Ego vehicles:
Actor(id=47, type=vehicle.mercedes.coupe_2020);

> Other actors:
Actor(id=48, type=vehicle.gazelle.omafiets);

> Simulation Information

```

Start Time	2024-05-06 11:40:29
End Time	2024-05-06 11:40:57
System Time	27.79s
Game Time	28.31s
Ratio (Game / System)	1.019

```

> Criteria Information

```

Actor	Criterion	Result	Actual Value	Success Value
mercedes.coupe_2020 (id=47)	RunningRedLightTest (Req.)	SUCCESS	0	0
mercedes.coupe_2020 (id=47)	RunningStopTest (Req.)	SUCCESS	0	0
mercedes.coupe_2020 (id=47)	OffRoadTest (Req.)	SUCCESS	0	0
mercedes.coupe_2020 (id=47)	CollisionTest (Req.)	SUCCESS	0	0
mercedes.coupe_2020 (id=47)	WrongLaneTest (Req.)	SUCCESS	0	0
mercedes.coupe_2020 (id=47)	OnSidewalkTest (Req.)	SUCCESS	0	0
mercedes.coupe_2020 (id=47)	CheckDrivenDistance (Req.)	SUCCESS	118.5	118.19667019462806
	Timeout (Req.)	SUCCESS	28.31	100000
	GLOBAL RESULT	SUCCESS		

```

=====

```

Figure 17 Collision Not Registered under “CollisionTest” Despite Hero Vehicle (SUT) Colliding with Bicycle

A final experiment was conducted to evaluate whether the LLM can perform similarly or closer to BO if the number of iterations is increased. To assess this, we conducted an experiment using sampling techniques on the Test 01 scenario with 50 iterations instead of 15 iterations. Additionally, in this experiment, we assessed the effectiveness of the proposed evaluation metrics described in Section 6.2.1, namely manual categorization using class metrics, by experimenting with BO and LLM techniques both with and without class metrics. The experiment consisted of a total of 50 iterations, employing the following sampling techniques: random sampling, BO with class metrics,

BO without class metrics, LLM with class metrics, and LLM without class metrics. The results are illustrated in Figure 18, and two valuable observations were noted.

Firstly, the random sampling technique adapted and manipulated the original scenario randomly without using any test score feedback, resulting in random FAILURE scenarios without any discernible patterns. However, both BO and LLM initially behaved like the random sampling technique until the 13th – 15th iteration, even with the feedback of previous test scores, indicating that they were exploring the test parameter space. Once BO or LLM identified FAILED scenarios, they exploited that parameter range to produce more FAILED scenarios, which can be observed between iterations 14th and 16th in the case of Bayesian optimization with class metrics, showing patterns that denote the importance of test score feedback. With the above observations, we can conclude that BO and LLM initially behave like a random sampling technique and take some time to optimize and induce failure in the hero vehicle, resulting in FAILED scenarios.

