

Generating Automotive Code: Large Language Models for Software Development and Verification in Safety-Critical Systems

Sven Kirchner¹, Alois C. Knoll¹

Abstract—Developing safety-critical automotive software presents significant challenges due to increasing system complexity and strict regulatory demands. This paper proposes a novel framework integrating Generative Artificial Intelligence (GenAI) into the Software Development Lifecycle (SDLC). The framework uses Large Language Models (LLMs) to automate code generation in languages such as C++, incorporating safety-focused practices such as static verification, test-driven development and iterative refinement. A feedback-driven pipeline ensures the integration of test, simulation and verification for compliance with safety standards. The framework is validated through the development of an Adaptive Cruise Control (ACC) system. Comparative benchmarking of LLMs ensures optimal model selection for accuracy and reliability. Results demonstrate that the framework enables automatic code generation while ensuring compliance with safety-critical requirements, systematically integrating GenAI into automotive software engineering. This work advances the use of AI in safety-critical domains, bridging the gap between state-of-the-art generative models and real-world safety requirements.

I. INTRODUCTION

Recent advancements in the automotive domain are driving a paradigm shift from hardware-defined to software-defined intelligent vehicles, where software complexity and safety-criticality have increased significantly. Traditional linear processes, such as the V-model or the waterfall model [1], offer limited flexibility to adapt to dynamically evolving requirements. As the volume of automotive software grows, each change in requirements requires extensive changes to the code base and repeated validation cycles, increasing both development time and cost. As a result, software architects and developers face increasing challenges in ensuring safety compliance, especially given the continuous expansion of regulatory frameworks and standards, which are now reaching levels of complexity that are difficult to track and implement manually [2]. Large Language Models (LLMs), such as Chat GPT-3 [3], have recently shown promise in addressing this complexity by transforming the role of developers from code authors to orchestrators of generative pipelines. Instead of writing all application-level software manually, engineers can leverage LLMs for automated code generation, using the same standards that once hindered rapid development as structured data sources for compliance.

In this work, we propose a novel framework that integrates Generative Artificial Intelligence (GenAI) into the software

development lifecycle (SDLC). By using LLMs in conjunction with test-driven development (TDD) and static analysis, our approach enables modular system architectures that can be rapidly adapted to evolving requirements, while ensuring compliance with critical safety standards. A core element of the proposed framework is its focus on software generation during the test and integration phases, making the LLM an active participant in the iterative refinement loop. Under this paradigm, test suites and integration scripts assist the LLM by guiding automated code generation to meet specified requirements. The automated process reduces the need for manual recoding and retesting when system requirements change. Our methodology therefore shifts engineering effort to the creation of specification artefacts and robust tools, rather than traditional manual coding. We detail how this framework improves development speed by minimizing human intervention in code production and compliance checks and we illustrate its capabilities with an automotive case study that demonstrates its ability to save time and reduce error rates. We therefore present the following **contributions**:

- **GenAI-Integrated SDLC:** A novel LLM-driven development cycle that combines TDD, static analysis and iterative refinement for safety-critical automotive software.
- **Safety Monitoring Pipeline:** A unified framework for static analysis, formal verification, and automated integration validation to ensure safety compliance in automotive software systems.
- **LLM Handling:** Evaluating the benchmark performance of LLMs and optimizing their implementation for automated code generation and refinement in automotive software development.
- **ISO-based ACC Case Study:** Validation through automated generation of an ISO-based ACC system in C++, tested on the CARLA simulator [4].

II. RELATED WORK

The fundamental principles of software engineering are based on critical design decisions in software development. To integrate safety as a priority, a focus on robust design capabilities can be complemented by effective test and validation methodologies and the use of software verification tools. This enables GenAI to create scalable and safety-critical applications while ensuring compliance with automotive standards.

¹ Technical University of Munich, Garching, Bayern, Germany

This research was funded by the Federal Ministry of Education and Research of Germany (BMBF) as part of the CeCaS project, FKZ: 16ME0800K.

