CHALMERS
UNIVERSITY OF TECHNOLOGY

Multi-Agent Large Language Model as AD/ADAS System Engineer

Master's Thesis in Computer science and engineering

Ali Alkhaled, Ali Malla

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's Thesis 2025

Multi-Agent Large Language Model as AD/ADAS System Engineer

Ali Alkhaled, Ali Malla

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2025

Multi-Agent Large Language Model as AD/ADAS System Engineer
Ali Alkhaled, Ali Malla

Academic Supervisor: Christian Berger, Tayssir Bouraffa, Department of Computer Science and Engineering
Industrial Supervisor: Ali Nouri, Zhennan Fei, Volvo Cars Corporation.
Examiner: Daniel Strüber, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Ali Alkhaled, Ali Malla
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Recent advancements in generative AI, particularly in Large Language Models (LLMs) have sparked a major revolution and a qualitative shift in various fields, including software code generation and unit test generation, offering new opportunities to automate various aspects of the software development process. At the same time, the demand of sophisticated software in the automotive industry has grown rapidly. This trend motivates the exploration of the potential of LLMs in supporting the development of AD/ADAS functions.

A pipeline, *CoTeGen*, for code generation, test case generation, and the automation of virtual simulation-based testing in Esmini is designed following three iterative development cycles. The pipeline is designed to address four AD/ADAS functions. The first two are constrained to relatively elementary maneuvers, namely simple braking and lane changing, whereas the latter are dedicated to more sophisticated control tasks, specifically Adaptive Cruise Control and Collision Avoidance.

Across these iterative cycles, the pipeline progressed from generating non-compilable software components to providing compilable and functional software. Based on a multi-run experimental evaluation involving five open-source LLMs, Codellama:7B, Mistral:7B, DeepSeek-Coder-v2:7B, Gemma3:4B, and Qwen2.5-Coder:7B, the pipeline shows a clear ability to generate correct source code for the simpler functions, while proving far less effective for the more advanced functions. Finally, we discuss the challenges and limitations of applying LLMs to code and unit test generation within the proposed pipeline.

Keywords: Multi-agent, code generation, unit test generation, large language model, AD/ADAS, Esmini

Acknowledgements

We would like to express our sincere gratitude to our partner, Volvo Cars, with special thanks to Ali Nouri for his supervision and support, and to Zhennan Fei for his valuable technical support.

We are also very grateful to our academic supervisors, Christian Berger and Tayssir Bouraffa for their support throughout this journey. Our appreciation extends to our examiner, Daniel Strüber, whose feedback during the half-time report and presentation was highly valuable.

Finally, we would like to thank our families and friends for their encouragement and support during this journey.

Ali Alkhaled, Ali Malla, Gothenburg, September 2025

Contents

List of Figures

List of Tables

1

Introduction

Autonomous driving systems (ADS) and advanced driver assistance systems (ADAS) require increasingly sophisticated software to meet safety and performance demands. However, developing and maintaining software for such systems is a highly complex process, which presents a significant challenge in the automotive industry [2]. In recent years, ADAS have become an essential part of modern transportation, enhancing vehicle safety and driving efficiency. Similarly, ADS are expected to become a fundamental component of modern transportation in the coming years, further increasing the need for efficient software development.

At the same time, technological advancements have accelerated, particularly in artificial intelligence (AI). Recent breakthroughs in large language models (LLMs) and AI agents, have brought about a major revolution and a qualitative shift in various fields, including software code generation, software testing and validation, offering new possibilities for automating and optimizing complex development processes [2].

To respond faster to new insights, maintain competitiveness, and keep pace with the rapid evolution of technology, while also increasing road safety and reducing the number of fatalities caused by car accidents, original equipment manufacturers (OEMs) are increasingly compelled to accelerate software development cycles. Achieving this acceleration, however, presents significant challenges.

A key challenge in this process lies in the task of software development itself, particularly ensuring that the implemented code is both functionally correct and safe for deployment in real-world automotive environments. Consequently, this stage must be followed by comprehensive testing and validation procedures to ensure software correctness and safety as well as overall system reliability. These activities are inherently complex and time-consuming. Automating parts of this process, such as code generation, testing, and preliminary validation, and providing the results to human code reviewers for final assessment can help accelerate development while maintaining safety and quality standards.

This thesis explores the leveraging of multi-agent LLMs and their collaborative capabilities to automate and enhance the software development process of ADS/ADAS, while rigorously ensuring correctness. The proposed approach involves designing a fully automated, iterative collaborative pipeline in which distinct AI agents are

assigned specific tasks, such as code generation and unit test suite generation, without any human intervention. To rigorously validate the correctness of the generated code, simulation-based testing is also performed using the Esmini simulation environment, which is a traffic simulation open-source tool. It provides a realistic and controlled virtual setting to assess software behavior.

1.1 Problem Statement

Developing new features or fixing bugs in a Software-Intensive System of Systems (SISoS), particularly in safety-critical domains like ADS and ADAS, is an inherently complex process. The safety-critical nature of these systems also poses a significant challenge in integrating the engineering of these safety-critical systems with the fast-paced, iterative DevOps process [3]. However, the implementation of a new software component for such systems requires considerable resources and must undergo multiple rigorous stages, including testing and validation & verification, to ensure that the code is functionally correct, safe for deployment in real-world automotive environments, and compliant with industry standards [4]. Moreover, unlike other software domains, there is no room for compromise or trade-offs due to the critical safety requirements of these systems. Consequently, this process often results in delays in system deployment.

Current development approaches in the automotive industry, particularly for code generation, often rely on model-driven development and formal methods. For example, tools such as SCADA [5, 6] have been used to develop such systems, producing qualified and standards-compliant software. However, these tools come with notable drawbacks, including a steep learning curve, low user-friendliness, and high cost. In the area of unit test generation, methods such as search-based testing and feedback-directed random testing have been explored. Despite their potential, these approaches suffer from two main limitations: first, the generated test cases are often less readable and harder to interpret than those written manually; second, they typically lack meaningful assertions or include only generic ones [7].

Fortunately, in simulation based testing context, simulation technologies have proven to be invaluable, as they complement physical testing by enabling functional verification of software within a virtual environment. This approach not only substantially reduces logistical challenges and associated costs but also opens up opportunities to automate the entire development process.

These limitations highlight the need for a new solution that enhances development efficiency while addressing the shortcomings of existing tools. LLMs offer a promising alternative due to their strong performance in code generation and their ability to understand and generate human-readable content through natural language processing. As highlighted in [4, 2], LLM-based multi-agent systems can significantly improve efficiency in software engineering tasks such as code generation and testing & quality assurance, while also addressing complex challenges inherent in the field. Hence, it is worthwhile to explore how multi-agent LLMs can be leveraged

to automate specific tasks within the development process of ADS/ADAS, driving significant efficiency improvements.

Recently, numerous studies have explored the use of LLMs for code generation and unit test generation [7, 8, 6]. However, the majority of these efforts have focused on general-purpose and basic functions rather than domain-specific functions, such as those found in automotive systems, which tend to be more complex.

Despite this growing interest, only a few LLM-based researches have attempted to address the challenges specific to automotive software development, such as [9, 10]. Even among those, results have been limited, often due to suboptimal pipeline designs or a lack of effective integration between components responsible for different engineering tasks.

The proposed solution involves employing an iterative and fully automated pipeline of multi agents, where each agent is specialized in a specific engineering task and collaborates with others, combining the LLM code generation with an automatic assessment within a Software-in-the-Loop (SIL) virtual test environment. This virtual environment consists of two parts: one for unit-level testing to verify the correctness of the code at a low level, and another for simulation-based testing to validate software behavior in realistic scenarios. The results from these tests are used for the automated evaluation of the generated code, enabling continuous feedback and improvement. Additionally, the generated code, test cases and corresponding test results are provided to the user for review, allowing human experts to validate and approve the outputs before deployment. This approach can have the potential to improve the efficiency of the development process for ADS/ADAS, by automating tasks typically handled by human teams, such as coding, testing, and validation.

1.2 Purpose of The Study

The purpose of this study is to design and implement an AI based tool that can help with automating the development process of the AD/ADAS functions. The main idea of this research is to explore how specialized LLM agents can work together within a controlled pipeline to generate functioning software components for ADS/ADAS. This study aims to the automotive industry, as well as researchers in software engineering, by exploring the integration of AI-driven automation in complex systems. This study will offer insights into the practical use and performance of multi-agent LLM pipelines, contributing to more efficient and automated software development practices in the field of ADS/ADAS technology.

1.3 Research Questions

RQ1: How can a multi-agent LLMs pipeline be designed and prototyped to deliver software components that meet the specific requirements of ADS/ADAS?

RQ2: What are the limitations of using LLMs for automating software develop-

ment tasks, such as code generation and unit test generation, in the context of ADS/ADAS?

RQ3: How effective is the proposed multi-agent LLM pipeline in automatically generating AD/ADAS software functions and validating their correctness?

1.4 Significance of The Study

This study conducts a proof of concept (PoC) study to explore the potential of leveraging multi-agent LLMs as ADS/ADAS engineering assistants to improve the efficiency of complex software development processes. Specifically, it focuses on automating tasks that do not require human creativity, such as code generation, testing, and validation. In essence, this study refines and extends existing knowledge by exploring how multi-agent LLMs can automate tasks in the DevOps cycle for ADS/ADAS. Substantively, it offers a practical solution to automate the software development process. Theoretically, it deepens the understanding of LLMs as collaborative agents in engineering workflows.

The findings of this research hold significant implications for practitioners, particularly in the automotive industry. It explores a practical tool design to enhance productivity while ensuring the correctness of the generated code. By supporting AD/ADAS function developers, it could perhaps accelerate the development process of such function and potentially supporting quicker deployment cycles. This can be highly beneficial for automotive companies looking to enhance innovation and reduce costs.

For researchers, this study offers a foundation for further exploration of AI-powered development pipelines, highlighting the benefits of adopting LLM-based automation in complex systems, expanding on the methods and results presented here. It also identifies key challenges and gaps in deploying such systems, which will guide future research in the field.

1.5 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 reviews the background, covering the fundamental concepts and technologies relevant to this study. It explains large language models and their applications in code generation, delves into prompt engineering techniques, and discusses the challenges associated with automotive software development. It establishes the necessary technical context for the research.

Chapter 3 examines the related work in the field, reviewing existing research, including LLM-based code generation and unit test generation, AI-assisted software development on complex systems, multi-agent LLMs systems, and automated pipelines. It discusses the strengths and limitations of current methodologies, identifying re-

search gaps and motivating the need for this study.

Chapter 4 presents the methodology of the research, detailing the design of the proposed multi-agent LLM pipeline. It outlines the system architecture and describes the roles of individual agents.

Chapter 5 & 6 presents the results and provides a discussion of the findings. It analyzes the collected data through the proposed approach. It also discusses key findings, challenges encountered, and implications of the result. In addition, it concludes the thesis by summarizing the study's key findings and contributions. It reflects on the research outcomes and suggests directions for future work to further enhance the proposed approach.

2

Background

2.1 Generative AI

Generative AI refers to a group of machine learning algorithms capable of processing input data, such as instructions or prompts, and synthesizing new content, such as text, images, or audio, by learning statistical patterns from large-scale datasets [11]. Unlike traditional AI, which often relies on discriminative modeling for decision making, generative AI uses a technique known as generative modeling. The key distinction between these AI models lies in the fact that discriminative models focus on classifying data into different categories, while generative models aim to capture the structure of the data and replicate its underlying distribution.

Many contemporary generative AI models rely on the Transformer architecture [12], a deep learning model first introduced in 2017 [13], to efficiently process input data and therefore generate more accurate output. In a transformer architecture, there are two main components, Encoder which handles the understanding of input data, and Decoder which generates the output based on the understanding made by Encoder. One of the mechanisms of this architecture, which is present in both Encoder and Decoder, focuses on the significance of individual words in the text input and their contextual relationships to one another. This, in turn, enables the model to understand the input more efficiently than previously used architectures, such as those that rely on sequence-based processing of input data.

2.1.1 Large Language Models

Large language models (LLMs) are a subtype of generative AI trained on extensive datasets, which can comprise up to billions of tokens [14], [11], to generate human-like textual output using Natural Language Processing (NLP) techniques to process input data. Some well-known examples of LLM that are widely used today include GPT-4.5 and GPT-4o, both developed by OpenAI. Due to their demonstrated ability to generate accurate text based on the specific prompt they receive, there are numerous use cases for LLMs across various industries and research domains.

A particularly relevant use case for this thesis is the integration of LLMs into software development to write new code or debug existing code. Since code can be regarded as text in this context, LLMs can be leveraged to generate specific software artifacts

and automate certain aspects of the software development process.

For adaptation to certain specialized domains or to achieve highly optimized assistance from an LLM in complex real-world scenarios, the model can be fine-tuned. Fine-tuning refers to the process of retraining the model on additional data sets from specific domains to improve its ability to perform tasks within those particular domains [15].

To use a large language model in a software application, it is typically done through a remote reference, which means API-based communication. The prompt is usually sent as an HTTP request to the target LLM server, and the output of the LLM will be returned as a response to the application server.

Another way to interface with an LLM, which works well for open-source models, is local inference, which means installing and running the LLM locally. To be able to train or even run LLMs efficiently, substantial and powerful computational hardware resources are required, in particular GPUs or TPUs. The larger the model and, consequently, the more parameters it has, the greater the GPU capacity needed for efficient handling of inference or training [16].

2.1.2 LLM-based Agents

There are several varying yet similar definitions of what LLM or AI agents entail. A well-formulated definition is that an LLM agent is an advanced AI system that incorporates one or more LLMs at its core, with the purpose of reasoning through or solving a complex real-world problem. However, in many cases, the problem is too large or complex to be solved by a single agent, which leads to the need for an agent to focus instead on solving a specific sub-problem.

The behavior of an agent is determined by the configurations provided during its design, which also define its constraints and the order of tasks to be executed. This provides the opportunity to direct the agent to execute various tasks in the order that is most optimal for the problem being solved, or to assign it the necessary permissions to complete a task. This means that AI agents are not limited to complex problems, but are also applicable to simple repetitive or time-consuming tasks [17]. In order for the agent to perform custom and useful actions for various industrial use cases, it is often necessary to incorporate a set of tools and components in addition to the response generated by the LLM. Such tools may include, for example, other software services or APIs, memory modules, or access to a specific compiler. To provide a clearer understanding of what an agent may encompass in a software context, Figure 2.1 presents a conceptual representation of the architectural components that may be required when designing such an agent.

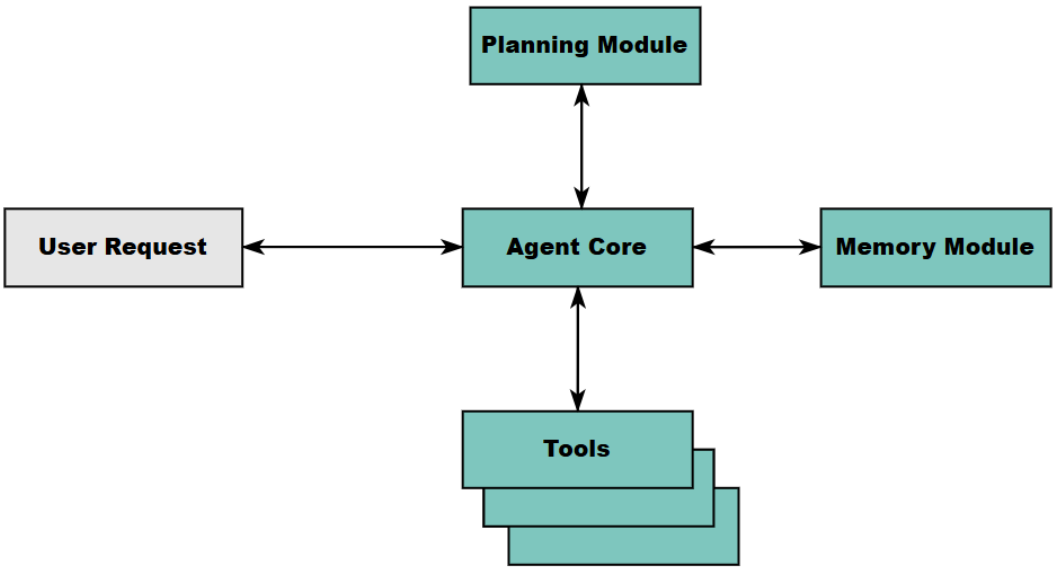


Figure 2.1: An example of general components of an agent [1].

2.2 Multi-Agent Systems

A multi-agent system (MAS) is an intelligent computerized system consisting of a number of specialized autonomous agents that collaborate to solve an industrial or research problem within a specific domain. It can also be described as a graph where the nodes refer to the agents themselves and the edges represent a type of interaction or communication between them [18]. As each agent within a MAS operates on its respective, well-defined subproblems to collectively achieve a shared goal, it is essential to establish a well-structured environment that facilitates effective interaction and coordination among them. These environments can be physical, such as factories, or virtual, such as software frameworks that regulate the data flow between agents, computational processes, and the resources accessible to each agent [18].

2.3 Prompt Engineering

As mentioned in Section 2.1, a prompt refers to the input data provided to a generative AI model in order to guide the model in generating the desired output. For many contemporary models, the prompt usually consists of textual data; however, in numerous other cases, the prompt may also consist of audio, images, or a combination of these modalities [11]. In industrial real-world scenarios where LLMs are adopted, a prompt is commonly constructed as a prompt template, a function that contains prompt instructions regarding the task to be performed by the LLM, along with one or more variables. The variables are then replaced with the actual data before the prompt is passed to the LLM. This is particularly useful when a

prompt depends on data that is provided at runtime, such as a response from another LLM in a multi-agent system, or when multiple scenarios are addressed by the same prompt.

The process of designing and iteratively refining a prompt or a prompt template through the use of various prompting techniques, with the aim of optimizing the generated output, is referred to as *prompt engineering* [14, 11, 19]. As discussed in [11], this iterative process typically consists of the following three main steps:

- **Performing inference on a dataset.** The process begins by inputting a dataset into the LLM along with a prompt that is being experimented. The objective is to establish a comprehensive basis for assessing how the LLM processes all data points in relation to the desired responses.
- **Evaluating performance.** At this step, the relevance of LLM responses is assessed through a manual review of response quality and applying quantitative metrics such as accuracy and precision.
- **Refining the prompt template.** Based on the performance evaluation, an analysis is carried out to determine whether the responses of LLM were sufficient using the applied prompt. If the LLM does not achieve the expected level of accuracy, the prompt is refined by adding more context or changing the instructions. This refinement is also performed with respect to those data points where the LLM generated incorrect or insufficient responses.

2.4 Esmini

Esmini, Environment Simulator Minimalistic, is an open-source software tool for simulating traffic scenarios to test and validate ADS/ADAS functions in a virtual environment. During the initial development of Esmini, considerable focus was placed on interoperability and portability as the aim was to provide a cross-platform, OpenSCENARIO-based simulator for basic testing [20]. Given this, and despite the ongoing updates with new features, Esmini mainly focuses on the logical execution of test scenarios, with basic visualization and very limited vehicle mechanics [20]. The architecture of Esmini can be categorized into six main modules, each responsible for managing different aspects of the simulation environment [20, 21]. These modules include:

- **RoadManager** – Manages OpenDRIVE road format.
- **ScenarioEngine** – Parses OpenSCENARIO and executes triggers/actions.
- **Controllers** – Customize the behavior of entities.
- **ViewerBase** – Renders 3D visuals of roads and entities.
- **PlayerBase** – Bridges ScenarioEngine and ViewerBase for scenario execution.

- **CommonMini** – Provides utility functions, including timing, mathematical computations, and logging.

An important module that is relevant for simulating effective test scenarios, is Controllers. This module can provide additional functionality and custom behaviors for entities during runtime [22, 20]. Esmini has a set of embedded controllers that can be used for certain pre-defined or user-defined test scenarios. In these embedded controllers, the behavior of the entity, such as vehicle speed, is predetermined and explicitly defined within the OpenSCENARIO file for the given scenario [22]. However, it is also possible to define custom controllers to meet certain simulation requirements and evaluate specific traffic scenarios.

2.4.1 OpenSCENARIO

OpenSCENARIO, defined in XML format, is a standard file format commonly used to define scenarios for scenario-based testing, which is particularly valuable for virtual driving and traffic simulation. In these scenarios, explicit instructions are given about how and where various objects, such as road conditions and all involved vehicles, are placed, and in some cases, how dynamic objects should behave at run-time. However, the behavior of some dynamic objects can be handled by an external custom controller implemented in a programming language supported by the target simulator. In Esmini, which this thesis targets, custom controllers can be written in either C++ or Python.

2.4.2 EsminiLib

As esmini provides support for handling the behaviors of objects in a scenario externally and at runtime, as discussed in 2.4, esmini provides a C++ library that exposes a range of functions. The core logic for configuring a scenario typically begins by calling the *SE_Init()* function, which initializes the scenario engine and subsequently starts the scenario. This function comes with different parameters and variants, such as *SE_InitWithArgs()*, which allow for customization of the configuration and full control over Esmini's runtime arguments [20].