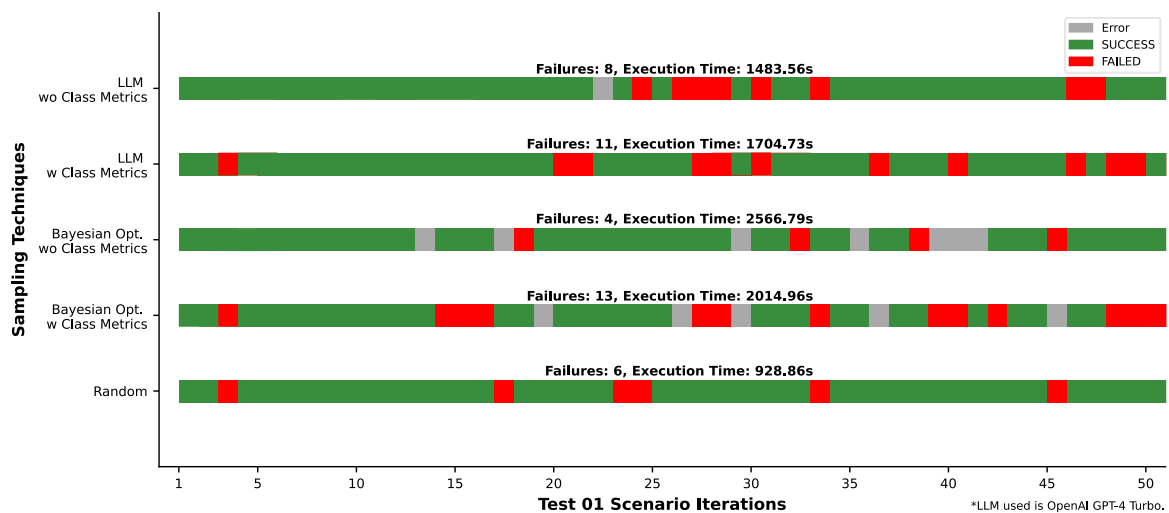


Figure 18 Bar Chart Comparing the Test Scenario 01 Experiment Results of Sampling Techniques

Finally, the BO sampling technique with class metrics outperformed other techniques, resulting in 13 failed cases out of 50 iterations, whereas the BO sampling technique without class metrics exhibited the worst performance among all techniques. Additionally, the LLM sampling technique with class metrics performed almost similarly to the BO with class metrics, with less execution time, showing an improvement in LLM performance with iterations. The execution time of the BO sampling techniques increased with the increase in iterations, as it had to store and compute all the previously sampled values and results to sample new values. These results demonstrate that LLM performance improves with iterations, and sampling techniques work better with class metrics, meaning test score feedback as a number between 0 and 100 rather than a digital number (0 or 100), even though these class metrics were created without exclusive knowledge in risk analysis.

7 Conclusion and Outlook

This thesis has significantly advanced the development of an automated test infrastructure for validating and challenging the safety of autonomous vehicles. By leveraging cutting-edge AI technologies, we established a fully automated pipeline that focuses on scenario adaptation and manipulation to induce failures in the CARLA AutoPilot (SUT) within the CARLA simulation environment. We chose this approach due to limited research in this area and to align with our goal of evaluating the robustness of autonomous systems. We used the CARLA AutoPilot as the controller and chose Town01 and Karlsruhe as the maps, since the variant-rich ADAS system from MT3644 was not ready at the time of the experiment. Future work could include testing the variant-rich ADAS system from MT3644 using the techniques introduced in this thesis. Our initial attempt to use the Safety Pool scenario database through a semi-automatic pipeline (RP3488) faced challenges due to standard mismatches, but we overcame this by creating a fully automated pipeline to generate diverse scenarios using the pyoscx Python package.

Our methodology involves two main steps: generating the test parameter space (converting functional scenarios to logical scenarios) and generating test cases (converting logical scenarios to concrete scenarios). For the former, we utilized various Large Language Models (LLMs), including both open-source and closed-source models on a novel benchmark, with OpenAI GPT-4 Turbo proving to be the most effective. In the test case generation process, we employed sampling techniques such as random sampling, Bayesian Optimization (BO), and LLM (OpenAI GPT-4 Turbo), with BO demonstrating superior efficiency across the ten test scenarios evaluated. Additionally, LLM showed promising results similar to BO when running the experiment with a greater number of iterations. Future work can include improving test feedback metrics, increasing test iterations, and using more powerful LLMs than GPT-4 Turbo, which can possibly increase the performance of the LLM model in this use case.

Despite our overall success, we identified minor discrepancies in the CARLA simulator's ability to accurately record SUT failures, underscoring the need for continued refinement and validation of simulation environments for autonomous vehicle testing. Along with this, we faced many challenges with the CARLA simulation environment, such as installation, GPU demands, supported standards, and bugs. Installation was time-consuming, requiring a powerful GPU like the NVIDIA GeForce RTX 3080TI. Supported standards led to information loss, and bugs caused delays. Discrepancies with real-world scenarios also existed. Some challenges with LLM include the requirement of payment for closed-source LLMs like GPT-4 Turbo and GPT-3.5, and the need to rent powerful GPUs due to resource constraints when running larger open-source models like Llama-3 70B. LLMs sometimes produced extraneous output, but rerunning them with the same input usually resolved the issue.