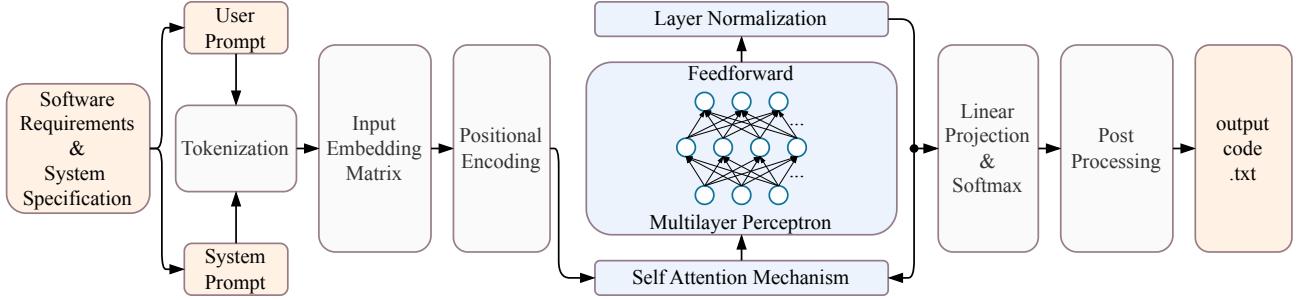


Fig. 1: Introduction of Attention Mechanism and Transformer Architecture in Large Language Models for Code Generation.

A. Reducing software complexity through design choices

Managing complexity in software engineering requires a systematic approach to design choices, aimed at constraining degrees of freedom and thereby reducing the likelihood of errors. Consequently, maintaining logical consistency in algorithms and flow control becomes essential for verifying correctness [5]. Structured interactions between the system and its environment reduce ambiguity and improve integration [6]. Efficient organization of data underpins performance and scalability [7], while modular architectures ensure extensibility and adaptability [8]. Test-driven development integrates correctness into the development process [9], while computational precision mitigates numerical instability [10]. Dependency management and platform compatibility ensure consistent behaviour across environments [11]. Addressing security vulnerabilities is essential for demonstrating system safety [12], while effectively managing concurrency is critical for handling dynamic and parallel systems [13].

Design principles and standards provide a formal framework for managing the inherent degrees of freedom in software development, ultimately enabling the creation of robust, scalable and extensible systems [8]. Strategically controlling each degree of freedom minimizes the potential for error, thereby increasing the overall safety and reliability of the software.

B. Safety-Critical Software

Safety-critical software requires a systematic approach, treating programming as an exact science with predictable and provable behaviour under all conditions [14]. Achieving this requires careful selection of programming languages, robust compiler validation and comprehensive verification and testing methodologies.

C++ is widely used in safety-critical design because of its balance of high performance, precise memory control, and deterministic resource management, which enables strict real-time and reliability constraints to be met. Compiler validation further strengthens these guarantees by translating the code correctly. C++ compilers such as GCC and Clang are highly optimized for performance and reliability, offering advanced static analysis, code optimization and diagnostics [15]. Beyond language and compiler choice, static code analysis is essential to ensure logical consistency

[5]. Tools such as cppcheck for C++ [16] identify memory leaks, race conditions and guideline violations, significantly reducing the likelihood of unexpected behaviors. Before full integration, system behaviour is validated through unit testing. Frameworks like Google Test [17] verify functional correctness by adapting testing preconditions and edge cases. By integrating language safety, certified compilers, static analysis and rigorous testing, this approach improves correctness, compliance with safety standards and robustness in safety-critical applications.

C. Foundations of Requirements Engineering and Software Development for Automotive Systems

Automotive software development bridges complex safety and functional requirements with robust code design. A Software Requirements Specification (SRS) defines both functional (e.g., system behaviour) and non-functional (e.g., performance, security) requirements, guiding the development process. Stakeholder alignment is achieved through use cases, user stories and prototypes, ensuring clarity and addressing safety from the outset. Safety and quality standards are central. ISO 26262 [18] defines functional safety requirements for road vehicles, while Automotive Software Process Improvement and Capability Determination (ASPICE) [19] provides a framework for assessing software quality and processes. The MISRA guidelines [20] standardize programming practices, often implemented in C or C++ for their reliability and performance in safety-critical systems. Detailed testing, including static analysis and unit testing, ensures compliance with safety standards. General requirements align with frameworks like ISO 26262, while function-specific requirements address unique system needs.