Other functions related to the state of objects, such as updating the status of an object (e.g., assigning a new speed to the ego vehicle), are particularly relevant when designing an external custom controller. There are two main mechanisms exist for dynamically updating the state of an object, namely by calling either inject functions or report functions. Injecting a new internal state value creates entries in the engine queue that are scheduled over time and will be injected during scenario execution. Updating a state attribute using report functions, on the other hand, performs an immediate override of the simulation internal state, that is, the new value of that attribute is set directly without any scheduling.

2.4.3 Logging Report

In Esmini, there are various methods and formats available to generate logs after a scenario simulation has completed [23]. Some logs provide an overview, such as the default log text file `log.txt`, which is created automatically by Esmini and contains the same log entries that are displayed in the terminal window during simulation. To accommodate custom use cases, more detailed logs can be generated for in-depth analysis of the simulation of the representative scenario. It is possible to generate specific simulation state data for each dynamic entity at each simulation timestep. This includes, for example, the speed, lane, and global positions of each entity. These data entries can be saved in a structured format, such as a CSV file, where screenshots of the simulated scenario can also be created in specific time steps [23].

2.5 Automotive Software Testing

Safe autonomous vehicles undergo extensive testing at various levels to assess their performance before they are approved. Some testing can be performed at a low software level using virtual simulation, while other testing involves integrated physical hardware components. At a later stage, on-road testing is conducted, where a real vehicle is tested against various real-world driving scenarios.

2.5.1 Unit Testing

This type of testing is not limited to automotive software; it is widely adopted across software testing and is regarded as an essential and foundational level of testing in order to indicate the quality of the code base [24]. It usually targets the code functionality of software components at the class or function level. The idea behind it is to design and execute various test cases for the system in the form of code scripts, ideally using testing frameworks. Most popular programming languages have well-designed standard frameworks for this type of testing, such as JUnit for Java, PHPUnit for PHP, and GTest for C++.

2.5.2 Software-in-the-Loop

Performing low-level testing of complex autonomous systems, especially in safety-critical domains, is not sufficient to verify the system under test. The purpose is to test how such a system behaves in real-world scenarios and whether its behavior under different conditions aligns with the expected behavior.

Unlike Hardware-in-the-Loop [25], which involves hardware and physical systems, Software-in-the-Loop focuses on testing software functionality in a fully virtual world [25, 26, 9]. Testing software in a well-defined virtual simulation environment is a cost-effective method to check the safety of AD/ADAS functions in safety-critical traffic situations by continuously identifying software bugs and safety hazards. Such potential software faults would otherwise consume more resources if they were discovered during real hardware testing [25].

3

Related Work

3.1 LLMs in Software Engineering

To minimize the risk of system failure in safety-critical systems such as ADS, a process called Hazard Analysis and Risk Assessment (HARA) is performed to help identify and assess potential risks with the system. HARA is therefore a crucial first step in engineering effective safety requirements [27]. Since HARA consists of a complex process, it must be broken down into smaller subtasks so that LLMs can handle each specific aspect when generating safety requirements. The authors of [27] have designed a pipeline in which they applied prompt engineering techniques to streamline the output of LLMs, focusing on each specific safety goal. The prompts were structured according to different contexts, such as information from legacy requirements, existing safety standards, and explicit instructions on how to shape safety requirements. These LLM-generated requirements were iteratively compared with their own requirements to refine and validate the pipeline.

System-Theoretic Process Analysis (STPA) is a safety goals method used during the design phase to identify unsafe interactions between all components of an entire system, primarily within the aerospace field [10]. The challenge of applying this method in an AI enabled software-intensive system of systems such as ADS/ADAS, whose architecture is divided into multiple abstraction levels with different suppliers, lies in the interdependen cies between these levels. In addition, the development of subsystems in ADS is distributed, leading to a lack of traceability and information between individual sub-system providers on the whole system [10]. This, in turn, makes it difficult to adopt the original STPA to ADS. The authors of [10] propose an extension of STPA, called Sub STPA, which involves applying STPA with a focus solely on the specific subsystem being analyzed with respect to vehicle-level safety goals.

Adopting continuous development, deployment, and monitoring (CDDM) in various non-safety-critical software systems has proven effective for bug fixing, releasing new software updates, as well as reducing time-to-market. However, this approach presents challenging obstacles when adopted in automotive software systems due to the strict safety requirements in this domain [28]. A significant challenge lies in identifying how the changes in the software components may impact other parts of

the system, or in case these changes lead to conflicts with existing requirements [28]. For instance, a newly defined safety requirement may potentially violate another cyber-security-related requirement within the system. To address such challenges, activities such as impact analysis and conducting HARA must be performed, which is time-consuming and requires extensive resources [28]. Moreover, the need to redo activities related to verification and validation of the affected software leads to additional challenges. Specific difficulties in verifying new AI-enabled features are understanding the differences between the simulation testing and real-life situations using actual hardware.

3.2 LLMs for Code Generation

Recent advancements in AI, particularly in LLMs, have revolutionized the generation of program code. Among these, pre-trained GPT-4 stands out as a powerful model that is significantly transforming software development by offering new possibilities for automation [6]. However, despite these advancements, delivering high-quality, domain-specific, fully functional code, particularly in the automotive domain, remains a significant challenge.

One technique used to enhance the accuracy of LLM response is prompt engineering, which involves effective prompts to guide the model toward more precise and context-aware outputs. Prompt engineering has proven to be a powerful method for improving both functionality and quality in software development. In [6], the authors introduced their own augmented prompt framework, called *Prompt-FDC*. This framework integrates basic functional requirements, domain feature generalization, and standard specification constraints. This framework progressively refines and generalizes domain requirements, enabling a more thorough understanding of the domain and facilitating the generation of higher-quality code. Moreover, *Prompt-FDC* has proven effective in addressing safety-critical software requirements, such as those specified in *ISO 26262*, a key standard in automotive software development. Furthermore, the researchers in [29], also highlighted the importance of prompt engineering in enhancing model performance. They specifically mentioned *Chain-of-Thought (CoT)* prompting, a technique that significantly boosts LLMs' ability to handle complex reasoning by incorporating structured examples in the prompt.

The study in [6] highlights a key challenge: generating code based on overall requirements often results in code that is not directly usable or runnable. In contrast, generating code based on specific requirements, particularly when enhanced through augmented prompts, achieves higher levels of functionality and quality.

A promising approach explored in [27] to enhance LLM performance is task decomposition. This technique involves breaking down complex tasks into smaller, more manageable subtasks. By breaking down the problem into smaller problems, LLMs can generate more precise and contextually relevant responses.

Nevertheless, achieving high-quality results often requires iterative refinement of the

output, as some requirements may not be fully implemented in the initial output. Subsequent iterations focus on refining domain-specific requirements to achieve better results [6].

Another study also presents an approach to enhance the accuracy of LLM-generated outputs, which is the Retrieval-Augmented Generation (*RAG*) framework. To generate more accurate, domain-specific code and improve the relevance of code generation tasks without the need for continuous model fine-tuning, the *RAG* framework is highly effective [2]. It combines a retrieval system with a generation system, making it particularly useful in scenarios where an LLM may lack some of the necessary information in its training data and requires external sources to answer queries or generate content. Additionally, the *RAG* framework helps address critical issues such as hallucination and outdated information, which are common challenges with standalone LLMs.

The author of [2] tested this technique within the *AUTOSAR* framework, which stands for AUTomotive Open System Architecture. *AUTOSAR* aims to establish an open, standardized, and highly modular software architecture for automotive systems, facilitating the integration of various components from different manufacturers. The experiment produced positive results, showing that the use of *RAG* significantly enhanced the outputs of LLMs. For the evaluation of the *RAG* system, the authors utilized the Retrieval-Augmented Generation Assessment (*RAGAS*) framework, which offers a comprehensive evaluation pipeline for assessing the performance and effectiveness of *RAG* in automotive applications. They also performed a human evaluation by designing a survey to gather qualitative feedback. However, according to [29], the use of *RAG* comes with certain implications, as the quality of retrieved information directly affects overall performance.

According to [29], LLMs such as *GPT-3* and its successors have made notable advances in automated code generation. However, challenges remain in their practical usability. The researchers in [29] highlighted the use of various benchmark datasets to evaluate LLM performance in code generation. These benchmarks primarily assess the model's ability to generate correct code, rather than verifying whether the generated code is practically usable. Examples of such benchmarks that have been used include *HumanEval*, *MBPP*, *BigCodeBench*, and *RepoEval*. *HumanEval* consists of 164 manually scripted Python problems but lacks real-world complexity. *MBPP* includes 974 entry-level programming tasks with descriptions, reference solutions, and test cases. *BigCodeBench* offers a more advanced benchmark with 1,140 complex Python tasks, designed to test LLMs' ability to handle cross-library function calls and intricate instructions. *RepoEval*, in contrast, focuses on repository-level code completion, assessing challenges like import dependencies, parent classes, and cross-file interactions. This makes *RepoEval* particularly relevant for evaluating LLMs in large-scale software development. However, repository-level code generation still faces challenges due to complex interdependencies between files, shared utilities, and configurations.

Beyond standalone LLMs, autonomous multi-agent systems have emerged as a

promising approach to code generation. These systems leverage LLMs as central reasoning engines, facilitating collaboration with other tools. This technology enables a more structured and iterative development process, helping to overcome some of the limitations of traditional LLM-driven code generation [29].

3.3 LLMs for Testing

Various industrial and research techniques for test generation have been employed to achieve partially or fully automated test generation in software development. These techniques include feedback-directed random test generation and fuzzing [30]. A particularly notable observation from various surveys on LLM-based test generation is that the test cases generated are often insufficient for verifying the correctness of the source code [31]. In many cases, the generated test cases lack relevant assertions, instead containing spurious or overly generic assertions, such as merely checking whether a variable is non-null rather than validating its expected behavior [32, 33].

To achieve more promising results in test generation using LLMs, numerous studies have focused on pre-training the model on additional datasets through various fine-tuning techniques. However, the authors of [31] pursue an alternative approach, aiming to enable an existing LLM to generate reasonable test cases without any additional training. Their approach rely heavily on the design of clear and well-structured prompts. These prompts include explicit instructions to guide the LLM, along with the signature of the function under test, its source code, usage example, and relevant documentation if available. The generated test cases are thereafter compiled and executed to validate their syntactic correctness and functional accuracy. In the event of a test case failure, the corresponding error logs are incorporated into a subsequent prompt to enable the LLM to diagnose and rectify the identified errors. As a final conclusion, the authors assert that further research is necessary to explore how LLMs can be utilized for designing test cases with non-trivial assertions. While automated LLM-based test generation can serve as a valuable approach to accelerate the testing phase, it can not currently replace the need of manually designing, or at least reviewing, test cases [31].

However, the author of [31] investigated LLM-based unit test generation as an alternative to established techniques such as fuzzing, feedback-directed random testing, dynamic symbolic execution, and evolutionary algorithms. These traditional techniques often suffer from two main drawbacks: the generated tests are typically less readable than manually written ones, and they either lack meaningful assertions or include only generic checks.

In contrast, the LLM-generated tests in the study demonstrated improved readability and incorporated non-trivial assertions. Moreover, the results showed that LLM-based approaches achieved higher statement and branch coverage compared to feedback-directed methods. The authors used the GPT-3.5 Turbo and Code-Cushman-002 models, providing prompts that included the function signature, documentation, usage examples, source code, and in case of failure, the failing test

cases and associated error messages. They concluded that more advanced LLMs could further enhance the results.

Another relevant study presented in [34] introduced AGOBETEST, a framework for automated unit test generation and evaluation at the Java class level. AGOBETEST assesses the quality of generated test cases using key metrics such as code coverage, mutation testing, and test smells. In their evaluation, the compilation success rate of the generated test classes ranged from 64% to 76%, with a relatively low test pass rate of 30%–38%, often due to syntax errors or incorrect import statements. To improve these results, the authors proposed incorporating human-in-the-loop interventions or automating the extraction of contextual information from project configuration files.

The study also benchmarked LLM-generated test suites against human-written ones. While human-written tests were 100% fully compilable and executable, tests generated by LLMs, particularly GPT-4o, achieved higher scores in line, method, and instruction coverage, reaching 85%–87%. However, human-written tests outperformed in branch coverage (80%) and mutation score (69%), indicating stronger robustness. Overall, the findings highlight the promise of LLMs in generating meaningful, high-coverage unit tests, while also acknowledging persistent challenges in reliability and test robustness.

Another study [8] investigated the effectiveness of LLMs in automated unit test generation by focusing on the transformation of expert-defined test descriptions/scenarios into executable JavaScript test code. The test scenarios, totaling 106, were collected through interviews with experts. The study evaluated several LLMs, including GPT-4, Claude 3-5, Command-r-plus-08-2024, and LLaMA 3.1.

In their setup, the prompt provided to the models included the function under test along with the corresponding test scenarios. The evaluation of the generated test code was based on multiple metrics, including success rate, code coverage (statement, branch, function, and line), and mutation testing scores.

Among the tested models, Claude achieved the highest performance, with a success rate of 93.33%, statement coverage of 98.01%, branch coverage of 95.39%, function coverage of 99.22%, line coverage of 98.30% and a mutation score of 89.23%. These results demonstrate that LLMs, particularly Claude, can effectively translate high-level test descriptions into high-quality, executable unit tests with strong code coverage and robustness.

3.4 Collaborative LLM-based Agents

Futhermore, LLM-based agents offer opportunities for addressing complex real-world Software Engineering (SE) tasks. To request an update to a software system using natural language and achieve effective results from LLMs in complex SE scenarios, a systematic breakdown of the tasks to be performed is needed. This involves designing at least one dedicated agent for each phase of the SE lifecycle. For the

requirement engineering phase alone, the authors of [6] propose a subpipeline consisting of an agent for each of the following tasks: elicitation, modeling, negotiation, specification, and verification of requirements. The purpose of this is to ensure that the requirements are clearly understood by the pipeline and to minimize the risk of ambiguous requirements before initiating code generation [6]. The same principle applies to the code generation phase, which involves multiple agents not only to verify that the generated code is statically accurate but also to ensure it is optimally written from both an algorithmic and functional perspective. However, the authors' proposed pipeline includes human collaboration in multiple phases at different levels in order to get sufficient results. The primary reason for not designing a fully automated pipeline was that human participation enhances the performance of agents and prevents bottlenecks that agents might otherwise encounter during software development in a fully automated pipeline [6].

According to [2], employing collaborative multi LLM agents significantly offers advancements in solving complex challenges and tasks in software engineering. LLMs enhance the cognitive capabilities of agents, enabling better collaboration and autonomous problem-solving. The paper also emphasizes ways to improve LLM agent performance, including prompt engineering, using domain-specific agents, expert consultation, and iterative refinement. However, the complexity of software projects presents challenges, as a single LLM-based agent may struggle with high-level tasks. To address these challenges, the paper suggests employing multiple agents in a hierarchical structure, where high-level agents delegate tasks to lower-level agents in order to improve task allocation and manage interdependencies effectively.

4

Methods

This chapter outlines the research methodology used for this thesis. The chapter first introduces the principles and stages of the Design Science Research (DSR) methodology, followed by a detailed explanation of how it was applied throughout the research process. The chapter also explains the design and development of the proposed pipeline, the integration of various agents, and the evaluation framework used to assess the effectiveness of the solution.

4.1 Design Science Research

This thesis adopts the DSR methodology to design, develop and evaluate a multi-agent LLM-based pipeline for automated software development in ADS/ADAS. In addition, it investigates the challenges and limitations of LLMs within this domain. DSR is an appropriate approach for this research as it focuses on the iterative design, implementation, and evaluation of an artifact that addresses a real-world problem. This study follows the six-stage DSR framework as outlined by Hevner et al. [35] and Wieringa [36]. As shown in Figure 4.1, the process begins with identifying the problem, followed by setting specific objectives to address it. After that, an artifact is designed and developed as a potential solution. It then undergoes testing and evaluation to assess its effectiveness. In the last stage, the findings are communicated to contribute to both practical applications and academic knowledge.

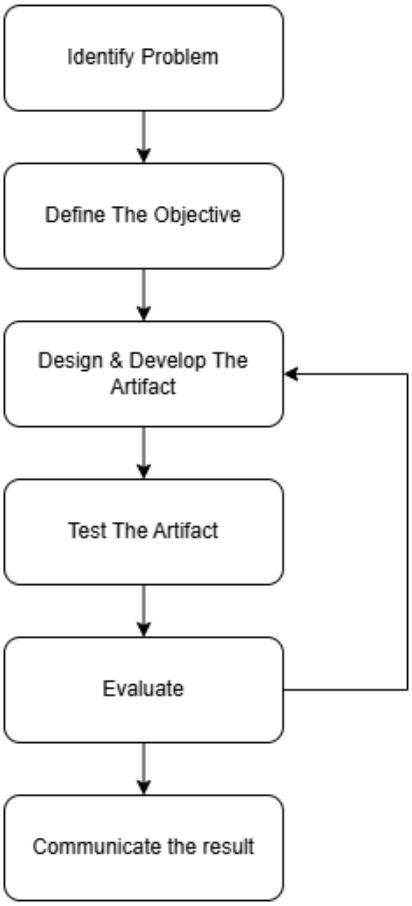


Figure 4.1: The six stages of the DSR process.

4.2 Delimitations

The implementation of the pipeline is tailored to the Esmini simulation, which uses specific OpenSCENARIO files to define driving situations and behaviors. The generated C++ code functions as a custom controller that manipulates vehicle behavior within Esmini and represents a single software unit. As a result, the findings are specific to Esmini and may not be directly applicable to other simulators or scenario formats.

Additionally, the pipeline includes a unit test generation stage to validate the functional correctness of the generated code. However, due to time limitations, test coverage analysis and satisfaction of all possible test cases not addressed in this study. The focus remains on basic functional validation rather than comprehensive testing.

Moreover, the pipeline is designed to support a limited subset of ADS/ADAS functionalities, with a specific focus on features such as Automatic Emergency Braking (AEB), Adaptive Cruise Control (ACC), Collision Avoidance (CAEM) and change lane (CHANGE_LANE). As a result, the findings and demonstrated capabilities

may not be directly generalizable to other AD/ADAS functions outside this scope.

4.3 Design of The Proposed Solution

To improve the efficiency of the development process for ADS/ADAS, and to overcome the limitations and challenges identified in existing tools and related research papers, this work proposes a fully iterative, multi-agent system that supports the automated development of AD/ADAS functions in C++ programming language.

The system assumes a minimal existing codebase that both provides the necessary interfaces for communicating with the Esmini simulator and includes supporting functions that the generated code can call, the environment setup is illustrated in Section 4.5. The functions under study are generated and added to this codebase for validation. Parts of this system specification are provided to the LLM in the prompt to ensure that it produces code compatible with the existing project.

The overall architecture of the proposed system is illustrated in figure 4.2. The process begins with user input that describes the desired function. Based on this input, the system generates the corresponding function implementation, followed by the generation of unit test cases, designed to be compilable with the Google Test framework. These test cases serve as the first layer of validation, helping to detect potential bugs in the generated function before simulation. To ensure the quality of unit tests, the system verifies compilability, executability, and coverage metrics (line and branch), reducing the risk of incorrect validation. Additionally, mutation testing is performed, with the resulting mutation score reported to assess test robustness. While these measures help catch some defects, they do not guarantee full correctness of the software, serving instead as a complementary check prior to simulation.

After the initial layer of validation, the generated code is executed within the Esmini simulation. Logs from this execution are analyzed in the final validation stage to ensure functional correctness in a simulated context. However, if failures occur at any validation layer, the system loops back to the code generation agent for refinement.

As a final step before deployment of the generated function, a code reviewer manually inspects the pipeline output, or baseline, which includes the generated function, its unit tests, and all relevant logs and reports. Because the pipeline currently depends solely on LLM-generated artifacts and automated checks, the reviewer serves as an additional safeguard against subtle semantic errors, unsafe behavior, or gaps that automated validation can miss, helping ensure correctness, safety, and compliance with development standards, particularly in safety-critical domains like ADS/ADAS.

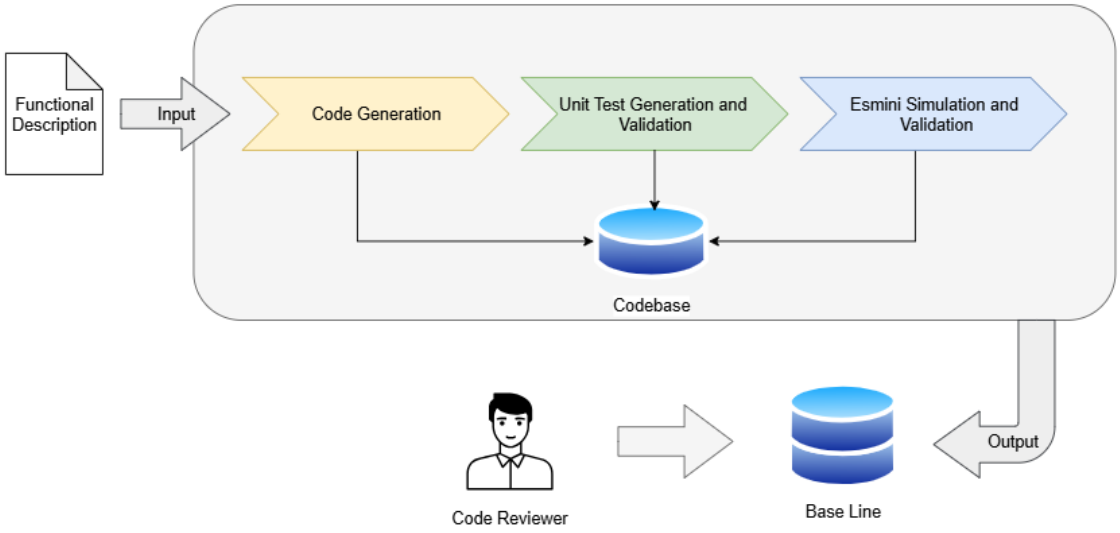


Figure 4.2: The overall architecture of the proposed system.