The introduction highlighted IAS's mission, emphasizing the need for advanced AI techniques and robust testing to handle rapid evolution and variant explosion in

software-defined systems. This thesis contributes by establishing a robust test infrastructure using advanced AI to generate and manipulate scenarios.

In conclusion, this thesis has made substantial contributions to the field of autonomous vehicle testing by developing innovative methods to induce failures and validate SUT performance. The methodology and findings presented herein offer valuable insights and tools for future research and development, ultimately contributing to safer and more reliable autonomous vehicles.

Bibliography

- [1] Bergstra, James, and Yoshua Bengio. "Random search for hyper-parameter optimization." *Journal of machine learning research* 13, no. 2 (2012).
- [2] Hutter, Frank, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration." In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, pp. 507-523. Springer Berlin Heidelberg, 2011.
- [3] Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams. "Practical bayesian optimization of machine learning algorithms." *Advances in neural information processing systems* 25 (2012).
- [4] Ginsbourger, David, Rodolphe Le Riche, and Laurent Carraro. "Kriging is well-suited to parallelize optimization." In *Computational intelligence in expensive optimization problems*, pp. 131-162. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
- [5] Williams, Christopher KI, and Carl Edward Rasmussen. *Gaussian processes for machine learning*. Vol. 2, no. 3. Cambridge, MA: MIT press, 2006.
- [6] Li, Lisha, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. "Hyperband: A novel bandit-based approach to hyperparameter optimization." *Journal of Machine Learning Research* 18, no. 185 (2018): 1-52.
- [7] Falkner, Stefan, Aaron Klein, and Frank Hutter. "BOHB: Robust and efficient hyperparameter optimization at scale." In *International conference on machine learning*, pp. 1437-1446. PMLR, 2018.
- [8] Ramakrishna, Shreyas, Baiting Luo, Yogesh Barve, Gabor Karsai, and Abhishek Dubey. "Risk-aware scene sampling for dynamic assurance of autonomous systems." In *2022 IEEE International Conference on Assured Autonomy (ICAA)*, pp. 107-116. IEEE, 2022.
- [9] Jomaa, Hadi S., Josif Grabocka, and Lars Schmidt-Thieme. "Hyp-rl: Hyperparameter optimization by reinforcement learning." *arXiv preprint arXiv:1906.11527* (2019).
- [10] Zhang, Michael R., Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. "Using Large Language Models for Hyperparameter Optimization." *arXiv preprint arXiv:2312.04528* (2023).
- [11] Zheng, Mingkai, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. "Can GPT-4 Perform Neural Architecture Search?." *arXiv preprint arXiv:2304.10970* (2023).
- [12] Chen, Angelica, David M. Dohan, and David R. So. "EvoPrompting: Language Models for Code-Level Neural Architecture Search." *arXiv preprint arXiv:2302.14838* (2023).
- [13] Zhang, Shujian, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. "AutoML-GPT: Automatic Machine Learning with GPT." *arXiv preprint arXiv:2305.02499* (2023).
- [14] Senthil, Jane Huffman Hayes Alex Dekhtyar, and Karthikeyan Sundaram. "Advancing Candidate Link Generation for Requirements Tracing: the Study of Methods."

- [15] Dekhtyar, Alex, Jane Huffman Hayes, Senthil Sundaram, Ashlee Holbrook, and Olga Dekhtyar. "Technique integration for requirements assessment." In 15th IEEE International Requirements Engineering Conference (RE 2007), pp. 141-150. IEEE, 2007.