By integrating these principles, automotive software development transforms complex requirements into reliable, maintainable and safety-compliant systems.

D. Integrating Generative AI into Software Engineering

The introduction of the Attention Mechanism revolutionized natural language processing, enabling the development of Large Language Models (LLMs) through the innovative Transformer Architecture. This architecture facilitates non-sequential data processing, overcoming the limitations of earlier recurrent models and introducing greater efficiency and scalability [21].

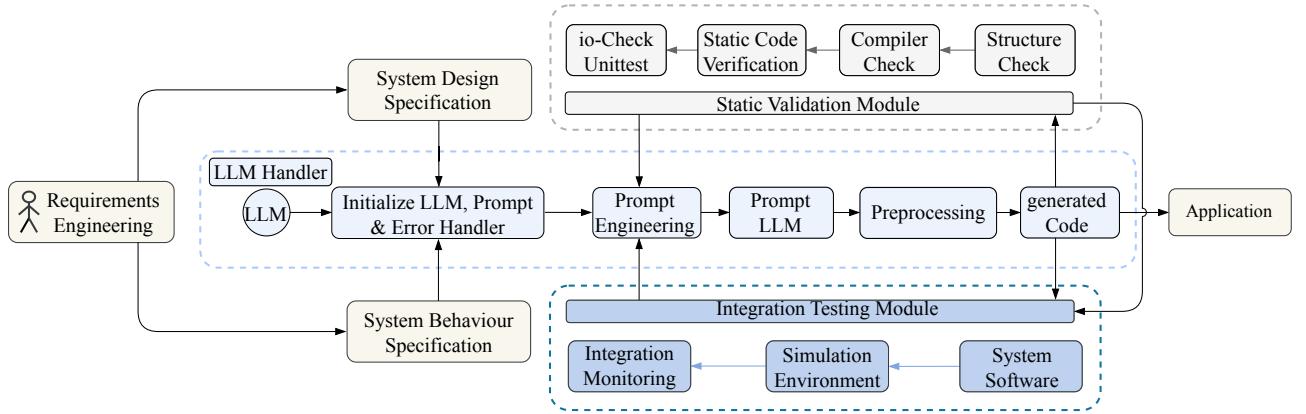


Fig. 2: The code generation architecture consists of three components: the LLM handler (light blue), the static validation module (grey) and the integration testing module (dark blue). The user (light yellow) provides specifications and the LLM Handler generates executable code that is iteratively refined by static checks and integration tests to ensure safety and functionality.

In the framework illustrated in Figure 1, a given requirement provided as textual input is tokenized and processed through input embeddings, where it is enriched with positional encodings to preserve the sequential context. The tokens are then iteratively passed through a stack of self-attention layers and feedforward multilayer perceptrons. These mechanisms ensure that the model captures both local and global dependencies within the input requirements. Subsequently, the processed tokens undergo linear projection and post-processing steps to generate the next token in the sequence. This iterative process enables the generation of software in the form of coherent and contextually relevant text. The ability to train or fine-tune foundation models like llama3 [22] on specific tasks has been further enhanced by leveraging large-scale datasets. For instance, LLMs such as those designed for code generation like Qwen2.5-Coder [23] benefit significantly from task-specific training datasets. The performance of LLMs is heavily influenced by the quality of the training data, the design of the prompt and the system specifications.

An essential aspect of the effective use of LLMs is the management of the context size, which is inherently limited by the architecture. To maximize the utility of the available context, effective prompting techniques have been developed, ensuring that critical information is succinctly presented within the limited input space. Strategies such as Zero-shot or Few-shot Prompting [24], Chain-of-Thought [25] and Role-based Prompting [26] enable LLMs to generate accurate and high quality output for a variety of tasks. Combination with further validation and formal verification methods can improve the overall quality of the generated code [27].