4.4 Data Set

In this research, we do not utilize any specific datasets. The only input available is the textual specification of the ADAS functions that were reported by Volvo Cars in a research paper [9]. As the pipeline needs more detailed functional description of the target ADAS functions, we are responsible for explicitly constructing the functional descriptions for each relevant ADAS feature when designing the input data for the purpose of experimentation. These descriptions are determined on the basis of the specifications of the ADAS functions. The functions are specified as follows:

- **F1:** "*The ego vehicle shall start braking if the speed exceeds 10 m/s²*".
- **F2:** "*The ego vehicle shall perform a lane change to the right if there is a vehicle in the same lane as the ego vehicle*".
- **F3:** "*The ego vehicle shall adapt its speed to the vehicle in front to avoid collision (exhibiting so-called Adaptive Cruise Control behaviour, ACC)*".
- **F4:** "*In unsupervised Collision Avoidance by Evasive Manoeuvre (CAEM), the ego vehicle shall perform a lane change to avoid imminent collision with the vehicle in front. The lane change is preferably to be conducted to the left*".

4.5 Development Environment Setup

To enable an uncomplicated development of the custom controller and to avoid unnecessary complexity and ambiguity that the multitude of functions and attributes in esminilib may cause for the LLM during code generation, we design a dedicated

interface for this purpose. This interface acts as an API that manages the communication with esmini and encapsulates low-level simulation details, with the aim that custom controller is dedicated exclusively to the logic related to the target ADAS function. The interface consists of two C++ source files, *Vehicle.cpp* and *State.cpp*. Vehicle is solely a data class that represents a vehicle within the simulation and contains the following attributes, which are considered relevant to our problem domain:

- A unique vehicle identifier.
- Spatial coordinates in a 3D Cartesian space (x, y, z).
- The vehicle's current speed.
- Heading angle indicating the vehicle's orientation.
- Lane identifier (*lane_id*).
- Longitudinal position (s).
- Lateral offset from the road centerline (t).

The State class contains functions that continuously fetch internal simulation state data, as well as a set of encapsulated functions that the custom controller needs in order to implement its logic. These functions are defined as follows:

- `getVehicles()` — Returns a list of all vehicles in the simulated scenario, including the ego vehicle.
- `setEgoSpeed(float speed)` — Sets the ego vehicle's speed to the specified value.
- `brakeEgo()` — Commands the ego vehicle to brake.
- `switch_lane(int lane_id)` — Initiates a lane change of the ego vehicle to the specified lane.

Figure 4.3 shows the integration between the custom controller and the Esmini simulator. As shown, the State class serves as the communication link between the two components. It leverages the Esmini library to retrieve and update vehicle data, which is then stored in the Vehicle model. The CustomController class performs its actions by interacting with the State class.

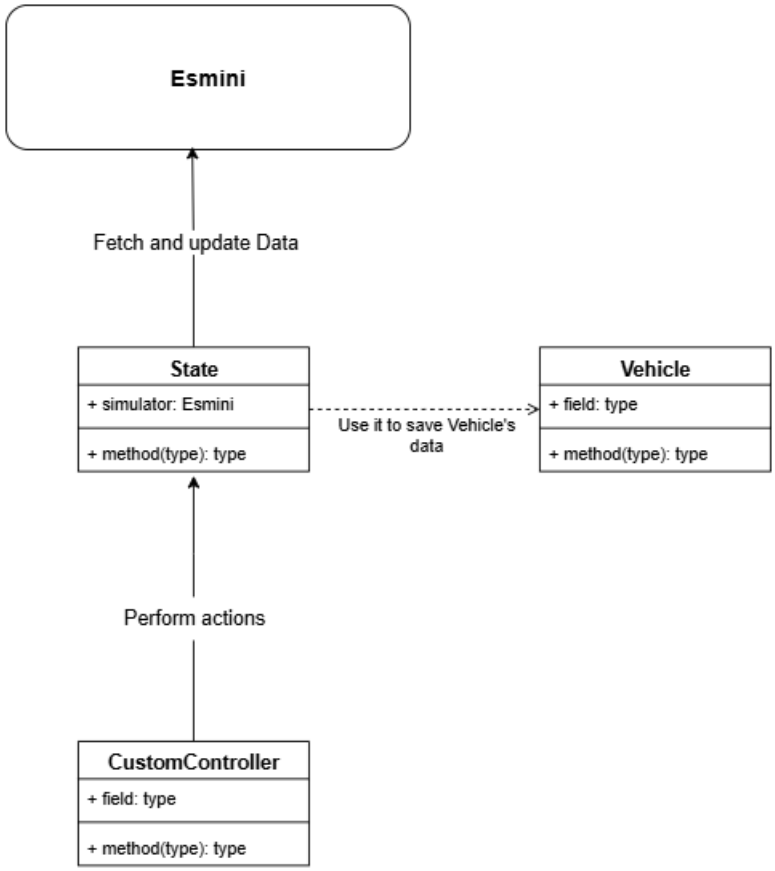


Figure 4.3: The communication flow with Esmini.

4.6 Definition of Done

The generated function is considered done when it meets all the following criteria:

- **Compilable**: The code compiles successfully without errors or warnings in the target development environment.
- **Executable**: The software runs without crashes or critical runtime errors.
- **Pre-simulation Validation**: The software must achieve at least 75% line and 40% branch coverage by passing relevant unit tests. These thresholds provide a preliminary check that the function is partially correct and exercised before simulation, they do not guarantee full correctness. Mutation testing is also integrated to assess test robustness, with experiments reporting a minimum mutation score of 28%.
- **Simulation Validation Passed**: The software operates correctly within the simulation environment, producing expected outputs and behavior.

4.7 Iterative Development Process

The final pipeline was designed through different iterations of refinement conducted throughout the research. Each iteration contained multiple sprints, and the following subsections explain more in-depth how the pipeline evolved in each iteration. The main issue with the first iteration was that the prototyping code could not be relied upon as a valid input to the pipeline, whereas the primary challenge in the second iteration stemmed from a lack of context for the LLM and cumbersomely long prompts.

4.7.1 First Iteration

In the first iteration, our aim was to explore and gain an understanding of how open-source models would handle code generation within a specific domain. Generating code using LLMs was not entirely new to us, as we had previously used Chatgpt for personal use. However, the idea in this thesis was to automate code generation by sending iterative requests to LLMs until the desired output was achieved. Therefore, we planned to initiate code generation in a straightforward manner by generating C++ code from prototyping code written in Python. This prototyping code was part of experiments previously conducted at Volvo Cars and, in this iteration, acted as input data for our proposed pipeline. The prompt sent to the LLM was therefore aimed at transforming the Python code into C++ code while taking the simulation environment into account. When this approach was tested on simple functions such as F1 and F2, we were, during some manual trial runs, able to generate compilable C++ code.

Since the LLM quite often did not generate compilable C++ code, we added a mechanism to our pipeline for error correction. To prompt the LLM to fix the compilation errors found in the previously generated code version, we provided both the code and the error logs within the prompt.

At this stage, the pipeline struggled to resolve these compilation errors and, in many cases, ended up in an infinite loop. A key reason for this was that the LLM frequently made assumptions about the existence of API functions that were not present and used them to fix the existing errors, which in turn caused new compilation errors.

Another bottleneck observed during iterative error correction was that the LLM sometimes returned code with placeholders in the code lines where the compilation error had occurred. This led to a situation where the generated code would compile, not because the errors had been resolved, but because they had been removed or replaced with TODO-comments. As a solution, we implemented an additional mechanism in our pipeline that would compare the compiled generated code with the prototyping code and return true if they were structurally equivalent before proceeding to the test generation stage.

For the unit test generation, this iteration started by designing a simple prompt that could be invoked to an LLM to generate automated unit tests. The primary

goal of this iteration was to evaluate the ability of an LLM to generate compilable test cases when provided with minimal contextual information. Through this experimentation, a number of early limitations were uncovered, most notably, the insufficiency of basic prompts in conveying the necessary context. These findings highlighted the need to enrich the prompts with more detailed information, such as specific implementation details and constraints, in order to enable at least the generation of compilable test cases. This initial iteration laid a solid foundation for designing more effective prompts and structuring more robust workflows in the subsequent development iterations.

4.7.2 Second Iteration

In the second iteration, we modified the pipeline to generate C++ code from a functional description instead of from the prototyping code. This decision was based on an update of the requirements for this research. Certain mechanisms from the first iteration, such as the structural comparison between the generated code and the prototyping code, were retained.

At this stage, the generated code exhibited multiple issues, such as the use of non-existent esmini API functions or incorrect use of existing functions. Furthermore, the LLM showed a tendency toward overgeneration, where the output included additional components that were not requested in the prompt, such as a main function or simulation setup code.

To improve the LLM-generated code and support the LLM in generating the specifically requested code with the correct structure and use of the available API functions, the prompt was extended to provide more in-context learning. To this end, the prompt was updated to also include the complete documentation of the *Vehicle* and *State* classes and their attributes, as well as an example illustrating how a custom controller is intended to be structured.

For the unit test generation, this iteration focused on addressing the limitations identified during the initial iteration. In this phase, the prompt was enhanced with richer contextual information, including system specifications and explicit constraints. This iteration aimed primarily to evaluate the extent to which enriching the prompt with more detailed context could produce compilable test cases. To further improve reliability, an iterative refinement mechanism was introduced during this phase. In this feedback loop, test cases that failed to compile were analyzed, and their compilation error logs were fed back into the LLM as part of the prompt, enabling the agent to revise and regenerate the tests accordingly.

While this iteration demonstrated clear improvements in the structure of the test cases, it also revealed that the generation of fully compilable and executable tests was still not reliably achieved. One significant issue was the lack of test case manageability; a single test case failure (e.g., a compilation error) could impact the entire test suite, making it difficult to trace and isolate faults. Additionally, the generated tests often included redundant code, repeated parts already present in the system, or

incomplete and skeleton implementations that lacked sufficient detail for execution. Furthermore, syntax errors and incorrect or missing *include* statements contributed to frequent compilation failures. In many cases, the generated test suites remained minimal and failed to provide sufficient coverage. These findings demonstrated that simply enriching the prompt with detailed contextual information is not sufficient to produce reliable output. The overall design of the system also plays a crucial role. Consequently, these results highlighted the need for a more structured and modular approach to further improve test generation performance.

4.7.3 Third Iteration

Based on the observations from iterations 1 and 2, we redesign the code generation in our final solution to address the bottlenecks present in the previous approach. As mentioned earlier, we need to rely on the functional description of the custom controller to be generated rather than on the prototyping code. This is partly because these requirements are clear and intended for use, making the entire pipeline more robust, and partly because fully functioning prototyping code is not always available. However, as explained in iteration 2, using raw functional descriptions alone is insufficient for generating domain-specific, functioning code within the esmini environment.

The new code generation approach is more focused on functional decomposition and further subtasking, which, in the approaches of previous iterations, were performed within a single LLM request. That is, the decomposition of the functional description into distinct logical conditions, the insertion of the required API functions, the generation of the step() logic in C++, and the integration of the code into the class template.

For the unit test generation, this iteration addressed the challenges uncovered in the previous phase by introducing a modular and step-wise test generation approach. Instead of relying on a single agent to generate an entire test suite in one step, the task was decomposed into smaller, more manageable sub-tasks. These included generating test descriptions in natural language, generating the corresponding code for each test description, compiling and running the generated code, and finally adding it to the appropriate file. This shift enabled greater control, accuracy, and maintainability across the generation process. It also made it easier for the LLM to focus on one specific task at a time, rather than attempting to handle multiple aspects simultaneously, resulting in more consistent and reliable outputs.

During this phase, several prompt engineering techniques were introduced. These included adopting a well-defined prompt structure, applying the one-shot example technique, and crafting more targeted, context-aware prompts that delivered only the most relevant implementation details and constraints. Together, these strategies helped minimize irrelevant information, increased prompt clarity. Moreover, this iteration introduced a new mechanism integrated into the unit test generation pipeline called the Coverage Guided Test Suite Extender. It is designed to improve test suite completeness by identifying untested paths and automatically generating

additional test cases to cover them.

This structured approach led to the development of a significantly more robust and scalable solution, approaching a production-ready implementation. The developed system showed substantial improvements in modularity, the reliability of generated tests, and its potential for seamless integration into automated software development pipelines.

We observed some issues during this phase, there were some cases where the LLM generated test cases that successfully compiled and executed, but failed, even though the software-in-the-loop contained no defects or bugs. In such cases, the failure was due to issues within the generated test cases themselves, not the underlying software. To mitigate these false positives and prevent the pipeline from incorrectly initiating a refinement cycle through the code generation agent, an oracle-based validation mechanism was integrated into the system. Specifically, before forwarding failed test cases to the code generation agent, they are first checked by the oracle-based validation agent. This agent determines whether the failure is due to an actual defect in the software or in the test case. The oracle-based validation agent rely on the functional description, the software-in-the-loop implementation, the test case description and the test code to make a decision.

The following sections outline the design and implementation details of the final artifact produced through this iterative development process.

4.8 Implementation of The Final Artifacts

The core of the artifact is a multi-agent system, where each agent is responsible for a specific task within the software development lifecycle. The main pipeline is structured into several key stages, including code generation, unit test generation, simulation execution, and validation. Notably, both the code generation and unit test generation stages are implemented as independent pipelines composed of multiple collaborative agents. Each of these sub-pipelines handles its own internal workflows, such as prompt construction, iterative refinement, and validation, before passing outputs to the next stage. This layered architecture ensures a modular, extensible, and fully automated development process.

The high-level architecture of the pipeline is visualized in figure 4.4, showing the stages and key components and their interactions within the pipeline. The pipeline takes a functional description as input, which passes through several stages to produce a custom controller implementation written in C++, along with its corresponding test cases. The process starts by generating a custom controller based on the provided functional description. After the successful completion of the code generation stage, the subsequent stage begins, generating unit test cases using the same functional description and executing them. If any test cases fail during execution, they are first passed to the oracle-based validation agent, which determines whether the failure is due to an issue in the test case itself or in the software-in-the-loop.

This step ensures that only genuine failures are forwarded to the code generation stage for correction, thereby avoiding unnecessary modifications to an already functional custom controller. Notably, unit test cases are generated only during the first iteration, in subsequent iterations, the pipeline reuses the existing test suite.

The next step involves running the simulation to evaluate the generated code, while capturing the logs for validation. At the last stage, the created logs are analyzed to validate the correctness and functionality of the generated C++ code. If a failure occurs at any stage of this pipeline, the responsible agent receives an error log as a prompt, to fix the issue. This iterative refinement continues until a satisfactory result is achieved.

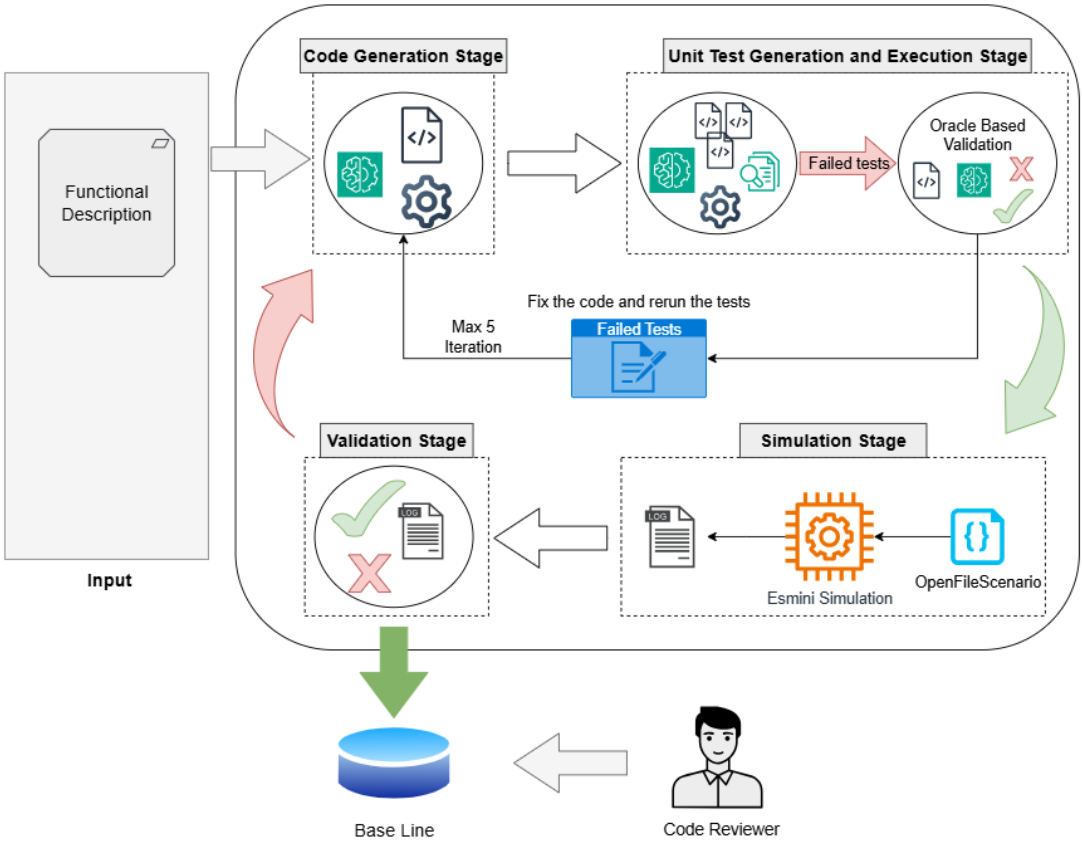


Figure 4.4: The high level architecture of the pipeline.

4.8.1 Code Generation

To generate useful code using LLMs, clear prompts that contain some form of sufficient specification on how the code should behave should be provided to the LLM.

When code is to be generated in the code generation pipeline, functional decomposition is used to divide the problem into smaller subproblems, as Figure 4.5. A first step before any code generation is initiated is to prompt the LLM to break down the input data, the functional description of the target ADAS function, into different logical conditions and actions written in natural language. We specify that

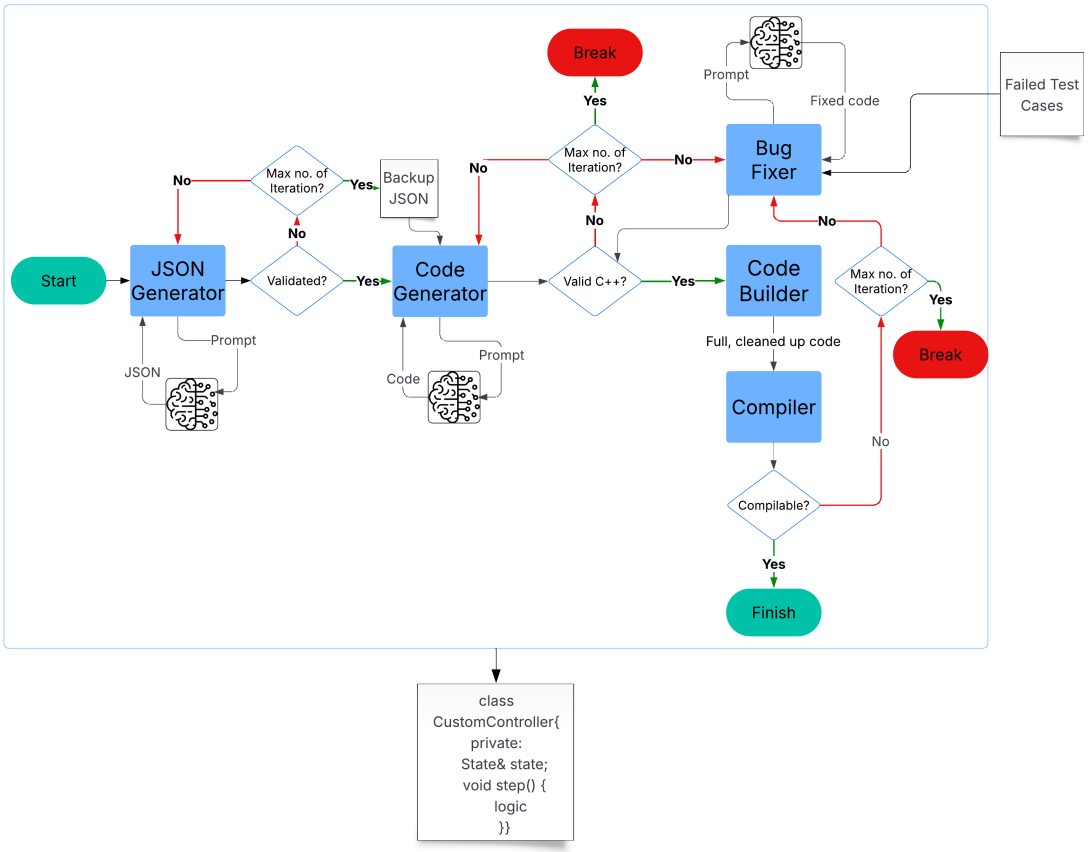


Figure 4.5: The subprocesses of the code generation.

the output of this step shall be a JSON object in order to enable validation against a JSON schema whose structure we have designed.