- [16] Asuncion, Hazeline U., Arthur U. Asuncion, and Richard N. Taylor. "Software traceability with topic modeling." In Proceedings of the 32nd ACM/IEEE international conference on Software Engineering-Volume 1, pp. 95-104. 2010.
- [17] De Lucia, Andrea, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. "Enhancing an artefact management system with traceability recovery features." In 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., pp. 306-315. IEEE, 2004.
- [18] Guo, Jin, Jinghui Cheng, and Jane Cleland-Huang. "Semantically enhanced software traceability using deep learning techniques." In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 3-14. IEEE, 2017.
- [19] Lin, Jinfeng, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. "Traceability transformed: Generating more accurate links with pre-trained bert models." In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 324-335. IEEE, 2021.
- [20] Lin, Jinfeng, Amrit Poudel, Wenhao Yu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. "Enhancing automated software traceability by transfer learning from open-world data." arXiv preprint arXiv:2207.01084 (2022).
- [21] Rodriguez, Alberto D., Katherine R. Dearstyne, and Jane Cleland-Huang. "Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability." In 2023 IEEE 31st International Requirements Engineering Conference Workshops (REW), pp. 455-464. IEEE, 2023.
- [22] Safety Pool Scenario Database. Accessed February 9, 2024.
- [23] Fremont, Daniel J., Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. "Scenic: a language for scenario specification and scene generation." In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 63-78. 2019.
- [24] Majumdar, Rupak, Aman Mathur, Marcus Pirron, Laura Stegner, and Damien Zufferey. "Paracosm: A test framework for autonomous driving simulations." In International Conference on Fundamental Approaches to Software Engineering, pp. 172-195. Cham: Springer International Publishing, 2021.
- [25] Wang, Sen, Zhuheng Sheng, Jingwei Xu, Taolue Chen, Junjun Zhu, Shuhui Zhang, Yuan Yao, and Xiaoxing Ma. "ADEPT: A Testing Platform for Simulated Autonomous Driving." In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-4. 2022.
- [26] Hou, Xinyi, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. "Large language models for software engineering: A systematic literature review." arXiv preprint arXiv:2308.10620 (2023).
- [27] Fan, Angela, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. "Large language models for software

- engineering: Survey and open problems." arXiv preprint arXiv:2310.03533 (2023).
- [28] Luo, Xianchang, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. "PRCBERT: Prompt Learning for Requirement Classification using BERT-based Pretrained Language Models." In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-13. 2022.
 - [29] Luitel, Dipeeka, Shabnam Hassani, and Mehrdad Sabetzadeh. "Improving requirements completeness: Automated assistance through large language models." arXiv preprint arXiv:2308.03784 (2023).
 - [30] Zhang, Jianzhang, Yiyang Chen, Nan Niu, and Chuang Liu. "A Preliminary Evaluation of ChatGPT in Requirements Information Retrieval." arXiv preprint arXiv:2304.12562 (2023).
 - [31] Moharil, Ambarish, and Arpit Sharma. "Tabasco: A transformer-based contextualization toolkit." Science of Computer Programming 230 (2023): 102994.
 - [32] Wang, Yawen, Lin Shi, Mingyang Li, Qing Wang, and Yun Yang. "A deep context-wise method for coreference detection in natural language requirements." In 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 180-191. IEEE, 2020.
 - [33] Arora, Chetan, John Grundy, and Mohamed Abdelrazek. "Advancing Requirements Engineering through Generative AI: Assessing the Role of LLMs." arXiv preprint arXiv:2310.13976 (2023).
 - [34] Arvidsson, Simon, and Johan Axell. "Prompt engineering guidelines for LLMs in Requirements Engineering." (2023).
 - [35] Andersson, Mikael, Irene Natale, Andreas Tingberg, and Jakob Kath. "Pyoscx/Scenariogeneration: Python Library to Generate Linked OpenDRIVE and OpenSCENARIO Files." GitHub. Accessed April 14, 2024.