III. METHOD

To integrate generative AI into the software development cycle, we propose an approach that combines test-driven development with previously introduced verification meth-

ods and static code analysis, ensuring safety through rapid feedback and consistency.

A. Architectural Design and Software Version Control

The framework illustrated in Figure 2 shows the GenAI integrated SDLC. User input, including detailed specifications, serves as the foundation. The LLM Handler transforms this input into executable code and iteratively refines it based on feedback from subsequent phases. The Static Validation Module analyzes the generated code for compliance with safety standards and design principles. Finally, the Integration Testing Module evaluates the system in a dynamic test environment, ensuring robust performance and functional correctness.

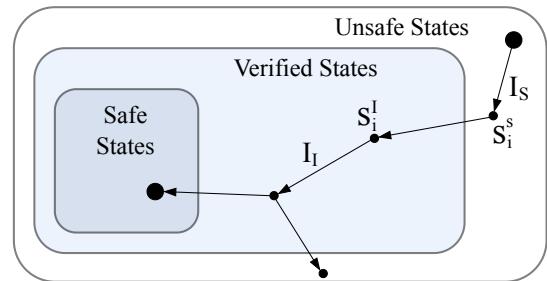


Fig. 3: Software versions are categorized by safety classification, progressing through three primary states. The "Verified State" is achieved upon successful completion of static testing. "Safe States" are achieved after meeting static and integration test criteria. The LLM Handler dynamically manages the generation of LLMs based on the current state of the software.

The use of automated code refinement and regeneration is highly dependent on the software's current development stage and version. In safety-critical systems, version control adheres to a methodology combining integration monitoring,

static code analysis and iterative refinement (Figure 3). The ultimate objective is to achieve a "Safe State," wherein all predefined functional and safety requirements are satisfied.

Each iteration I_S involves static analysis managed by the Static Validation module to detect and resolve logical inconsistencies and design violations. Upon successful completion, the code moves from a static state (s_i^S) to an integration state (s_i^I) within a simulation environment where iterative validation and refinement occurs (I_I). Transition to a safe state is only realized when both static and integration criteria are conclusively met.

This cyclical process of static and integration iterations ensures continuous improvement of the software. Each subsequent state builds incrementally upon its predecessor, anchored in validated safety and integration protocols. Importantly, a new verified state only generated when the integration phase is successful and all static checks are resolved, ensuring an uncompromising commitment to safety and correctness.

B. LLM based generation: Specification and User Input

User input is given in two distinct classes: System Design Specification and System Behaviour Specification. The System Design Specification focuses on the inputs and outputs of the system, using precise mathematical language to define algorithmic preconditions and postconditions. The System Behaviour Specification describes the overall structure and behaviour of the system. For prompting LLMs, structured text is provided as input. JSON is a widely used in software engineering due to its standardization and compatibility. YAML's human-readable features, such as whitespace structuring, optional quotes and support for inline comments, enhance usability and interpretability during the specification phase [28]. To maximize prompt efficiency, JSON is used for the system design specification, while YAML is used for the system behaviour specification. This dual-format strategy ensures effective context management and optimal use of prompt space to generate accurate and interpretable output.

The framework integrates zero-shot and few-shot prompting to optimize LLM performance. Zero-shot prompting establishes baseline output, followed by iterative few-shot refinement to improve accuracy and resolve errors. This process transitions from exploratory prompts to precise adjustments based on output quality. Chain-of-Thought reasoning improves version control by providing the LLM with the best prior solution, enabling iterative improvement towards a safe state, as shown in Figure 3. Role-based prompting defines the LLM's role as a "specialized AI assistant for safety-critical automotive code generation", ensuring outputs are tailored to the framework's requirements. Each iteration builds on previous results, driving continuous improvement and alignment with specifications.