A separate prompt is then sent to the LLM to insert the relevant API functions needed to perform each action, respectively. This prompt includes a description of the API functions available, their signatures, and brief documentation on the purpose of each function. The output at this stage is still a code-free JSON object which serves as a structured preparation step before the actual code generation.

Furthermore, the JSON object is sent to another agent tasked with transforming these instructions into C++ code. In the prompt for this step, see Figure 4.6, more detailed context about the environment API is provided. By way of illustration, the attributes and functions of each class, in this case State and Vehicle, are presented to equip the LLM with additional particularized context. With the aim of simplifying the process, the LLM is requested to generate the corresponding code only for the `step()` function of the custom controller, without including the class definition or any other structure-related components typically present in a C++ class. The reason is that the LLM struggled with these aspects and made assumptions when attempting to correct the compilation errors during the iterative error correction in the first development iteration.

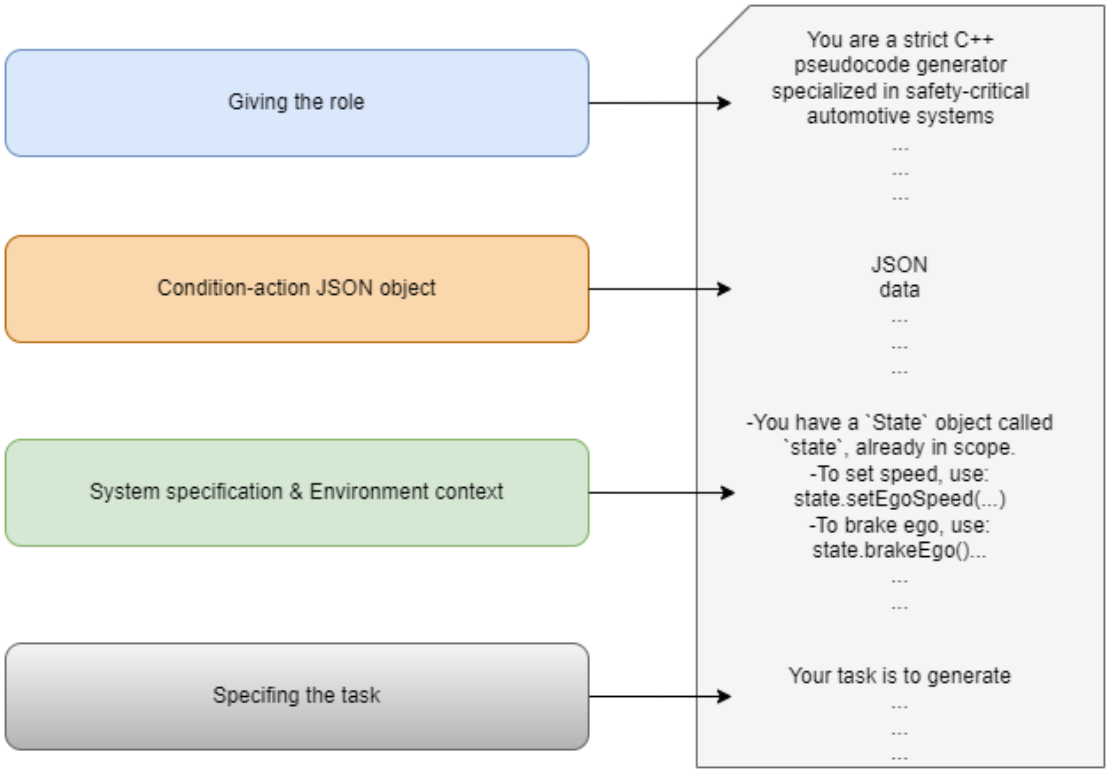


Figure 4.6: The structure of the prompt used for code generation.

Upon receiving the response to the query, the code block comprising the `step()` logic is extracted. Once extracted, we insert the code into the class template that aligns with the interface we have designed for communicating with esmini API. At this stage, we perform a basic check of the dependencies that are used by the LLM when it generates the code in order to clear out any redundant dependencies that might cause compilation errors. If the generated code is not written in C++, as has been observed in certain instances, the agent rejects the response and issues a new request for code generation.

To guarantee that the code is syntactically correct and compilable, another sub-module compiles the generated code as an implementation file (with cpp extension). Before compilation, the system ensures that the file is located within the appropriate environment required for successful compilation, that is, an environment where all dependencies, such as related headers or source files, are accessible.

Once more, if the code fails any of the aforementioned verification steps, the agent requests a new code generation from the LLM in an iterative manner, incorporating relevant feedback and more context. For instance, if the code is not compilable, a prompt template designed for error correction is used, in which the buggy code containing inline error messages as well as the compilation error logs are provided as parts of the prompt. The structure of the prompt used for error correction is illustrated in Figure 4.7.

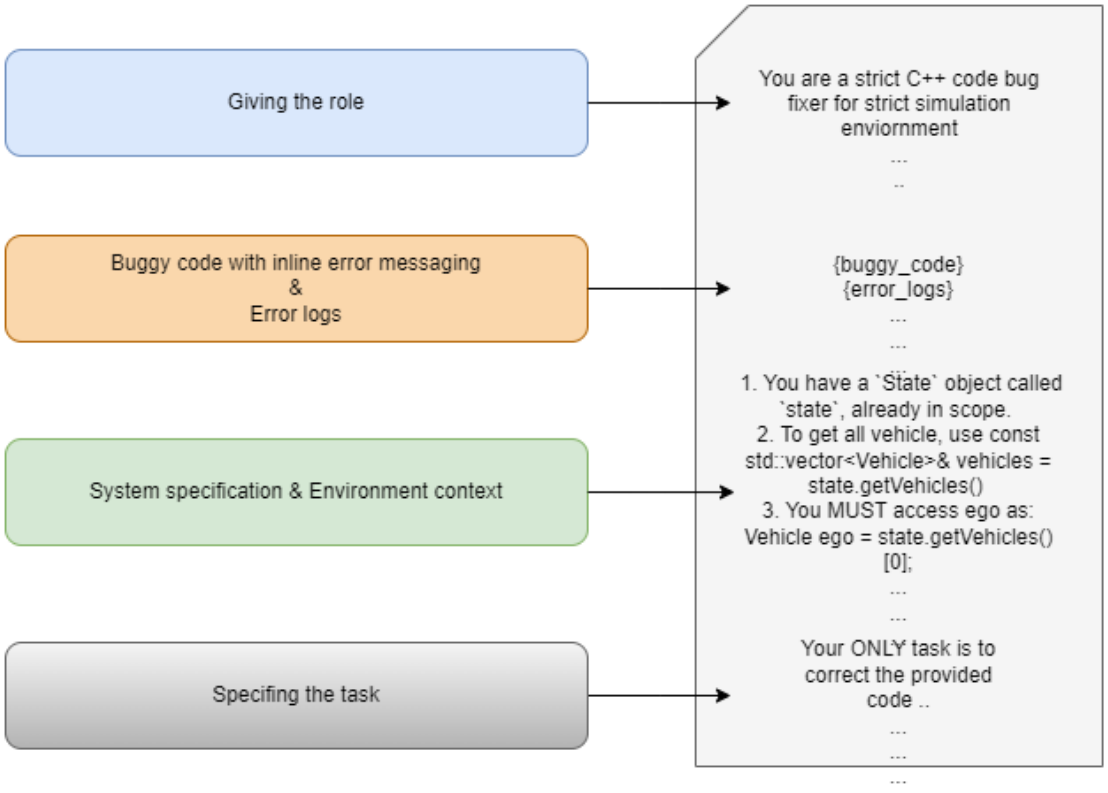


Figure 4.7: The structure of the prompt used for compilation error correction.

4.8.2 Unit Test Case Generation

This stage is responsible for generating unit tests using the provided functional description to validate the C++ Custom Controller code produced by the code generation agent. It acts as an additional validation layer, ensuring the generated code meets expected functionality. For example, if the code produced by the previous stage contains defects, this testing process can identify them. However, the main advantage of this validation step is its ability to detect defects in the generated C++ code early in the pipeline, before reaching more complex and resource-intensive stages, such as scenario preparation and simulation execution. Identifying issues during unit testing is more efficient and cost-effective, avoiding the additional cost of using OpenSCENARIO files, running simulations, and analyzing logs. By detecting issues early, the pipeline can iterate faster and improve the generated code without having to run the entire simulation workflow.

The Google Test (gtest) framework is used, as it offers a variety of assertions and testing features for software written in C++ [37]. Additionally, the Google Mock (gmock) framework is used to simulate the behavior of the Esmini simulation state [37]. This is essential because the generated custom controller code is designed to interact with a simulation environment. Therefore, mocking the simulation's state is essential for being able to run the tests against the software. By using gmock, the system bypasses running the simulation at this stage, allowing to focus only on

functional validation of the generated code. These requirements are clearly specified within the prompt to guide the agent's behavior accordingly.

Figure 4.8 illustrates the internal structure of this stage, highlighting the collaboration between agents to generate unit tests and extend the test suite. This stage is composed of multiple sequential steps. The process begins with the generation of test case descriptions in natural language in a predefined structured format, as shown in Figure 1. These descriptions are extracted from the LLM outcome and stored in a list, where each item represents an individual test case scenario. For each description, the system prompts the LLM to generate the corresponding C++ unit test code independently.

Once generated, each test case is compiled and executed in isolation to assess its validity. If the test case fails to compile or encounters a runtime error during execution, the system initiates a regeneration loop to attempt generating a correct version of the test case. This loop is constrained to a maximum of five iterations to prevent infinite retries. If, after these attempts, the test case remains non-compilable or non-executable, it is logged for further analysis, and the system proceeds to the next test case. Based on the outcome, compilable and executable tests are then sorted into two distinct files: those that compile, run successfully, and pass their assertions are appended to a passed tests file, while those that compile and run but whose assertions fail are recorded in a failed tests file. Keeping these results separate streamlines traceability and simplifies subsequent refinement and expansion of the test suite.

To enhance both effectiveness and reliability, the pipeline was augmented with a coverage-guided test suite extender mechanism. After generating the initial tests, the system performs a coverage analysis using the gcovr tool to determine the percentage of source code executed by the suite. If coverage is below a predefined threshold, the system constructs a new prompt for the LLM that contains: (1) the functional requirement, (2) the existing test descriptions, and (3) the relative data of the coverage analysis report. The LLM then generates additional test-case descriptions to target these gaps, and those descriptions are passed through the same code-generation and validation steps as before, and coverage is re-evaluated. To prevent an infinite loop, this cycle is limited to five iterations, if coverage remains insufficient after the fifth iteration, the pipeline proceeds to the next stage and logs the final coverage result.

In this design, coverage analysis not only assesses test-suite completeness but also drives automated test extension. By integrating coverage-guided test suite extender mechanism into the workflow, the pipeline iteratively produces a more thorough and robust suite without manual intervention.

The final step in this pipeline applies mutation testing to evaluate the robustness of the generated test cases.

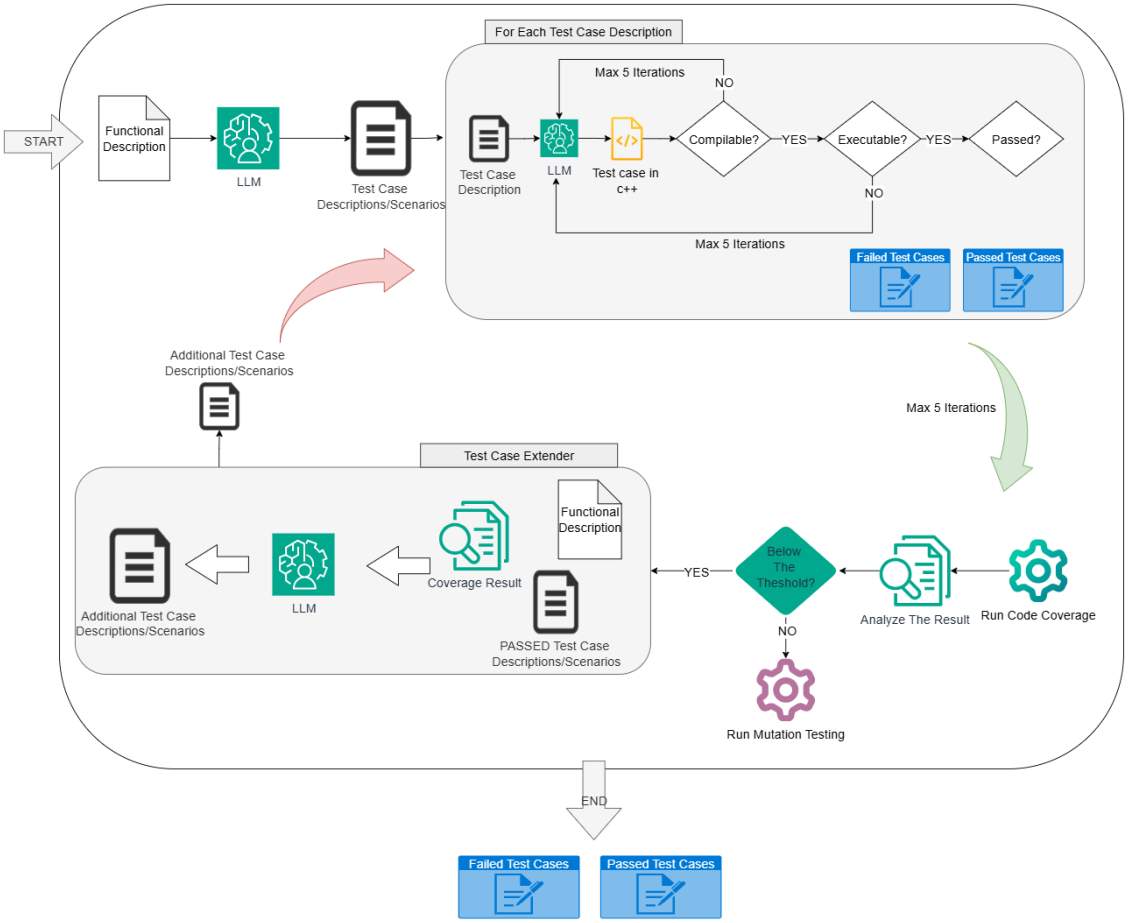


Figure 4.8: The pipeline of the unit test generation stage.

However, to maintain traceability and support future debugging, the pipeline logs failed test cases separately. Specifically:

- Test cases that fail to compile are recorded along with their corresponding error messages.
- Test cases that compile but fail to execute due to runtime errors are logged separately.
- Test cases that compile and execute but fail are saved to a dedicated file for further analysis. further analysis and potential refinement in later stages.

This organized handling of test outputs not only ensures reliability but also facilitates downstream validation and debugging throughout the pipeline.

After the final step of the unit test generation pipeline, and before transitioning to the next stage, all failed test cases are passed to the oracle-based validation agent. This agent independently analyzes each test case to determine whether the failure stems from a defect in the software-in-the-loop or from an issue within the test case itself.

The prompt structure employed across all agents within the unit test generation pipeline is illustrated in Figure 4.9. Each prompt begins by assigning the LLM a clear role aligned with the agent's responsibility. This is followed by a one-shot example, a prompt engineering technique used to guide the LLM by demonstrating the expected input-output pattern. For example, when the agent is tasked with generating C++ test code, the one-shot example consists of a test-case description as input and the corresponding C++ implementation as output. Following the one-shot example, the prompt incorporates system specifications to provide the necessary context. Finally, the prompt concludes with a clear task definition to instruct the model on what to generate.

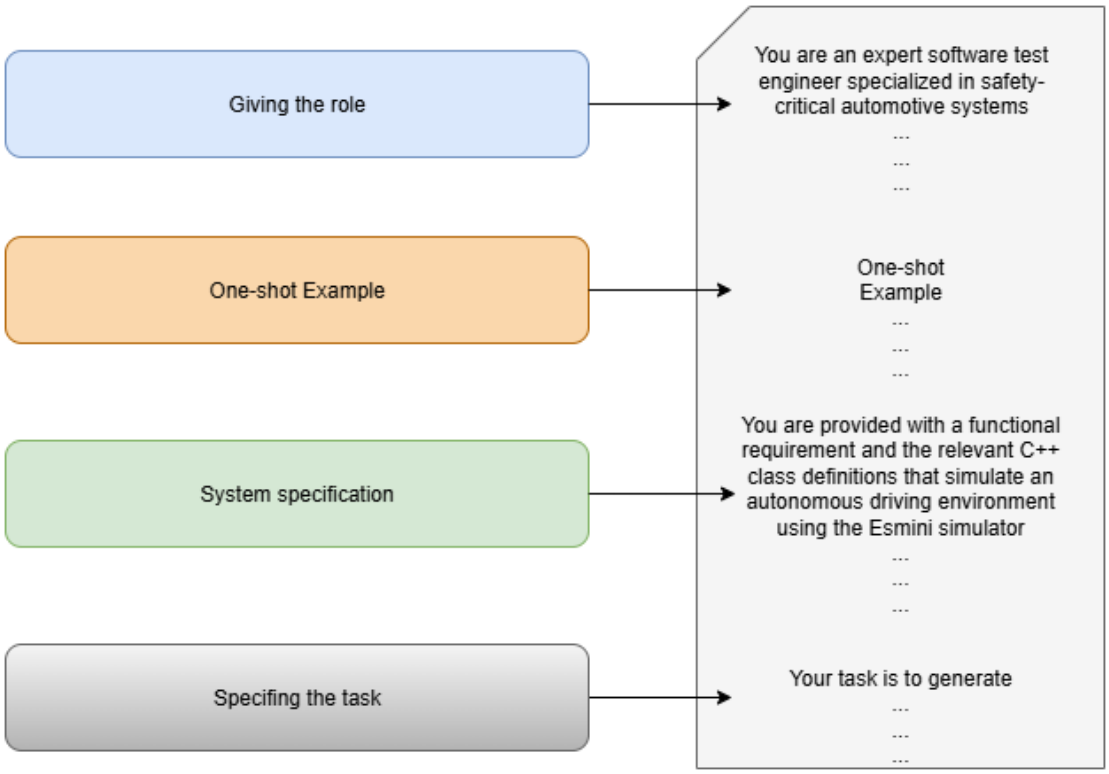


Figure 4.9: The structure of the prompts used in unit test generation stage.

4.8.3 Simulation & Validation

In this stage, Esmini simulation is executed in an automated manner with a selected set of OpenSCENARIO files that are relevant to test the generated custom controller C++ code under different driving conditions. In order to run Esmini within the pipeline, its shared library is utilized, and the generated C++ code acts as a custom controller to manage the vehicle's behavior in the simulation by modifying its internal state. For each scenario executed, we create a CSV file containing detailed timestep-based information about the scenario's objects, which is then analyzed to validate the behavior of the ego vehicle during the simulation. Based on the functional description of each of the four target ADAS functions, in other words, how the generated custom controller is expected to behave in each case, we imple-

ment four corresponding validation functions in our pipeline. These functions rely on real-time data about the internal simulation state, as recorded in the generated CSV file.

4.8.4 Used Tools

LangChain is utilized as the core framework for implementing the pipeline and defining various agents within it. LangChain is a Python library that provides a comprehensive suite of services for developing AI-powered, production-ready applications. In addition, it offers built-in support and specialized modules for designing custom agents and managing communication between them [38].

GCC is also used as a complementary tool within this pipeline to compile the generated C++ code. As discussed previously, the code generation agent is the designated component responsible for managing and executing this compilation process. GCC, which supports the compilation of several programming languages including C++, provides detailed logs upon completion of the compilation. These logs are stored in a dedicated log file and subsequently incorporated into the prompt to refine the generated code in case of any compilation error.

For the code coverage analysis, *Gcovr* is used. It is a Python-based, lightweight tool that can be easily integrated into automated CI pipelines to generate coverage reports efficiently [39]. Another tool used for evaluating the robustness of the test cases is *Mull* [40]. It is a mutation testing framework for C/C++ that injects controlled mutants into the software using *Clang* as the compiler.

4.8.5 Prompting Techniques

At the start of the experimental phase of this study, we used a relatively simple and straightforward prompt to evaluate the LLM's ability to generate source code. The primary objective at this stage was to gain an initial understanding of how effectively the LLM could handle a well-defined task, such as translating implementation code from one programming language to another. Following this preliminary assessment, we gradually refined our prompting strategies to obtain more contextually appropriate and useful outputs.