 - [36] Shao, Hao, Letian Wang, Ruobing Chen, Steven L. Waslander, Hongsheng Li, and Yu Liu. "ReasonNet: End-to-End Driving with Temporal and Global Reasoning." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 13723-13733. 2023
 - [37] Shao, Hao, Letian Wang, Ruobing Chen, Hongsheng Li, and Yu Liu. "Safety-enhanced autonomous driving using interpretable sensor fusion transformer." In Conference on Robot Learning, pp. 726-737. PMLR, 2023.
 - [38] Xia, Yuchen, Manthan Shenoy, Nasser Jazdi, and Michael Weyrich. "Towards autonomous system: flexible modular production system enhanced with large language model agents." arXiv preprint arXiv:2304.14721 (2023).
 - [39] Xu, Zhenhua, Yujia Zhang, Enze Xie, Zhen Zhao, Yong Guo, Kenneth KY Wong, Zhenguo Li, and Hengshuang Zhao. "Drivegpt4: Interpretable end-to-end autonomous driving via large language model." arXiv preprint arXiv:2310.01412 (2023).
 - [40] Fu, Daocheng, Xin Li, Licheng Wen, Min Dou, Pinlong Cai, Botian Shi, and Yu Qiao. "Drive like a human: Rethinking autonomous driving with large language models." In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, pp. 910-919. 2024.

- [41] Achiam, Josh, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida et al. "Gpt-4 technical report." arXiv preprint arXiv:2303.08774 (2023).
- [42] GitHub. Accessed February 14, 2024. <https://github.com/features/copilot>.
- [43] Dibia, Victor, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. "Aligning Offline Metrics and Human Judgments of Value for Code Generation Models." In Findings of the Association for Computational Linguistics: ACL 2023, pp. 8516-8528. 2023.
- [44] Judini. 2023. The future of software development powered by AI.
- [45] Mastropaolo, Antonio, Luca Pascarella, and Gabriele Bavota. "Using deep learning to generate complete log statements." In Proceedings of the 44th International Conference on Software Engineering, pp. 2279-2290. 2022
- [46] Chen, Fuxiang, Fatemeh H. Fard, David Lo, and Timofey Bryksin. "On the transferability of pre-trained language models for low-resource programming languages." In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 401-412. 2022.
- [47] Ahmed, Toufique, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. "Improving Few-Shot Prompts with Relevant Static Analysis Products." arXiv preprint arXiv:2304.06815 (2023).
- [48] Kang, Sungmin, Juyeon Yoon, and Shin Yoo. "Large language models are few-shot testers: Exploring llm-based general bug reproduction." In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2312-2323. IEEE, 2023.
- [49] Banks, Jeanine. "Gemma: Introducing New State-of-the-Art Open Models." Google, February 21, 2024.
- [50] Wang, Simin, Liguang Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. "Machine/deep learning for software engineering: A systematic literature review." IEEE Transactions on Software Engineering 49, no. 3 (2022): 1188-1231.
- [51] Yang, Yanming, Xin Xia, David Lo, and John Grundy. "A survey on deep learning for software engineering." ACM Computing Surveys (CSUR) 54, no. 10s (2022): 1-73.
- [52] Li, Mingyang, Ye Yang, Lin Shi, Qing Wang, Jun Hu, Xinhua Peng, Weimin Liao, and Guizhen Pi. "Automated extraction of requirement entities by leveraging LSTM-CRF and transfer learning." In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 208-219. IEEE, 2020.
- [53] Winkler, Jonas Paul, Jannis Grönberg, and Andreas Vogelsang. "Predicting how to test requirements: An automated approach." In 2019 IEEE 27th International Requirements Engineering Conference (RE), pp. 120-130. IEEE, 2019.
- [54] Rath, Michael, Jacob Rendall, Jin LC Guo, Jane Cleland-Huang, and Patrick Mäder. "Traceability in the wild: automatically augmenting incomplete trace links." In Proceedings of the 40th International Conference on Software Engineering, pp. 834-845. 2018.