Selecting an appropriate LLM is critical to achieving optimal results. We use McEval: Massively Multilingual Code Evaluation [29], a benchmark designed to evaluate LLM performance across 40 programming languages, including Rust and C++, using 16,000 test cases. This provides a

comprehensive framework for evaluating multilingual code generation. For reasoning and iterative code refinement, we adopt Aider's code editing benchmark [30]. It evaluates precision and consistency in modifying functions, implementing missing functionality and refactoring code from natural language instructions. By prioritizing editing over generation, the Aider benchmark evaluates the accuracy and consistency of code review and refinement in a variety of programming challenges, making it critical for frameworks that require robust iterative development capabilities. We also include the widely recognized HumanEval benchmark [31]. Table I summarizes the performance of various large language models across these benchmarks, highlighting their multilingual code generation and iterative refinement capabilities. We evaluate state-of-the-art open-source LLMs, including Qwen2.5-Coder-7B-Instruct, Llama-3-8B-Instruct, DeepSeek-Coder-V2 Lite Instruct [32], DeepSeek-Coder 33B Instruct [33], and CodeStral-22B [34], using GPT-4o as a benchmark for comparison.

LLM Code Generation and Refactoring Benchmark			
Model	Aider CodeEditing	McEval	HumanEval (0-shot)
Qwen2.5-Coder-7B-I	57.9	60.3	88.4
Llama-3-8B-Instruct	37.6	32.0	62.2
DeepSeek-Coder-V2 LI	48.9	54.7	81.1
DeepSeek-Coder 33B I	49.6	54.3	79.3
CodeStral-22B	48.1	50.5	78.1
GPT-4o (240513)	54.0	72.9	90.2

TABLE I: Evaluation of state-of-the-art large language models, with all values reported in percentages. McEval results represent Pass@1 performance, while HumanEval scores reflect 0-shot capabilities. DeepSeek-Coder-V2 LI denotes the Lite Instruct variant, DeepSeek-Coder 33B I refers to the Instruct version, and Qwen2.5-Coder-7B-I indicates the Instruct variant.

Given the sensitivity of the data and the stringent data protection requirements in the automotive domain, our use case necessitates a locally deployable LLM capable of handling complex programming tasks without compromising data security. Among models with a token size of less than 10B, Qwen2.5-Coder-7B-Instruct proves as the most effective option, offering excellent performance while being the smallest in this category.

Once the LLM has generated the output, the next step is to extract the code and install the necessary libraries. This phase includes dependency management, ensuring all necessary libraries and tools are properly defined and integrated into the software environment. By automating these tasks, the framework accelerates the development cycle while maintaining precision and reliability. This structured approach to LLM-based generation bridges the gap between user-provided input and actionable software artefacts.

C. Static Validation Module

Once the code has been extracted from the LLM analysis, the first step in the evaluation process is static code analysis.

This phase involves a series of checks to ensure the safety, functionality and adherence to design specifications of the generated code. Static code analysis plays a critical role in identifying potential problems early on, providing a strong foundation for seamless integration into the main framework. To advance to the next state (S) and proceed to integration testing, the generated software (SW) must successfully pass a series of static checks (c_i). The process shown in 1 involves an iterative cycle of software generation and refinement, where each iteration systematically resolves errors (E) identified during the previous analysis. Through this approach, the software progressively achieves compliance with the required safety and functional standards, ensuring readiness for the integration phase.

Algorithm 1 Static Analysis and Error Handling

```

1: while  $\exists s_i^s \in S$  such that  $s_i^s \neq \text{success}$  do
2:    $SW \leftarrow \text{generate\_code}(s_i^s, E)$ 
3:   for  $c_i \in C$  do
4:      $s_i^s \leftarrow \text{analyze}(SW, c_i)$ 
5:     if  $s_i^s == \text{success}$  then
6:       continue
7:     else
8:        $E \leftarrow \text{get\_error\_analysis}(SW, c_i)$ 
9:       break
10:    end if
11:   end for
12: end while
13:  $S \leftarrow \text{run\_integration\_monitoring}(SW)$ 

```

The static analysis process consists of several key checks: It starts with a structure check, which verifies that the function name of the primary functionality is in the appropriate place in the generated code. This ensures compatibility with the broader framework and establishes a baseline for integration.

The compilation check in C++ ensures that the code can be successfully compiled by validating it. The compiler performs multiple checks, including syntactic verification to enforce correct grammar and semantic analysis to ensure meaningful execution. Modern compilers employ sophisticated multi-layered validation mechanisms to detect type inconsistencies, scope violations and improper memory usage. This process guarantees both syntactic and semantic correctness, reinforcing code reliability and robustness in high-performance computing environments.