For example, we adopted *role prompting*, a method in which the LLM is explicitly instructed about the role it is expected to assume in addressing the task. In addition, we used *few shot prompts*, a technique in which a limited number of illustrative examples are provided within the prompt to enable in-context learning [41]. The later technique proved particularly beneficial when the LLM was requested to generate C++ code that is executable within the Esmini simulation environment.

5

Results

5.1 Pipeline Performance and Outputs

This section presents the output and performance results from each iteration for both code generation and unit test generation pipelines. In addition, it evaluates the pipeline after integrating all components, such as code generation stage and unit test generation stage in one single pipeline.

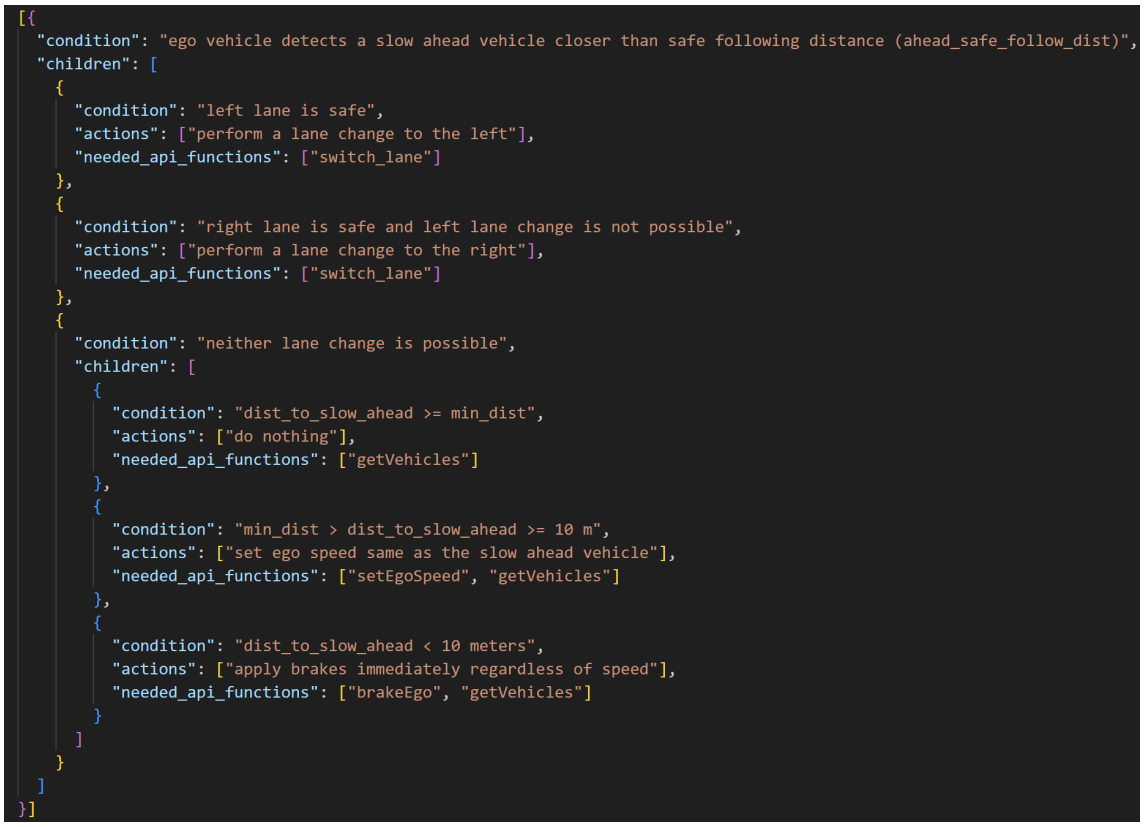
5.1.1 Code Generation

As discussed in 4.8.1, the input functional description is processed and transformed into a JSON object that represents the logical program flow, articulated in natural language. The example shown in 5.2 illustrates the final output generated after two requests to the LLM to construct the targeted JSON object: one request for mapping conditions and their associated actions, and the other for inserting the relevant API functions. The functional description presented in Figure 5.1 was passed as input to the pipeline when generating the JSON shown in Figure 5.2.

The ego vehicle shall during dynamic driving avoid collision with other vehicles as follows:
If ego vehicle detects a slow ahead vehicle closer than safe following distance (`ahead_safe_follow_dist`), the ego vehicle must do all the following:

- If the left lane is safe, perform a lane change to the left.
- Only if the left lane change is not possible and the right lane is safe, perform a lane change to the right.
- If neither lane change is possible, the ego vehicle shall do as follows:
  - `dist_to_slow_ahead` = distance to closest slow ahead vehicle
  - Do nothing if `dist_to_slow_ahead` $\geq$ `min_dist`.
  - Reduce its speed so it is the same as the vehicle ahead if `min_dist` $>$ `dist_to_slow_ahead` $\geq$ 10 m.
  - Apply brakes immediately regardless of speed if `dist_to_slow_ahead` $<$ 10 meters.

Figure 5.1: Functional description designed for F4.



```
[{
  "condition": "ego vehicle detects a slow ahead vehicle closer than safe following distance (ahead_safe_follow_dist)",
  "children": [
    {
      "condition": "left lane is safe",
      "actions": ["perform a lane change to the left"],
      "needed_api_functions": ["switch_lane"]
    },
    {
      "condition": "right lane is safe and left lane change is not possible",
      "actions": ["perform a lane change to the right"],
      "needed_api_functions": ["switch_lane"]
    },
    {
      "condition": "neither lane change is possible",
      "children": [
        {
          "condition": "dist_to_slow_ahead >= min_dist",
          "actions": ["do nothing"],
          "needed_api_functions": ["getVehicles"]
        },
        {
          "condition": "min_dist > dist_to_slow_ahead >= 10 m",
          "actions": ["set ego speed same as the slow ahead vehicle"],
          "needed_api_functions": ["setEgoSpeed", "getVehicles"]
        },
        {
          "condition": "dist_to_slow_ahead < 10 meters",
          "actions": ["apply brakes immediately regardless of speed"],
          "needed_api_functions": ["brakeEgo", "getVehicles"]
        }
      ]
    }
  ]
}]
```


Figure 5.2: An example output of a validated condition-action JSON structure including the suggested API functions generated by qwen2.5-coder for F4. The output follows a correct hierarchical structure, mapping driving conditions to implementable actions in alignment with the designed functional description of F4.

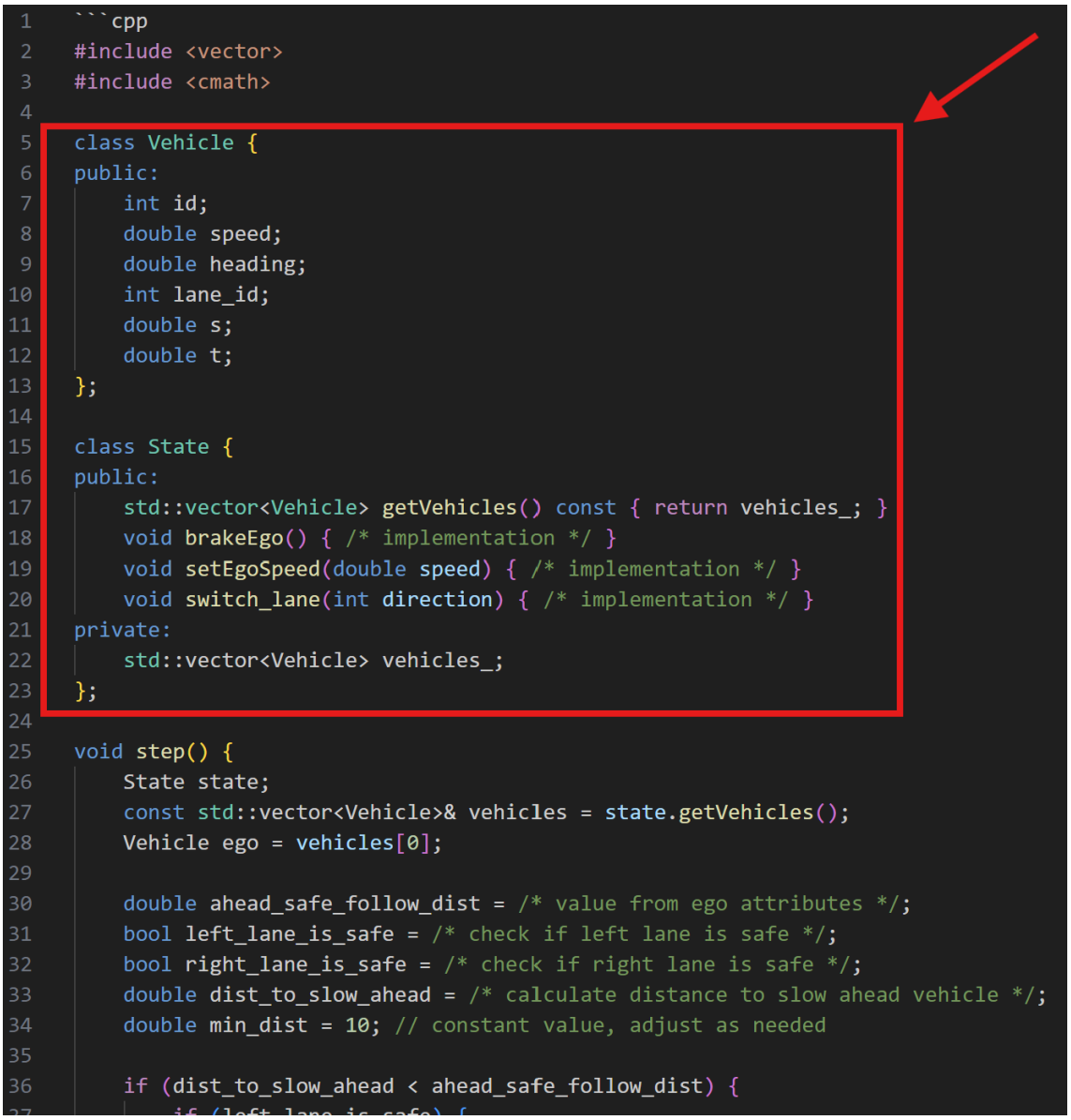
```cpp
#include "State.cpp"
#include <vector>
#include <cmath>
#include <algorithm>

class CustomController {
private:
    State& state;

public:
    // initialize the controller with
    //a reference to the simulation state.
    CustomController(State& state) : state(state) {}

    //Executes one step of the control logic.
    //This method is called at each simulation
    //step to determine the ego vehicle's behavior.

    void step() {

            const std::vector<Vehicle>& vehicles = state.getVehicles();
            Vehicle ego = state.getVehicles()[0];

            bool other_vehicles_present = false;
            for (const auto& vehicle : vehicles) {
                if (vehicle.id != ego.id && vehicle.lane_id == ego.lane_id) {
                    other_vehicles_present = true;
                    break;
                }
            }

            if (other_vehicles_present) {
                state.switch_lane(-1);
            }
    }

};
```

Figure 5.3: An example output of a final custom controller partially generated by gemma3 targeting F2. The code blocks within the green annotated rectangles are produced by the LLM, whereas those within the brown rectangles belong to the existing class template.

When the code for the program flow of each simulation step, corresponding to the `step()` function, is generated, it is subsequently integrated into a C++ class representing the custom controller. Following the successful extraction of the code from the LLM output, the code is inserted into the class template for further processing, as discussed in Section 4.8.1. The examples presented in Figure 5.3 and Figure 5.4 illustrate what parts are generated by the LLM versus those which belong to the class template. However, certain models, such as DeepSeek-coder, tend to generate superfluous code components that are not pertinent to the design of the pipeline. As the pipeline is explicitly equipped with mechanisms that rigorously extract only the `step()` function, ensuring the exact signature, it disregards any additional code fragments included in the output of the model. Figure 5.5 illustrates a representative example of a raw response produced by DeepSeek-coder during code generation.

```cpp
#include "State.cpp"
#include <cmath>
#include <vector>

class CustomController {
private:
    State& state;

public:
    // initialize the controller with
    //a reference to the simulation state.
    CustomController(State& state) : state(state) {}

    //Executes one step of the control logic.
    //This method is called at each simulation
    //step to determine the ego vehicle's behavior.

    void step() {

            const std::vector<Vehicle>& vehicles = state.getVehicles();
            Vehicle ego = state.getVehicles()[0];

            double timeGap = 1.8;
            double cruisingSpeed = 15.0;
            double minDist = 20.0;
            double decelerationRate = 2.0;

            std::vector<Vehicle> leadVehicles;
            for (const auto& vehicle : vehicles) {
                if (vehicle.lane_id == ego.lane_id) {
                    leadVehicles.push_back(vehicle);
                }
            }

            double safeDistance = std::max(timeGap * ego.speed, minDist);
            double closestLeadSpeed = -1.0;
            double closestLeadDistance = -1.0;

            for (const auto& leadVehicle : leadVehicles) {
                double distance = leadVehicle.s - ego.s;
                if (distance > 0.0) {
                    double relativeSpeed = leadVehicle.speed - ego.speed;
                    if (relativeSpeed > 0.0) {
                        if (closestLeadSpeed == -1.0 || distance < closestLeadDistance) {
                            closestLeadSpeed = leadVehicle.speed;
                            closestLeadDistance = distance;
                        }
                    }
                }
            }

            if (closestLeadSpeed == -1.0 || closestLeadDistance >= safeDistance) {
                if (ego.speed < cruisingSpeed) {
                    state.setEgoSpeed(cruisingSpeed);
                } else {
                    state.setEgoSpeed(cruisingSpeed);
                }
            } else {
                if (closestLeadSpeed < ego.speed) {
                    state.brakeEgo();
                } else {
                    state.setEgoSpeed(cruisingSpeed);
                }
            }
    }

};
```

Figure 5.4: An example output of a final custom controller partially generated by gemma3 targeting F3. The code blocks within the green annotated rectangles are produced by the LLM, whereas those within the brown rectangles belong to the existing class template.



```cpp
#include <vector>
#include <cmath>

class Vehicle {
public:
    int id;
    double speed;
    double heading;
    int lane_id;
    double s;
    double t;
};

class State {
public:
    std::vector<Vehicle> getVehicles() const { return vehicles_; }
    void brakeEgo() { /* implementation */ }
    void setEgoSpeed(double speed) { /* implementation */ }
    void switch_lane(int direction) { /* implementation */ }
private:
    std::vector<Vehicle> vehicles_;
};

void step() {
    State state;
    const std::vector<Vehicle>& vehicles = state.getVehicles();
    Vehicle ego = vehicles[0];

    double ahead_safe_follow_dist = /* value from ego attributes */;
    bool left_lane_is_safe = /* check if left lane is safe */;
    bool right_lane_is_safe = /* check if right lane is safe */;
    double dist_to_slow_ahead = /* calculate distance to slow ahead vehicle */;
    double min_dist = 10; // constant value, adjust as needed

    if (dist_to_slow_ahead < ahead_safe_follow_dist) {
```

Figure 5.5: An example of a raw response from DeepSeek-coder when generating code for F4. The code inside the red block represents superfluous code components that were not requested in the prompt.

Regarding error correction, as described in Section 4.8.1 and illustrated in Figure 4.7, compilation error messages are injected as comments into the specific lines of code where the faults arise before the buggy code is submitted to the LLM. Figure 5.6, in particular, provides an example of how these errors are inserted.

```
18
19    void step() {
20
21        const std::vector<Vehicle>& vehicles = state.getVehicles();
22        Vehicle ego = vehicles[0];
23
24        double ahead_safe_follow_dist = std::max(ego.speed * 2.0, 30.0);
25        double min_dist = std::max(ego.speed * 1.0, 20.0);
26        bool slow_ahead_found = false;
27        Vehicle* closest_slow_ahead = nullptr;
28        int lane_change_possible = -1;
29
30        for (const auto& vehicle : vehicles) {
31            if (vehicle.id != ego.id &&
32                vehicle.lane_id == ego.lane_id &&
33                vehicle.s > ego.s &&
34                fabs(vehicle.s - ego.s) < ahead_safe_follow_dist &&
35                vehicle.speed < ego.speed) {
36                slow_ahead_found = true;
37                if (!closest_slow_ahead || fabs(vehicle.s - ego.s) < fabs(closest_slow_ahead->s - ego.s)) {
38                    closest_slow_ahead = &vehicle;  //ERROR IN THIS LINE error: invalid conversion from ‘const Vehicle*’ to ‘Vehicle*’ [-fpermissive]
39                }
40            }
41        }
42
43        if (slow_ahead_found) {
```

Figure 5.6: An example of non-compilable code with an embedded inline error message, provided as part of the error correction prompt.

```
1   #include "State.cpp"
2   #include <cmath>
3   #include <vector>
4
5   class CustomController {
6   private:
7       State& state;
8
9   public:
10      // initialize the controller with
11      //a reference to the simulation state.
12      CustomController(State& state) : state(state) {}
13
14      //Executes one step of the control logic
15      //This method is called at each simulation
16      //step to determine the ego vehicle's behavior.
17      void step() {
18
19              const std::vector<Vehicle>& vehicles = state.getVehicles();
20              Vehicle ego = state.getVehicles()[0];
21
22              if (std::abs(ego.speed - 10.0) <= 1e-6 || ego.speed > 10.0) {
23                  state.brakeEgo();
24              }
25      }
26  };
```

Figure 5.7: An example output of a custom controller after bug fixing based on generated failed test cases using qwen2.5-coder:7b targeting F1. The condition inside the red block represents what is added to the code after bug fixing.

5.1.2 Unit Test Generation

The early iterations produced inaccurate results and faced numerous challenges and limitations. In many cases, the generated test cases referenced non-existent methods. For example, as shown in Figure 5.8, one test case attempted to call the *getEgoSpeed* function, which does not exist in the system. At time the output contained incorrect *include statements* or omitted essential ones which lead to compilation errors. In some instances, particularly in iteration 2 during the feedback refinement loop aimed at fixing test code bugs, the system produced only skeleton code instead of fixing the bug. As illustrated in figure 5.9, the LLM generated a skeleton code with only the structure of the test suit without any test cases. Additionally, there were cases where the LLM generated code that already existed in our system. As showed in Figure 5.10, it unnecessarily generated a mock class that was already part of the system.

```
1  TEST_F(CustomControllerTest, NormalSpeed){
2      double egoInitialSpeed = 5.0
3
4      EXPECT_CALL(mockState, getEgoSpeed()).WillRepeatedly(Return(egoInitialSpeed));
5      EXPECT_CALL(mockState, brakeEgo()).Times(0);
6
7      controller->step();
8  }
```

Figure 5.8: Test case generated with a call to the non-existent getEgoSpeed method.

```
class CustomControllerTest : public ::testing::Test {
protected:
    MockState mockState;
    CustomController* controller;

    void SetUp() override {
        controller = new CustomController(mockState);
    }

    void TearDown() override {
        delete controller;
    }
};

// Main function for running all tests
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
    }
```

Figure 5.9: Generated test case consisting solely of a skeleton structure.

```
class MockState : public State {
public:
    MOCK_METHOD0(fetchStates, void());
    MOCK_METHOD0(getVehicles, std::vector<Vehicle>());
    MOCK_CONST_METHOD0(getEgoSpeed, double());
    MOCK_CONST_METHOD0(getEgoDirection, double());
    MOCK_METHOD0(setEgoSpeed, void(double));
    MOCK_METHOD0(brakeEgo, void());
};

class CustomControllerTest : public ::testing::Test {
protected:
    MockState mockState;
    CustomController* controller;

    void SetUp() override {
        controller = new CustomController(mockState);
    }

    void TearDown() override {
        delete controller;
    }
};
```

Figure 5.10: Output containing code already present in the system.

The final iteration of the unit test generation pipeline demonstrated significant improvements in accuracy, completeness, manageability and reliability. By incorporating structured prompt design, a multi-step approach, iterative regeneration, and coverage-guided extension, the system was able to consistently produce compilable and executable test cases for the simpler functions (F1 and F2), though performance was not as consistent for the more complex ones. In this iteration, the system was able to produce a compilable and executable test suite file, unlike the old approach which never produced a compilable test suite.