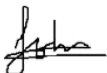
- [55] Wang, Yan, Xiaojiang Liu, and Shuming Shi. "Deep neural solver for math word problems." In Proceedings of the 2017 conference on empirical methods in natural language processing, pp. 845-854. 2017.
- [56] Yang, Kaiyu, and Jia Deng. "Learning to prove theorems via interacting with proof assistants." In International Conference on Machine Learning, pp. 6984-6994. PMLR, 2019.
- [57] Satpute, Ankit, Noah Giessing, Andre Greiner-Petter, Moritz Schubotz, Olaf Teschke, Akiko Aizawa, and Bela Gipp. "Can LLMs Master Math? Investigating Large Language Models on Math Stack Exchange." arXiv preprint arXiv:2404.00344 (2024).
- [58] Imani, Shima, Liang Du, and Harsh Shrivastava. "Mathprompter: Mathematical reasoning using large language models." arXiv preprint arXiv:2303.05398 (2023).
- [59] Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. "Chain-of-thought prompting elicits reasoning in large language models." Advances in neural information processing systems 35 (2022): 24824-24837.
- [60] Liu, Hui, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. "Deep learning based program generation from requirements text: Are we there yet?." IEEE Transactions on Software Engineering 48, no. 4 (2020): 1268-1289.
- [61] Fairbairn, Jon, and Stuart C. Wray. "Code generation techniques for functional languages." In Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 94-104. 1986.
- [62] Allamanis, Miltiadis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness." ACM Computing Surveys (CSUR) 51, no. 4 (2018): 1-37
- [63] "Leaderboard." CARLA Autonomous Driving Leaderboard. Accessed April 11, 2024.
- [64] Khayyam, Hamid, Bahman Javadi, Mahdi Jalili, and Reza N. Jazar. "Artificial intelligence and internet of things for autonomous vehicles." Nonlinear approaches in engineering applications: Automotive Applications of engineering problems (2020): 39-68.
- [65] Abbasi, Shirin, and Amir Masoud Rahmani. "Artificial intelligence and software modeling approaches in autonomous vehicles for safety management: A systematic review." Information 14, no. 10 (2023): 555.
- [66] Ravishankaran, Charan. "Impact on how AI in automobile industry has affected the type approval process at RDW." Master's thesis, University of Twente, 2021.
- [67] Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: An open urban driving simulator." In Conference on robot learning, pp. 1-16. PMLR, 2017.
- [68] Malik, Sumbal, Manzoor Ahmed Khan, Aadam, Hesham El-Sayed, Farkhund Iqbal, Jalal Khan, and Obaid Ullah. "CARLA+: An Evolution of the CARLA Simulator for Complex Environment Using a Probabilistic Graphical Model." Drones 7, no. 2 (2023): 111.

- [69] Bergamini, Luca, Yawei Ye, Oliver Scheel, Long Chen, Chih Hu, Luca Del Pero, Błażej Osiński, Hugo Grimmett, and Peter Ondruska. "Simnet: Learning reactive self-driving simulations from real-world observations." In 2021 IEEE International Conference on Robotics and Automation (ICRA), pp. 5119-5125. IEEE, 2021.
- [70] Zhang, Xizhe, Siddhartha Khastgir, and Paul Jennings. "Scenario description language for automated driving systems: A two-level abstraction approach." In 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 973-980. IEEE, 2020.
- [71] "ASAM OpenSCENARIO." Home. Accessed April 14, 2024.
- [72] Carla-Simulator. "Carla-Simulator/Scenario_runner: Traffic Scenario Definition and Execution Engine." GitHub. Accessed April 14, 2024.
- [73] Chiang, Wei-Lin, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang et al. "Chatbot arena: An open platform for evaluating llms by human preference." arXiv preprint arXiv:2403.04132 (2024).
- [74] "ASAM OpenDRIVE." Home. Accessed April 14, 2024.
- [75] "Roadrunner." MATLAB. Accessed April 14, 2024.
- [76] Mukherjee, Subhabrata, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. "Orca: Progressive learning from complex explanation traces of gpt-4." arXiv preprint arXiv:2306.02707 (2023).
- [77] Parnami, Archit, and Minwoo Lee. "Learning from few examples: A summary of approaches to few-shot learning." arXiv preprint arXiv:2203.04291 (2022).
- [78] Kojima, Takeshi, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. "Large language models are zero-shot reasoners." *Advances in neural information processing systems* 35 (2022): 22199-22213.
- [79] Xu, Can, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. "Wizardlm: Empowering large language models to follow complex instructions." arXiv preprint arXiv:2304.12244 (2023).
- [80] "Falcon LLM." LLM. Accessed February 20, 2024.
- [81] OpenAI platform. Accessed February 20, 2024.
- [82] "Gemini." Google DeepMind. Accessed February 20, 2024.
- [83] Team, Gemini, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut et al. "Gemini: a family of highly capable multimodal models." arXiv preprint arXiv:2312.11805 (2023).
- [84] Touvron, Hugo, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov et al. "Llama 2: Open foundation and fine-tuned chat models." arXiv preprint arXiv:2307.09288 (2023).
- [85] Touvron, Hugo, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière et al. "Llama: Open and efficient foundation language models." arXiv preprint arXiv:2302.13971 (2023).
- [86] Jiang, Albert Q., Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand et al. "Mistral 7B." arXiv preprint arXiv:2310.06825 (2023).
- [87] AI, Mistral. "Mixtral of Experts." Mistral AI | Open-weight models, Jan 31, 2024.

- [88] Tunstall, Lewis, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang et al. "Zephyr: Direct distillation of lm alignment." arXiv preprint arXiv:2310.16944 (2023).
- [89] Rafailov, Rafael, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. "Direct preference optimization: Your language model is secretly a reward model." *Advances in Neural Information Processing Systems* 36 (2024).
- [90] "Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date." AI at Meta. Accessed April 21, 2024.
- [91] White, Colin, Willie Neiswanger, and Yash Savani. "Bananas: Bayesian optimization with neural architectures for neural architecture search." In *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 12, pp. 10293-10301. 2021
- [92] Wu, Jia, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. "Hyperparameter optimization for machine learning models based on Bayesian optimization." *Journal of Electronic Science and Technology* 17, no. 1 (2019): 26-40.
- [93] EleutherAI. "ELEUTHERAI/LM-Evaluation-Harness: A Framework for Few-Shot Evaluation of Language Models." GitHub. Accessed February 20, 2024.
- [94] Prystawski, Ben, Michael Li, and Noah Goodman. "Why think step by step? Reasoning emerges from the locality of experience." *Advances in Neural Information Processing Systems* 36 (2024).
- [95] Menzel, Till, Gerrit Bagschik, Leon Isensee, Andre Schomburg, and Markus Maurer. "From functional to logical scenarios: Detailing a keyword-based scenario description for execution in a simulation environment." In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pp. 2383-2390. IEEE, 2019.
- [96] "Pegasus Research Project." pegasus. Accessed May 17, 2024.
- [97] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
- [98] Ebert, C., Weyrich, M., and Vietz, H., "AI-Based Testing for Autonomous Vehicles," SAE Technical Paper 2023-01-1228, 2023, doi:10.4271/2023-01-1228
- [99] Halder, Subir, Amrita Ghosal, and Mauro Conti. "Secure over-the-air software updates in connected vehicles: A survey." *Computer Networks* 178 (2020): 107343.
- [100] Zampetti, Fiorella, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study." In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 471-482. IEEE, 2021.

Declaration of Compliance

I hereby declare to have written this work independently and to have respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as images, drawings, text passages) are used in this work, I declare that these materials are referenced accordingly (e.g. quote, source) and, whenever necessary, consent from the author to use such materials in my work has been obtained.

Signature: 

Stuttgart, on the 18.06.2024