Figure 5.11 presents the generated test case descriptions from one run for generating unit test for AEB function that presented in Figure 5.12. Each description outlines a specific functional scenario to be validated. For instance, the first test case checks whether the ego vehicle triggers the brake function when its speed exceeds 10 m/s. Figure 5.13 shows the test suite file which contains all combined passed test cases, for instance, the test case description 1 is translated into C++ test code 1: a vehicle instance is initialized with a speed greater than 10 m/s, and the test verifies whether the brake function is called accordingly.

```
**Test Case 1: Verify Custom Controller Stops Ego Vehicle when Speed Exceeds 10 m/s**
  - **Test Case Description:** The controller should activate the brake function if the ego vehicle's speed exceeds 10 m/s.
  - **Test Case Conditions:**
    - The ego vehicle's initial speed is greater than 10 m/s.
  - **Expected Behavior:** The ego vehicle's speed should decrease to zero, and the brake function should be activated.
  - **TEST-CASE-ENDED**

 **Test Case 2: Verify Custom Controller Does Not Activate Brake When Speed is Below or Equal to 10 m/s**
  - **Test Case Description:** The controller should not activate the brake function if the ego vehicle's speed is below or equal to 10 m/s.
  - **Test Case Conditions:**
    - The ego vehicle's initial speed is less than or equal to 10 m/s.
  - **Expected Behavior:** The ego vehicle's speed should remain the same, and no brake function should be activated.
  - **TEST-CASE-ENDED**

 **Test Case 3: Verify Ego Vehicle Speed Decreases to Zero when Initial Speed is Above 10 m/s**
  - **Test Case Description:** The ego vehicle's speed should decrease to zero if its initial speed is above 10 m/s.
  - **Test Case Conditions:**
    - The ego vehicle's initial speed is greater than 10 m/s.
  - **Expected Behavior:** The ego vehicle's speed should decrease to zero, ensuring the vehicle stops.
  - **TEST-CASE-ENDED**

 **Test Case 4: Verify Custom Controller Handles Zero Speed Correctly**
  - **Test Case Description:** The controller should not activate the brake function if the ego vehicle's speed is initially zero.
  - **Test Case Conditions:**
    - The ego vehicle's initial speed is zero.
  - **Expected Behavior:** No brake function should be activated, and the ego vehicle should continue to drive normally.
  - **TEST-CASE-ENDED**

 **Test Case 5: Verify Custom Controller Handles Negative Speeds Correctly**
  - **Test Case Description:** The controller should not activate the brake function if the ego vehicle's speed is a negative value.
  - **Test Case Conditions:**
    - The ego vehicle's initial speed is a negative value.
  - **Expected Behavior:** No brake function should be activated, and the ego vehicle should continue to drive normally.
  - **TEST-CASE-ENDED**
```

Figure 5.11: Generated test case descriptions for the AEB (F1) function using Mistral.

All test cases in the test suite shown in Figure 5.13 were compilable, executable, and passed their respective assertions, except for Test Case 3, which is commented out by the system with a note indicating a compilation failure. This failure demonstrates an occasional limitation in the LLM's output, where syntactic or structural issues can occur.

```
class CustomController {
private:
    State& state;

public:
    // initialize the controller with a reference to the simulation state.
    CustomController(State& state) : state(state) {}

    //Executes one step of the control logic.
    //This method is called at each simulation step to determine the ego vehicle's behavior.
    void step() {

        const std::vector<Vehicle>& Vehicles = state.getVehicles();

        const Vehicle& ego = Vehicles[0];

        if (ego.speed > 10) {
            state.brakeEgo();
        }
    }
};
```

Figure 5.12: Custom controller for the AEB (F1) function.

```cpp
#include "gtest/gtest.h"
#include "gmock/gmock.h"
#include "CustomController.cpp"

//Test_case1
TEST(CustomControllerTest, VerifyCustomControllerStopsEgoVehicleWhenSpeedExceeds10mps) {
    MockState mockState;

    Vehicle egoVehicle(0, 0.0, 0.0, 0.0, 15.0, 0.0); // ego speed > 10 m/s
    std::vector<Vehicle> vehicles = { egoVehicle };

    EXPECT_CALL(mockState, getVehicles()).WillOnce(::testing::ReturnRef(vehicles));
    EXPECT_CALL(mockState, brakeEgo()).Times(1);

    CustomController controller(mockState);
    controller.step();
}

//Test_case2
TEST(CustomControllerTest, VerifyCustomControllerDoesNotActivateBrakeWhenSpeedIsBelowOrEqual10mps) {
    MockState mockState;

    Vehicle egoVehicle(0, 0.0, 0.0, 0.0, 5.0, 0.0); // initial ego speed <= 10 m/s
    std::vector<Vehicle> vehicles = { egoVehicle };

    EXPECT_CALL(mockState, getVehicles()).WillOnce(::testing::ReturnRef(vehicles));
    EXPECT_CALL(mockState, brakeEgo()).Times(0);

    CustomController controller(mockState);
    controller.step();
}

//Test_case3
// Test case 3 failed to compile.

//Test_case4
TEST(CustomControllerTest, VerifyCustomControllerHandlesZeroSpeedCorrectly) {
    MockState mockState;

    Vehicle egoVehicle(0, 0.0, 0.0, 0.0, 0.0, 0.0); // ego speed = 0 m/s
    std::vector<Vehicle> vehicles = { egoVehicle };

    EXPECT_CALL(mockState, getVehicles()).WillOnce(::testing::ReturnRef(vehicles));
    EXPECT_CALL(mockState, brakeEgo).Times(0); // No brake function should be activated.

    CustomController controller(mockState);
    controller.step();
```

Figure 5.13: Generated passed test cases for AEB (F1) function using Mistral, generated based on test descriptions presented in 5.11.

However, the unit test suite shown above achieved 100% line coverage and branch coverage, indicating a high level of completeness. This output did not need to run the test suit extender component because the result of code coverage anlaysis was perfect and above the thresholds.

As discussed before, when a test fails, it is sent to the oracle-based validation agent to determine whether the issue lies in the test itself or in the software under test. Figure 5.16 shows an example of a failed test case, along with its description, that was submitted to this phase.

In this case, the test was designed to verify the functionality of the AEB function,

specifically when the vehicle speed is exactly 10 m/s. According to the function's description, the vehicle should only brake when its speed exceeds 10 m/s, not when it is exactly 10 m/s. However, the test incorrectly expected braking at exactly 10 m/s. This mismatch in expectations caused the test to fail, indicating that the fault was in the test itself rather than in the software.

Nevertheless, the oracle-based validation agent made the correct decision: it identified the faulty test case as a bug in the testing process and eliminated it.

```
TEST(CustomControllerTest, VerifyCustomControllerStopsEgoVehicleWhenSpeedEquals10mps_TestCase2a) {
    MockState mockState;

    Vehicle egoVehicle(0, 0.0, 0.0, 0.0, 10.0, 0.0, 1, 10.0, 5.0);
    std::vector<Vehicle> vehicles = { egoVehicle };

    ON_CALL(mockState, getVehicles()).WillByDefault(ReturnRef(vehicles));
    EXPECT_CALL(mockState, brakeEgo()).Times(1);

    CustomController controller(mockState);
    controller.step();
}
```

Figure 5.14: One failed test case code.

```
Test Case 2a: Verify Ego Vehicle Break When Speed Exactly Equals 10 m/s
- **Test Case Description:** The ego vehicle should break if its current speed exactly equals 10 m/s.
- **Test Case Conditions:**
    - The ego vehicle's speed is 10 m/s.
- **Expected Behavior:** The controller should activate the brake function.
- **TEST-CASE-ENDED**
```

Figure 5.15: One failed test case description.

Figure 5.16: The unit test case that failed due to a bug in the generated unit test case.

5.2 Evaluation

An empirical evaluation approach is used to evaluate the output of the LLMs in this domain. Thus, five open-source models are being experimented for the purpose of evaluation and these models are as follows:

Table 5.1: Comparison of models by version, parameter count, and size.

Model Name	Version	Number of Parameters	Size
codellama	2.0	7B	3.8 GB
mistral	2.0	7B	4.4 GB
deepseek-coder-v2	1.0	16B	8.9 GB
gemma3	latest	4B	3.3 GB
qwen2.5-coder	2.0	7B	4.7 GB

5.2.1 Code Generation Pipeline

To evaluate the effectiveness and robustness of the code generation pipeline with respect to the designed prompts, comprehensive experiments are conducted in which the entire code generation pipeline is executed. To address various evaluation criteria and metrics, the experiments are divided into three main aspects: the first focused on measuring the correctness of the generated code, the second aimed at examining the frequency and efficiency of the error correction process, and the third assessing whether the LLM overgenerates additional unnecessary components. These three aspects are investigated and limited to the following:

- `Aspect 1` — the accuracy of the generated code. For each ADAS function presented in 4.4. 20 runs are performed in each of the five models. The results are reported separately for each function.

- `Aspect 2` — the number of loop iterations required for error correction to complete a given run. The same 20 runs for each of the ADS/ADAS functions F1 & F2 in each model used in the first category are applied here. The results are reported separately for each function.

- `Aspect 3` — the tendency of LLM to generate superfluous components that it was explicitly instructed to avoid. 20 runs are performed in each model, regardless of the target function.

The experiments for the first aspect of the experiments are limited by four main metrics and they are defined as follows:

- `None` — No code is extracted either because the LLM fails to generate any or because the output format is invalid, for example, only textual output.

- `Non-Compilable` — The generated code must be written in C++, follow the correct structure as defined in the prompt, and be integrable with the class template designed by us. If any of these criteria are not met, the generated code is considered structurally incorrect.

- `Compilable` — The generated code is considered compilable if it compiles successfully within the defined simulation environment, without any syntax or compile-time errors. Therefore, the generated compilation error log file must be empty, and the compiler generates an executable binary.

- `Validatable` — Subsequently, it is assessed whether the code can be executed within the Esmini simulator and can function as a controller for the selected scenarios without causing simulation execution errors. At this stage, the generated code is also evaluated through code review and comparison with corresponding custom controller that we have designed and tested in Esmini. Complete evaluation will be done after generating and running the test cases in the next stage of the pipeline.

The evaluation methodology records the maximum achieved success level per run across the three hierarchical metrics: Non-Compilable, Compilable, and Validatable.

In accordance with error correction, the second part of the experiments shall ultimately include metrics that indicate whether additional error correction iterations were required for the code to reach the desired outcome (i.e., compilable). The maximum number of iterations per run is set to 5; that is, if the code is still not compilable after 5 iterations, the best-performing code variant among those five iterations is reported for the respective run.

- `Without error correction` — No request is sent to the error correction agent, and the current run is completed in the initial iteration. Either because the generated code is already compilable in the initial iteration or because no valid C++ code is extracted in the initial iteration that could be subject to correction.
- `Successful error correction` — At least one error correction iteration is performed (with a maximum of five), and the error correction agent successfully resolves all reported compilation errors.
- `Unsuccessful error correction` — After five iterations, the error correction agent fails to resolve all reported compilation errors or introduces additional compilation errors in the code.

Based on observations obtained from several trial and experimental runs, certain models show a tendency to overgenerate additional components, either in the form of explanatory text or extraneous code. To investigate this, the third aspect of the experiments uses the following metrics:

- `None` — The LLM produces exclusively the expected output format specified in the prompt, without generating any additional content beyond what is requested.
- `Textual` — Textual overgeneration occurs, such as explanations before or after the generated code block.
- `Code Components` — Superfluous code components, such as creation of additional structs or classes.

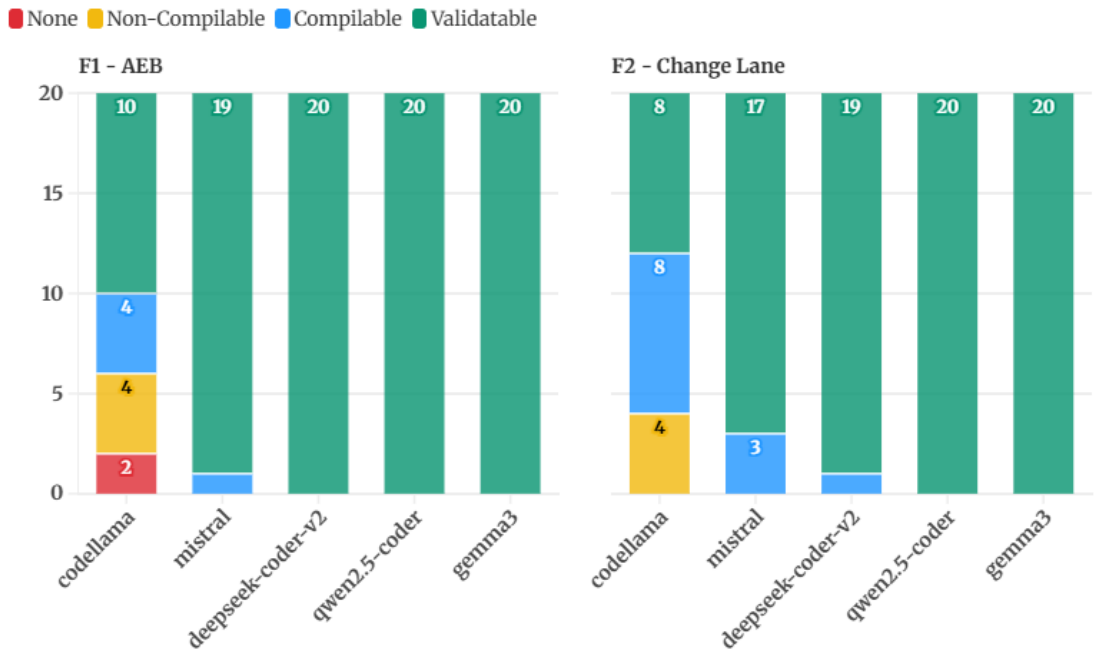


Figure 5.17: Metric-based evaluation outcomes for each of the five LLMs over 20 runs each, specifically for code generation targeting ADAS functions F1 and F2.

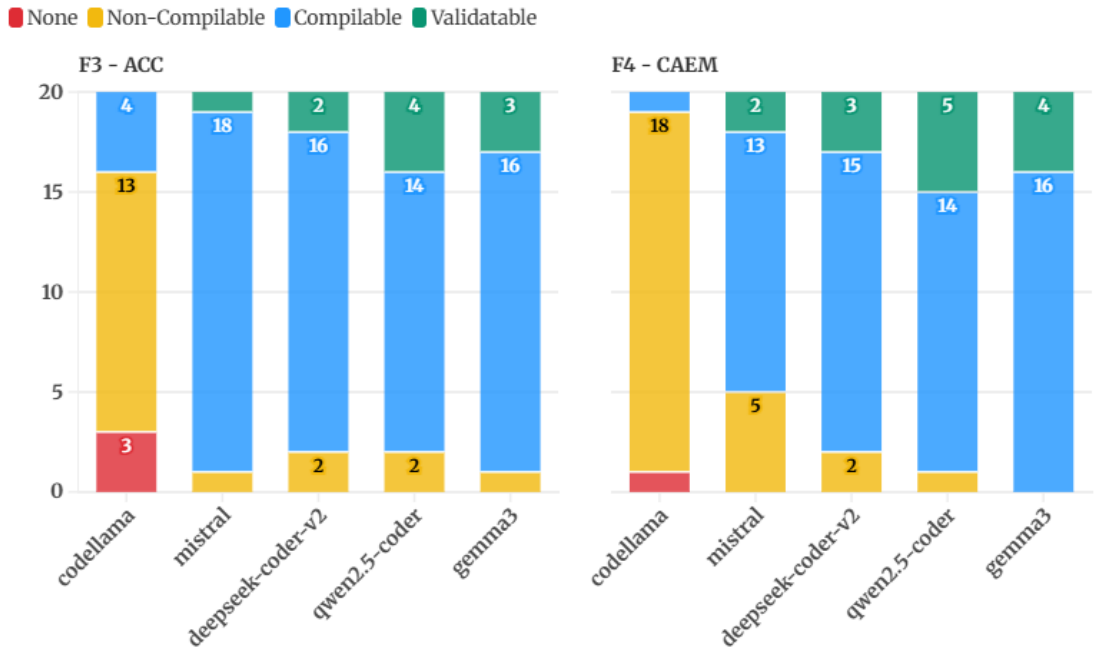


Figure 5.18: Metric-based evaluation outcomes for each of the five LLMs over 20 runs each, specifically for code generation targeting ADAS functions F3 and F4.

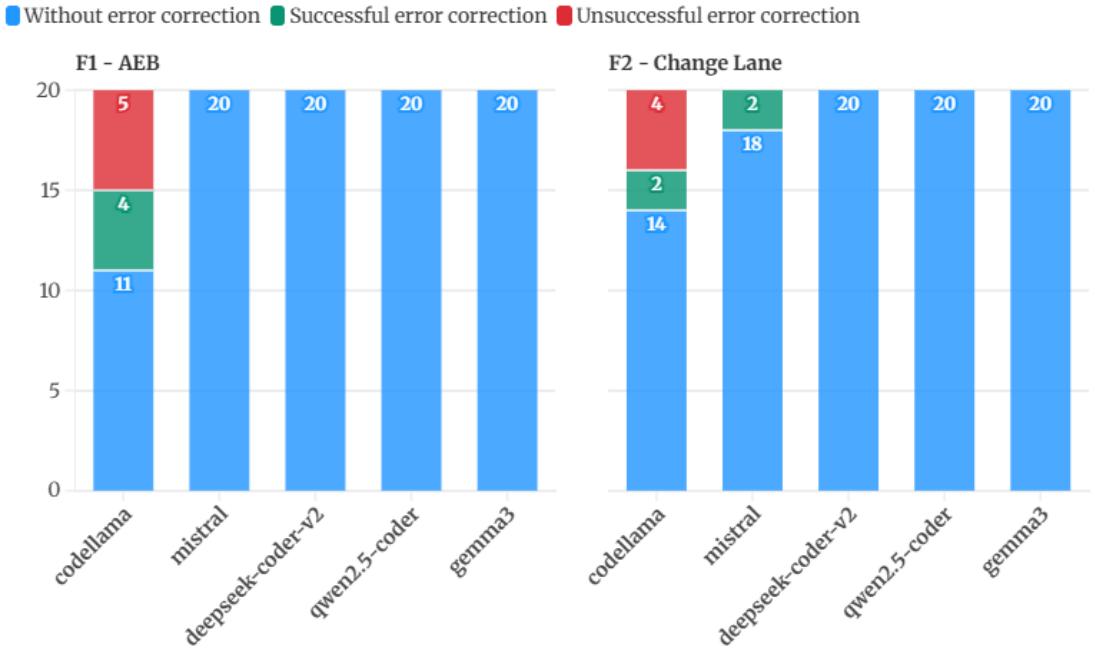


Figure 5.19: Metric-based evaluation outcomes for each of the five LLMs over 20 runs each, specifically for error correction targeting ADAS functions F1 and F2.

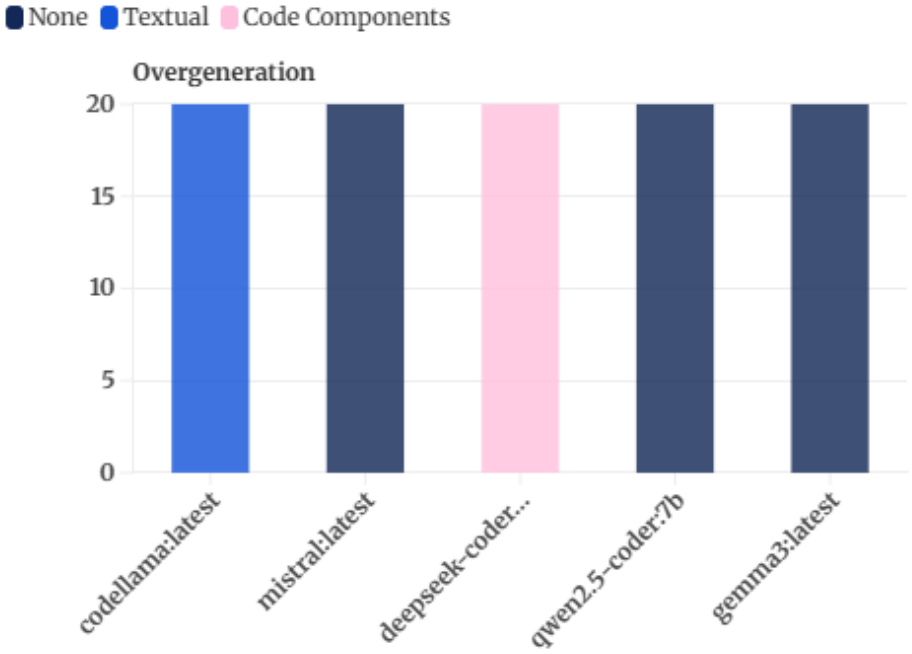


Figure 5.20: The results of overgeneration experiments for 20 runs in each model.

5.2.2 Unit Test Generation Pipeline

To evaluate the pipeline under realistic and varied conditions, a comprehensive experiment was conducted in which the entire unit test generation pipeline was ex-

ecuted a total of 400 times. This involved 20 independent runs for each function, covering a range from simple to complex AD/ADAS functions, across the five different large language models presented in table 5.1, in order to observe how the system behaves with different models. The choice of 20 iterations per function was intended to reduce the likelihood that a correct or high-quality output would occur merely by chance. Since LLMs are stochastic, repeating the runs may help capture results that are more representative of the model's typical behavior rather than isolated lucky cases. Although there is no strict evidence that 20 is the optimal number, similar studies, such as [9], have also employed 20 iterations when examining LLM behavior.

The four functions used in this evaluation were manually written rather than generated by an LLM, as the objective was to assess only the performance of the unit test generation pipeline before integrating it with the main pipeline. During each run, key performance indicators such as line coverage, branch coverage, and mutation scores were recorded. This repeated-measures approach enabled the calculation of average results, analysis of performance variability, and provided a reliable basis for assessing the stability and effectiveness of the proposed pipeline.

Branch and Line coverage metices employed to validate and assess the effectiveness of the generated unit test cases by measuring how thoroughly the test suite exercised the software-in-the-loop. The analysis was conducted using *gcovr*. Additionally, to assess the robustness of the test suite, mutation testing was performed. The mutation score served as a key metric to quantify how effectively the tests can identify deliberately injected changes (mutants) in the software. High coverage and mutation scores together indicate a test suite that is both effective and robust. Table 5.2 shows the used mutation operator for each function in this experiment.

In these experiments, the LLMs generated roughly 6,600 test cases. Given that each test suite was accompanied by textual descriptions, this suggests that a minimum of 7,000 requests were issued to the LLMs. These requests encompassed both the generation of test case descriptions and test suites. This volume of interaction underscores not only the computational effort involved, but also the significant communication overhead between the experimental framework and the models, reflecting the scale and complexity of the testing process. However, coverage thresholds variables were set to maximum during these experiments.

The results are presented through a series of graphs, each comparing the five large language models across the three key performance metrics, line coverage, branch coverage, and mutation score, for each of the four functions. These graphs give an overall view of how the models perform, showing patterns, differences, and strengths in creating effective unit test cases from only functional descriptions. At the end, a table summarizes all the results in one place, making it easy to compare models and metrics side by side.

Table 5.2: Mutation operators applied to the functions during the experiments.

Function	Mutation Operator Name	Operator Semantics
AEB	cxx_gt_to_ge	$> \rightarrow >=$;
	cxx_gt_to_le	$> \rightarrow <=$;
CHANGE_LANE	cxx_ne_to_eq	$!= \rightarrow ==$;
	cxx_eq_to_ne	$== \rightarrow !=$;
ACC	cxx_ne_to_eq	$!= \rightarrow ==$;
	cxx_eq_to_ne	$== \rightarrow !=$;
	cxx_sub_to_add	$- \rightarrow +$;
	cxx_ge_to_lt	$>= \rightarrow <$;
	cxx_le_to_gt	$<= \rightarrow >$;
	cxx_le_to_lt	$<= \rightarrow <$;
	cxx_ge_to_gt	$>= \rightarrow >$;
	cxx_gt_to_le	$> \rightarrow <=$;
	cxx_mul_to_div	$* \rightarrow /$;
	cxx_gt_to_le	$> \rightarrow <=$;
	cxx_gt_to_ge	$> \rightarrow >=$;
CAEM	cxx_ne_to_eq	$!= \rightarrow ==$;
	cxx_eq_to_ne	$== \rightarrow !=$;
	cxx_sub_to_add	$- \rightarrow +$;
	cxx_ge_to_lt	$>= \rightarrow <$;
	cxx_le_to_gt	$<= \rightarrow >$;
	cxx_le_to_lt	$<= \rightarrow <$;
	cxx_ge_to_gt	$>= \rightarrow >$;
	cxx_gt_to_le	$> \rightarrow <=$;
	cxx_mul_to_div	$* \rightarrow /$;
	cxx_gt_to_le	$> \rightarrow <=$;
	cxx_gt_to_ge	$> \rightarrow >=$;
	cxx_lt_to_ge	$< \rightarrow >=$;
	cxx_lt_to_le	$< \rightarrow <=$;

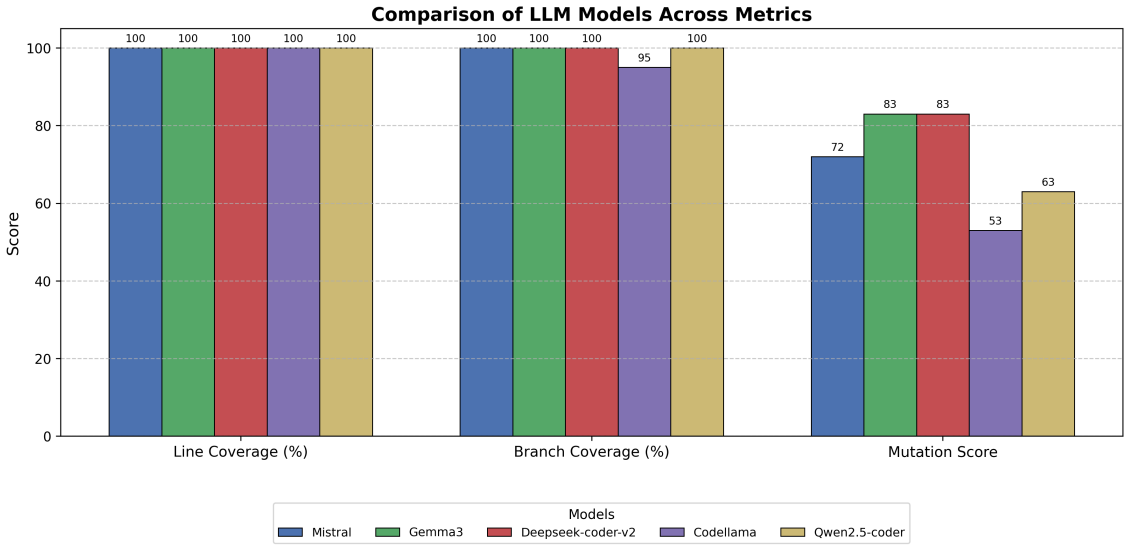


Figure 5.21: Average line coverage, branch coverage, and mutation score achieved by five LLMs on the AEB (F1) function, computed over 20 iterations for each model.

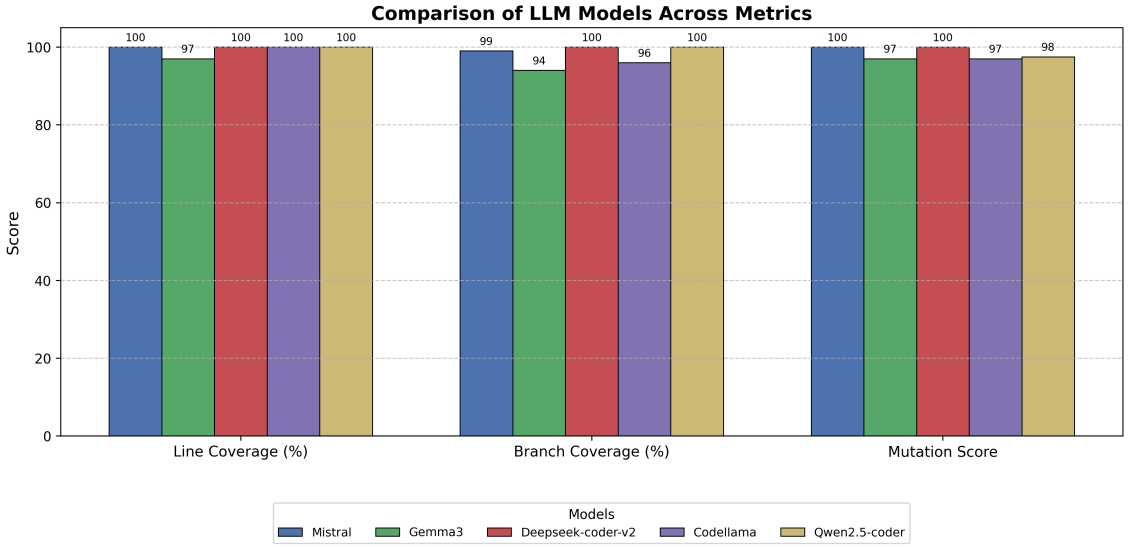


Figure 5.22: Average line coverage, branch coverage, and mutation score achieved by five LLMs on the CHANGE_LANE (F2) function, computed over 20 iterations for each model.

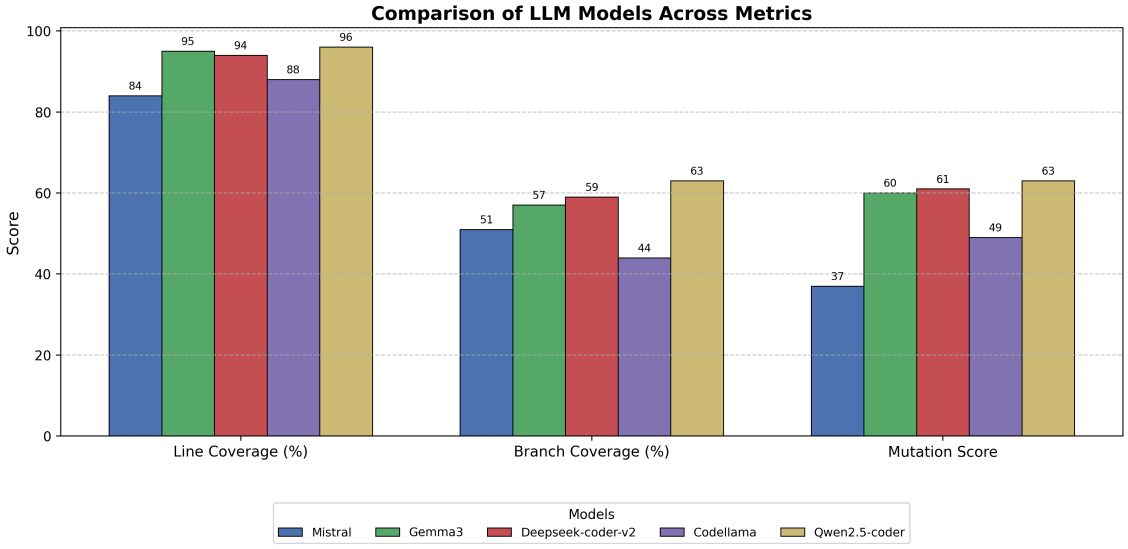


Figure 5.23: Average line coverage, branch coverage, and mutation score achieved by five LLMs on the ACC (F3) function, computed over 20 iterations for each model.

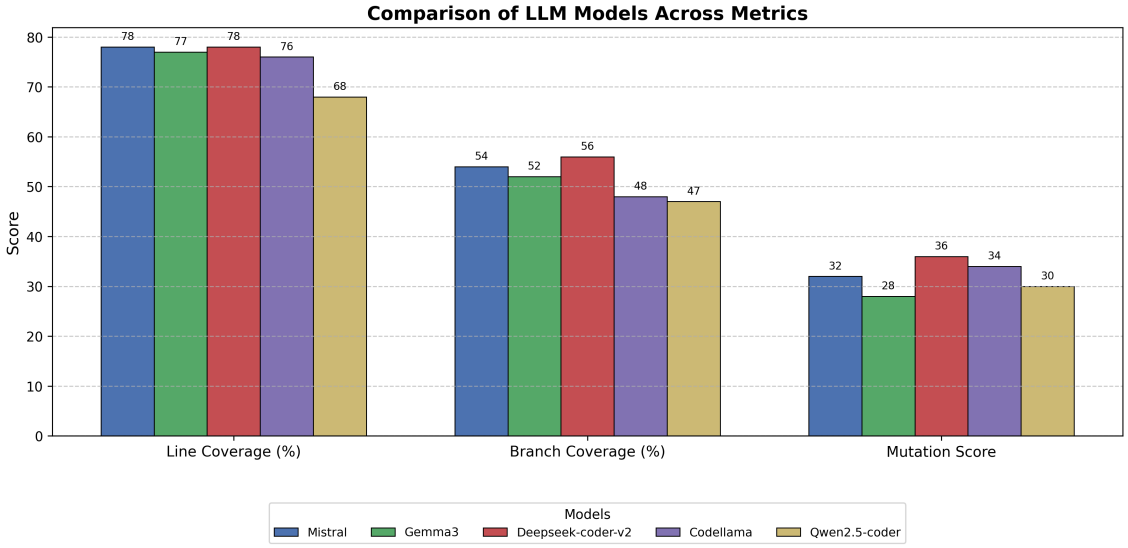


Figure 5.24: Average line coverage, branch coverage, and mutation score achieved by five LLMs on the CAEM (F4) function, computed over 20 iterations for each model.

Table 5.3 contains all the results of all experiments. Each value in the table represents the average outcome across 20 runs for a given function and model.

Table 5.3: Average evaluation metrics over 20 iterations per LLM model across four ADS/ADAS functions, ordered from simplest to most complex.

Function	Model	Line Coverage (%)	Branch Coverage (%)	Mutation Score (%)
AEB	Mistral	100	100	72
	Gemma3	100	100	83
	DeppSeek-coder-v2	100	100	83
	Codellama	100	95	53
	Qwen2.5-coder	100	100	63
Lane Change	Mistral	100	99	100
	Gemma3	97	94	97
	DeppSeek-coder-v2	100	100	100
	Codellama	100	96	97
	Qwen2.5-coder	100	100	98
ACC	Mistral	84	51	37
	Gemma3	95	57	60
	DeppSeek-coder-v2	94	59	63
	Codellama	88	44	49
	Qwen2.5-coder	96	63	63
CAEM	Mistral	78	54	32
	Gemma3	77	52	28
	DeppSeek-coder-v2	78	56	36
	Codellama	76	48	34
	Qwen2.5-coder	68	47	30

Figure 5.25 presents 20 subfigures, each showing data in JSON format extracted from the 20 iterations as a summary. This data includes the average number of total test cases, non-compiled tests, compiled tests, executed, passed and failed tests,

as well as code coverage metrics, mutation scores, and the number of iterations (nr_iteration) the unit test generation pipeline ran. For example, nr_iteration indicates how many times additional test descriptions were generated to improve coverage when line or branch coverage fell below the threshold. Note that, if the number of passed and failed test cases does not equal the total number of compilable test cases, this indicates that some test cases encountered runtime errors. "DC-V2" refers to DeepSeek-Coder-v2, and "QC" to Qwen2.5-Coder.

Figure 5.25: Average summaries over 20 iterations for each Model–Function combination (JSON format).

The data shown in Figure 5.25 are summarized in Table 5.4 for clarity. Each value in the table represents the average outcome across 20 runs for a given function and model.

Table 5.4: Extracted data from Over 20 Iterations per LLM model across four ADS/ADAS functions, ordered from simplest to most complex.

Function	Model	Total generated Tests	Number of Compilable Tests	Passed Tests	Failed Tests	Number of Iterations
AEB	Mistral	6.6	5.7	3.95	1.75	0.2
	Gemma3	7.35	7.35	6.0	1.35	0.0
	DeppSeek-coder-v2	8.7	8.7	5.35	3.35	0.05
	Codellama	9.45	9.4	8.05	1.35	1.0
	Qwen2.5-coder	6.0	5.95	4.55	1.4	0.0
Lane Change	Mistral	7.3	7.1	4.75	2.35	0.45
	Gemma3	15.25	14.15	10.65	3.5	1.55
	DeppSeek-coder-v2	7.05	6.95	6.0	0.95	0.05
	Codellama	11-95	11.9	8.95	2.95	1.3
	Qwen2.5-coder	10.8	10.75	8.4	2.35	0.55
ACC	Mistral	25.0	25.0	5.1	15.55	5.0
	Gemma3	27.85	27.25	14.45	12.7	5.0
	DeppSeek-coder-v2	26.6	26.6	13.3	13.15	5.0
	Codellama	19.6	19.2	8.2	10.8	5.0
	Qwen2.5-coder	42.2	42.2	13.4	28.8	5.0
CAEM	Mistral	18.2	18.2	3.6	12.4	5.0
	Gemma3	30.2	28.8	10.0	18.8	5.0
	DeppSeek-coder-v2	32.4	32.0	10.8	21.2	5.0
	Codellama	17.0	16.4	8.2	8.2	5.0
	Qwen2.5-coder	38.0	37.4	9.2	28.2	5.0

5.2.3 Entire Pipeline

In Sections 5.2.1 and 5.2.2, we present the evaluation and experimental results for code generation and test case generation separately. In this section, the entire pipeline is evaluated as a whole by executing it using the three models that demonstrated the most promising results in the individual sub-pipeline experiments: Gemma3, Qwen2.5-coder, and Deepseek-coder-v2. The pipeline is executed ten times for each ADS/ADAS function per model to account for the inherently non-deterministic behavior of LLMs. The metrics used to evaluate the complete pipeline are as follows:

- `Failed` — The custom controller generated fails in both unit testing and simulation validation.
- `Unit-Testing-Only` — The generated passed test cases cover at least 75% of lines and 40% of branches of the software under test, and achieve a minimum mutation score of 28%.
- `Simulation-Only` - The custom controller generated behaves as expected during the simulation in Esmini, as determined by executing the validation layer presented in Section 4.8.3.
- `Successful` — The custom controller generated passes both unit testing and simulation validation in Esmini.

For each target function, a carefully selected set of embedded scenarios is employed for simulation. The selection is based on whether the scenarios include the relevant road and traffic conditions required to evaluate the logic of the custom controller and to draw meaningful conclusions from the simulation. In particular, the scenarios used must satisfy the following aspects:

- **For F1:** The ego vehicle must reach a speed greater than 10 m/s at some point during the simulation.
- **For F2:** There must be at least one additional vehicle or object (e.g., a pedestrian) in the same lane as the ego vehicle, and at least one adjacent lane to the right must be available.
- **For F3:** There must be at least one slower vehicle ahead in the same lane as the ego vehicle.
- **For F4:** There must be multiple lanes, with at least one to the left and one to the right, and at least one additional vehicle besides the ego vehicle, preferably several.

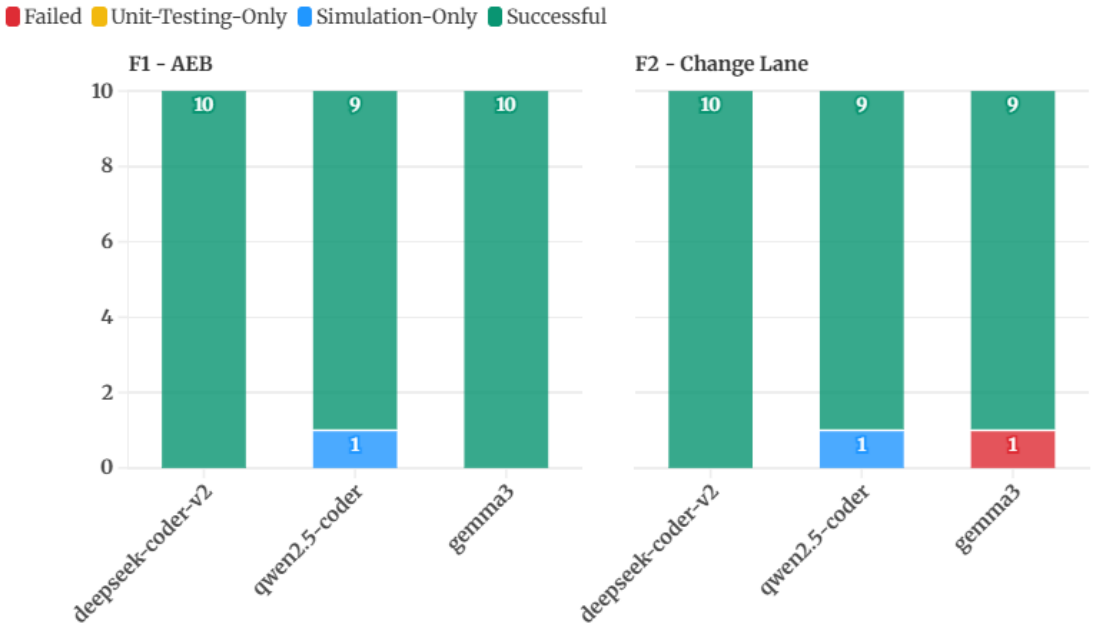


Figure 5.26: Metric-based evaluation results for each of the three LLMs over 10 runs each, specifically for the entire pipeline targeting F1 and F2.

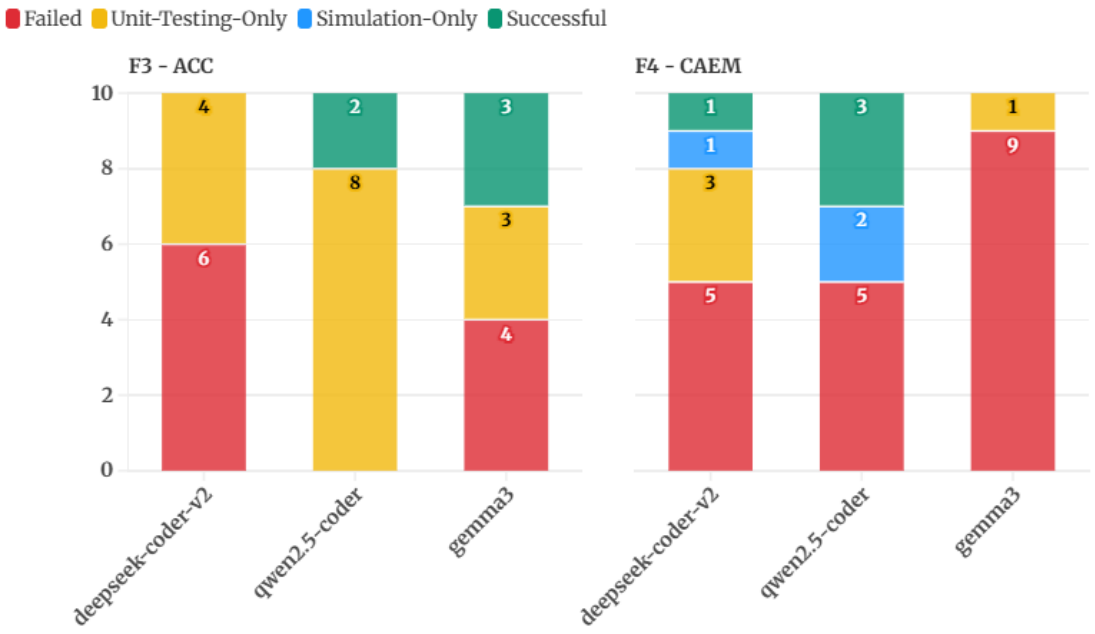


Figure 5.27: Metric-based evaluation results for each of the three LLMs over 10 runs each, specifically for the entire pipeline targeting F3 and F4.

6

Discussion

This chapter interprets the experimental results in light of the research questions, highlighting both the strengths and weaknesses of the proposed multi-agent LLM pipeline for ADS/ADAS software development. The discussion also reflects on the broader implications of this approach, compares findings with related work, and outlines areas for improvement and future research. The study's research questions are addressed in the following sub sections: subsection 1 covers RQ1, subsection 2 discusses RQ3, and subsection 3 addresses RQ2.

6.1 Architectural Modularity in Domain-Specific LLM Systems

As mentioned in Section 4.7, generating C++ source code from prototype-level code proved to be more efficient than the method later adopted in the second development iteration, namely generating code directly from raw, unprocessed functional descriptions. Our observations suggest that a significant reason for this difference is that the former approach contributed to the LLM making fewer assumptions regarding both the structural organization of the code and the availability of API functions for updating or accessing the simulation state. When converting code from one programming language to another, the prototyping code already provided a complete representation of the logical program flow, leaving the LLM primarily responsible for reimplementing the code while accounting for syntactic and language-specific distinctions.

The initial approach employed during the first development iteration served as a reference framework for code generation, despite being tested only on two relatively simple ADS/ADAS functions, F1 and F2 in this case. Hence, the aim was to generate the final implementation from an intermediate representation that explicitly reflects the logical program flow, analogous to the prototyping code. At the same time, this representation incorporates explicit suggestion regarding which API functions are relevant, thereby reducing the likelihood that the LLM introduces unwarranted assumptions about the availability of non-existent functions.

As our aim was to employ open-source models to generate a fully functioning custom controller for each targeted AD/ADAS function, the pipeline was designed in a

manner that would maximize the effectiveness of these models. In addition to the functional decomposition of the problem, which is relatively self-evident in a domain of this nature, we sought to reduce the amount of source code produced by the LLM, omitting the common class template used across all controllers and generating only the `step()` function, as it varies between controllers. In accordance with this approach, the output of the LLM is evaluated based on the fact that it contains a `step()` function with the exact required signature and that its logic is consistent with the functional description of the function in focus.

Drawing on observations made during numerous experimental and trial runs, together with the unequivocal numerical outcomes from the experiments done for overgeneration, it is important to highlight the output produced by certain models. It is both evident and essential to acknowledge that the model cannot be assumed to generate solely the components it was instructed to produce, nor to conform precisely to the output format explicitly specified in the prompt. As evidenced by the experimental results, particularly those presented in Figure 5.20, CodeLlama frequently produces textual explanations both preceding and following the requested code, despite being explicitly instructed in the code generation prompt to avoid such output. In the context of overgeneration, DeepSeek-Coder-v2 has likewise been observed to produce superfluous components, although in the form of code. An illustrative example, as shown in Figure 5.5, is the redefinition of new structs or classes for *State* and *Vehicle*. If such code were to be adopted in its entirety, it would reference these overgenerated State and Vehicle constructs rather than the actual simulation state, which, in turn, would result in a non-functioning implementation.

When it comes to the unit test generation, the results indicate that the proposed pipeline can consistently produce compilable and functional test suites to validate the targeted AD/ADAS functions. As illustrated in Figure 5.25, the number of compilable test cases closely matches the total number of generated test cases across all experiments, demonstrating a very high compilation success rate. This outcome suggests that the modularity and decomposition strategies have significantly enhanced the performance of the unit test generation pipeline. Following the introduction of modularity, the pipeline has become more reliable compared to earlier implementations. The generated test cases are now easier to manage and debug, which facilitates guiding the LLM or system to produce compilable tests. The observed compilation success rates provide strong evidence of this improvement. Furthermore, the application of the separation of concerns principle ensures that each test case is independent, further contributing to the ease of debugging and maintainability of the test suites.

In the final integrated pipeline, the results in Figure 5.26 highlight the strength of the CoTeGen tool, showing its ability to consistently deliver working and validated software components for AD/ADAS simple functions. The high success rate reflects the generation of components that behave as expected, confirmed through both unit tests and Esmini validation. For the complex functions, the pipeline achieved only a limited success rate in generating components that behave as expected, see Figure 5.27. Integrating code generation and unit test generation within a single

fully automated pipeline is inherently challenging, as each stage is susceptible to failure, and both rely on LLM-based outputs, meaning an error in one stage can directly compromise the other. The tool's ability to manage this challenge can be attributed to the modularity of its components and the system as a whole, as discussed earlier. This success is also supported by the applied prompt engineering techniques, including few-shot examples, carefully crafted context, and structured prompts that guide the LLMs toward more reliable outputs. In practice, modularity and prompt design worked in parallel, complementing each other to enhance the system's reliability.

These findings suggest that limitations in LLM based code generation are not solely due to model capabilities but also depend on how tasks are formulated and integrated within the system. A well-structured, modular pipeline can achieve strong results even with less resource-intensive models, whereas poor design can limit performance regardless of model scale. In this sense, successful automation with LLMs depends as much on smart system design and engineering as on the models' abilities. The experiment results from both code generation and unit test generation clearly support this point. The same principle applies to prompt design, which plays a critical role in guiding the models toward relevant, reliable, and context-aware outputs. The good designed prompts not only reduces the likelihood of irrelevant or hallucinated outputs but also ensures that the generated code and tests aligned with the intended function description and system specification.

6.2 Effectiveness of The Pipeline

The results presented in Figures 5.17, 5.26 indicate that the majority of open-source models used are capable of generating correct and functioning code for F1 and F2 within a pipeline specifically designed for this purpose. With regard to F3 and F4, see Figures 5.18 and 5.27, all models, except Codellama, are capable of generating compilable code. However, they are considerably less effective in producing functionally correct implementations. This limitation arises primarily from the substantial increase in logical complexity compared to F1 and F2.

Analogously, producing compilable and functional tests does not automatically guarantee that they effectively validate the targeted AD/ADAS functions. The results of evaluating effectiveness and robustness show that the pipeline performs very well for simple functions F1-F2, where the test cases can thoroughly cover expected behaviors and detect potential faults. When it comes to more complex F3-F4, the pipeline still generates usable tests, but the efficiency and robustness are more moderate. This is largely due to the increased number of conditions, branches, and continuous variables, which makes it more challenging for the system to anticipate all critical scenarios.

Although the moderate performance of the complex functions, the results are encouraging given the inherent difficulty of the task. The software under test operates in a simulation-based environment, which adds complexity to unit testing because

parts of the simulation state must be mocked. Lower coverage was expected, as the pipeline relies solely on natural language function descriptions without access to implementation details or full knowledge of all execution paths. This contrasts with prior studies discussed in Section 3.3, which leveraged implementation details to guide test generation.

Despite these challenges, the generated unit tests are valuable for early-stage validation or bug detection, particularly for simple functions. For more complex functions, while the tests may not capture every scenario, they still provide a meaningful level of validation and help identify some issues.

6.3 Limitations of Using LLMs for Automated AD-S/ADAS Software Development

For those models employed in code generation, certain recurring limitations are observed even among those demonstrating promising performance, such as Qwen2.5-Coder, Gemma3, and DeepSeek-Coder. One of the recurring limitations observed in a considerable number of runs is that the generated code occasionally omits certain conditions when the logical program flow increases in complexity, particularly when involving several nested conditions. This issue is especially evident in functions F3 and, to a greater extent, F4. For instance, consider a case where it is necessary to check two conditions, *condition1* and *condition2*, independently for each lane in order to determine whether the lane is safe for a lane change. In some generated code examples, the check for *condition1* is applied to the right lane while *condition2* is applied to the left lane, even though both *condition1* and *condition2* must be met for each lane individually. Such code cannot be considered validatable, as it permits lane changes in unsafe situations due to these omitted condition checks.

Another limitation arises when an error correction iteration is required because the initially generated code fails to compile. In general, most models exhibit significant difficulty in performing effective error correction. Certain models, such as CodeLlama, demonstrate limited capability in rectifying the non-compilable code provided to them, even when accompanied by detailed error logs. Naturally, the success of this process also depends on the number of errors and the extent to which they are non-trivial in nature.

Despite the strengths of the unit test generation pipeline, several limitations and issues have been identified. Occasionally, the system produces false-positive or buggy test cases that fail due to problems in the test itself rather than in the software under test. This can be a significant problem, as such tests could interfere with already functional custom controllers. To address this, we introduced an oracle-based validation agent, which mitigates the impact of these faulty tests.

We observed also that the system generates out-of-scope test cases. For example, when testing braking behavior at speeds above a threshold, it may attempt to validate unrelated functions, such as setting the vehicle's speed to move the ego vehicle

after stopping, which falls outside the intended function description and cause test failures. As shown in Figure 5.25 and Table 5.4, the average number of failed test cases per run ranges from 1–3 for simple functions and 8–28 for complex functions. The higher failure rate in complex functions is often due to multiple iterations aimed at filling coverage gaps, which can cause the LLMs to hallucinate additional tests.

In addition, we observed redundancy in the generated test cases, with some tests checking the same behavior or even having identical names, leading to redefinition errors. Furthermore, after a few iterations, the pipeline often fails to produce new tests that meaningfully increase line or branch coverage. For complex functions, this is evident from Figure 5.25 and Table 5.4, where the pipeline ran five iterations to extend the test suit to improve the coverage, yet coverage plateaued nearly as we observed after the second iteration.

Despite the positive results of the entire integrated pipeline experiments, one significant issue was observed during several iterations. The oracle-based validation agent does not always eliminate all false-positive test cases, which can prompt unnecessary changes to an already functional custom controller. Due to time constraints, we were unable to investigate in detail why the oracle-based validation agent failed to filter out all the false positives. Nevertheless, the system consistently selects the best-performing controller from all generated candidates. Code changes made solely to address false positives are not chosen unless they pass all existing tests, ensuring that the final outputs remain correct and reliable. This reduces the negative impact of this limitation, though further research is needed to improve the effectiveness of oracle-based validation.

6.4 Extensibility

In this study, our focus was limited to the four functions F1-F4, and the system currently does not handle other functions. However, the system is extensible and could support additional functions, provided that new components are added. For example, a validation function for checking the generated function in Esmini, and any necessary API functions for communication with Esmini.

It should be noted that the four validation functions in the pipeline discussed in Section 4.8.3 have been implemented by us and are not generated by any agent within our pipeline. In the event that this pipeline is extended in the future, for example to incorporate support for additional AD/ADAS functionalities, this module of the pipeline will accordingly require modification.

6.5 Threats to Validity

A potential threat to validity arises from the observation that a measurable subset of the generated test cases occasionally fails, even though the corresponding controller code is correctly implemented and demonstrates the expected behavior in simulation.

This suggests that the failing test cases may be too irrelevant or outside the intended scope of verification.

Another limitation concerns the interpretation of coverage metrics. Line and branch coverage do not necessarily guarantee that the generated controller will behave as intended during system-level testing in simulation. This limitation becomes particularly evident for the more complex functions, such as F3 and F4, where logical correctness cannot be fully captured by coverage measures alone.

Since the study focuses only on the ADS/ADAS, the solution may not be generalized to other software-intensive domains. This limits the ability to apply the finding to other types of safety-critical systems. Furthermore, while the study considers various pipeline agents for automation, it does not aim to explore all potential AI-based agents, focusing instead on LLMs as the central component of automation. Although cybersecurity and safety are critical aspects of ADS/ADAS, this study does not address these concerns or threats related to the pipeline's operation, as its primary focus is on functional and performance evaluation. These delimitations ensure a focused and manageable scope while directly addressing the core research questions.

6.6 Future Work

Although the proposed pipeline demonstrates relatively promising results in generating code, producing test cases, and validating them through simulation with the support of open-source LLMs, it remains platform-dependent, as it is currently limited to execution within the Esmini environment. A natural direction for future research is to investigate the applicability of such a solution in more advanced simulation frameworks, such as Carla.

In addition, one of the most immediate priorities is improving the oracle-based validation mechanism. Although it successfully reduces many false-positive test cases, some still remain. This is a significant issue since it could affect an already working custom controller. Exploring more advanced and reliable approaches to eliminate all false positives test cases or enhancing the unit test generation process to avoid producing such cases in the first place, represents an important direction for further research.

Another potential suggestion, in line with the aim of maximizing automation, is to extend the pipeline to generate OpenSCENARIO files that include custom traffic and road conditions. This would enable the generated custom controller to be tested against a broader range of scenarios, rather than relying entirely on predefined scenarios.

6.7 Conclusion

The results indicate that the proposed design solution of multi agent LLMs can deliver compilable software artifacts that can also be executed and validated with respect to the domain simulation environment. In this case, the software artifacts are comprised of source code and unit test code, both implemented in C++.

The iterative methodology used in this study helps refine parts of the initial pipeline design or even redesign the entire pipeline based on the bottlenecks observed. This applies not only to software modules that regulate data flow in the system, but also to the prompts and functional descriptions used to generate software artifacts for each target AD/ADAS function.

As expected, the performance of the CoTeGen tool varies across different functions. It achieves a high success rate for simpler functions, while its effectiveness is more moderate for complex ones. At the current stage, relying solely on the tool to generate fully functional code without any human involvement, such as code review, is not yet feasible. Nevertheless, CoTeGen can provide significant value as a starting point for developers, offering initial code and tests that can be further refined and validated.

Bibliography

[1] Tanay Varshney. Introduction to llm agents. https://developer.nvidia.com/blog/introduction-to-llm-agents/, 2023. Accessed: 2025-03-26.

[2] J. He, C. Treude, and D. Lo. Llm-based multi-agent systems for software engineering: Vision and the road ahead. *Journal of Software Engineering*, 1(1):1–22, Oct 2024.

[3] A. Nouri. Accelerating the design phase: Towards devsafeops for autonomous driving software. *Chalmers Digitaltryck*, 2024.

[4] D. Thakur, A. Mehra, R. Choudhary, and M. Sarker. Generative ai in software engineering: Revolutionizing test case generation and validation techniques. *IRE Journals*, 7(5), Nov. 2023.

[5] J.-L. Colaco, B. Pagano, and M. Pouzet. Scade 6: A formal language for embedded critical software development. In *TASE 2017 - 11th International Symposium on Theoretical Aspects of Software Engineering*, pages 1–10, Nice, France, September 2017.

[6] M. Liu, J. Wang, T. Lin, Q. Ma, Z. Fang, and Y. Wu. An empirical study of the code generation of safety-critical software using llms. *Applied Sciences*, 14(3):1046, 2024.

[7] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[8] Tharindu Godage, Sivaraj Nimishan, Shanmuganathan Vasanthapriyan, Vigneshwaran Palanisamy, Charles Joseph, and Selvarajah Thuseethan. Evaluating the effectiveness of large language models in automated unit test generation. In *2025 5th International Conference on Advanced Research in Computing (ICARC)*, pages 1–6, 2025.

[9] Ali Nouri, Beatriz Cabrero-Daniel, Zhennan Fei, Krishna Ronanki, Håkan Sivencrona, and Christian Berger. Large language models in code co-generation for safe autonomous vehicles, 2025. Available: https://arxiv.org/abs/2505.19658. Accessed: July 02, 2025.

[10] Ali Nouri, Christian Berger, and Fredrik Törner. On stpa for distributed development of safe autonomous driving: An interview study. In *Proceedings of the 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 5–12, Durres, Albania, March 2023. Available: https://arxiv.org/abs/2403.09509v1.

[11] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques. *arXiv preprint arXiv:2406.06608*, 2024.

[12] Manas Deb and Tokunbo Ogunfunmi. Information-theoretical analysis of a transformer-based generative ai model. *Entropy*, 27(6), 2025. Available: https://www.mdpi.com/1099-4300/27/6/589. Accessed: May 11, 2025.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

[14] Shubham Vatsal and Harsh Dubey. A survey of prompt engineering methods in large language models for different nlp tasks. *arXiv preprint arXiv:2407.12994*, 2024.

[15] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhi-Yuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *arXiv:2308.11432v7 [cs.AI]*, Mar 2025. Accessed: 2025-03-02.

[16] Christoforos Kachris. A survey on hardware accelerators for large language models. *arXiv preprint arXiv:2401.09890*, 2024. Available: https://arxiv.org/abs/2401.09890. Accessed: June 03, 2025.

[17] Boonyawee Sirimaya. 5 ways ai agents are making work easier. https://www.amitysolutions.com/blog/5-ways-ai-agents-make-work-easier?utm_source=chatgpt.com, 2024. Accessed: 2025-03-26.

[18] Lijun Sun, Yijun Yang, Qiqi Duan, Yuhui Shi, Chaol Yu, Yu-Cheng Chang, Chin-Teng Lin, and Yang Shen. Multi-agent coordination across diverse applications: A survey. *arXiv preprint*, 2502.14743v2, February 2025.

[19] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI*

Conference on Human Factors in Computing Systems, CHI '21. ACM, 2021.

[20] Esmini. Esmini. `https://github.com/esmini/`, 2021. Accessed: 2025-03-29.

[21] Esmini. Esmini official website. `https://esmini.github.io/`, 2025. Accessed: March 30, 2025.

[22] E. Knabe. Controllers in esmini. `https://github.com/esmini/esmini/blob/master/docs/Controllers.md`, 2020. Accessed: March 30, 2025.

[23] Esmini. Logging - esmini official documentation, 2025. Accessed: July 10, 2025.

[24] Han Wang, Sijia Yu, Chunyang Chen, Burak Turhan, and Xiaodong Zhu. Beyond accuracy: An empirical study on unit testing in open-source deep learning projects. *arXiv preprint arXiv:2402.16546*, 2024. Available: `https://arxiv.org/abs/2402.16546`. Accessed: August 15, 2025.

[25] Yunpeng Zhong, Kang Pei, Bo Li, Qi Li, and Nidhi Kalra. A survey on scenario-based testing for automated driving systems in high-fidelity simulation, 2021. Available: `https://arxiv.org/abs/2112.00964`. Accessed: July 02, 2025.

[26] Zhennan Fei, Mikael Andersson, and Andreas Tingberg. Correlation of software-in-the-loop simulation with physical testing for autonomous driving, 2024. License: CC BY-NC-SA 4.0. Available: `https://arxiv.org/abs/2406.03040`. Accessed: May 17, 2025.

[27] Ali Nouri, Håkan Sivencrona, Beatriz Cabrero-Daniel, Christian Berger, and Fredrik Törner. Engineering safety requirements for autonomous driving with large language models. In *Proceedings of the 32nd IEEE International Requirements Engineering Conference (RE)*, Iceland, 2024. Available: `https://arxiv.org/abs/2403.16289`.

[28] Ali Nouri, Christian Berger, and Fredrik Törner. An industrial experience report about challenges from continuous monitoring, improvement, and deployment for autonomous driving features. In *Proceedings of the 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 358–365, Gran Canaria, Spain, 2022.

[29] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *Journal of the ACM (J. ACM)*, 37(4):1–70, August 2018.

[30] Emanuele Arteca, Samuel Harner, Michael Pradel, and Frank Tip. Nessie: Automatically testing javascript apis with asynchronous callbacks. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1494–1505, Pittsburgh, PA, USA, May 25–27 2022. ACM. Available: `https://doi.org/10.1145/3510003.3510106`.

[31] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *arXiv preprint*, 2302.06527, December 2023. Available: `https://arxiv.org/abs/2302.06527`.

[32] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Ajay A. Sawant, and Vladimir J. Hellendoorn. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533, Adelaide, Australia, September 28–October 2 2020. IEEE. Available: `https://doi.org/10.1109/ICSME46990.2020.00056`.

[33] Fabio Palomba, Davide Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST@ICSE)*, pages 5–14, Austin, Texas, USA, May 14–22 2016. ACM. Available: `https://doi.org/10.1145/2897010.2897016`.

[34] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. A system for automated unit test generation using large language models and assessment of generated test suites. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 29–36, 2025.

[35] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[36] R. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, January 2014.

[37] Google. Googletest: Google testing and mocking framework, 2025. Accessed: 2025-08-17.

[38] LangChain. Langchain documentation, 2025. Available: `https://python.langchain.com/docs/introduction/`. Accessed: April 4, 2025.

[39] Lukas Atkinson and Michael Förderer. Gcovr: A utility for managing the use of the gnu gcov utility and generating summarized code coverage results, 2025. Accessed: 2025-08-16.

[40] Alex Denisov and Stanislav Pankevich. Mull: Practical mutation testing and fault injection for c and c++, 2025. Accessed: 2025-08-16.

[41] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. Available: `https://arxiv.org/abs/2005.14165`. Accessed: August 15, 2025.