Next, the static code style and design check enforces adherence to established formatting and design principles, ensuring maintainability and consistency. Tools such as cpcheck for C++ validate compliance with standards such as MISRA to address stylistic and semantic issues.

Finally, unit testing validates the functional behaviour of the code. Frameworks such as Google Test in C++ are used to run comprehensive test suites, covering diverse scenarios to confirm that inputs and outputs align with specified requirements. These tests serve as a safety net against regressions

and ensure correctness in future iterations. When using unit testing as feedback, it is crucial to withhold specific failure details from the LLM, preventing over-fitting to individual tests and preserving generalizable behaviour.

Taken together, these steps constitute a rigorous and systematic approach to static analysis, ensuring that the code meets the highest standards of safety, compatibility and reliability, while laying a robust foundation for subsequent development and integration.

D. Integration Monitoring Module

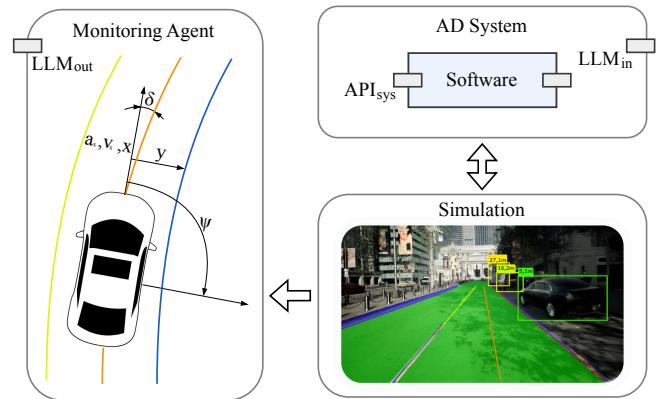


Fig. 4: The integration monitoring framework consists of three key components: the monitoring agent, which acts as a client to the simulation server and has access to ground truth data; the simulation server, which provides synthetic sensor and vehicle state information; and the AD system, which provides an autonomous driving system that allows to integrate the generated software through clear APIs.

After successfully passing the static analysis phase and demonstrating full compliance with static safety requirements, the generated software is integrated into the Autonomous Driving (AD) system. This integration is achieved through the use of well-defined and strictly verified Application Programming Interfaces (APIs), which have been validated during the static analysis phase for consistency with the previously verified software components. These validated System APIs (API_{sys}) ensure a seamless and robust connection between the generated software and the broader AD system architecture. The AD system, which receives both vehicle state information and sensor data from a simulation environment, consists of the three core components of autonomous driving systems: detection, planning and control. For the detection module, YOLOP [35] is used, enabling simultaneous semantic segmentation and object detection, providing essential contextual information. The CARLA simulator serves as the simulation environment, generating synthetic sensor data, such as camera feeds, as shown in Figure 4. To enrich the object detection with depth information, a depth camera is integrated to provide distance measurements relative to the ego vehicle.

This data, along with other vehicle states obtained from the inertial measurement unit (IMU), is transmitted to the AD system. The integration monitoring framework includes a monitoring agent (see figure 4) that uses a simple API to inspect the environment and system behaviour through its access to the ground truth data from the simulation server. User-provided system behaviour specifications serve as a fundamental element in this framework. These specifications articulate simple mathematical functions designed to validate vehicle behaviour against pre-defined performance criteria. The integration process follows a structured timeline that includes three distinct phases: initialization, data processing and evaluation. During the initialization phase, multi-processing is used to simultaneously start the simulation environment, the monitoring agent, and the automated driving (AD) system, ensuring that the performance of the generated function remains unaffected. Once operational, the integration monitoring system performs three critical functions: `get_data_from_carla_server`, `process_carla_data()` and `add_data_to_statistics()`. At the end of a predefined integration test duration, the `evaluate_data()` function is triggered to perform the mathematical validations defined in the system behaviour specifications. These validations span a spectrum of criteria, ranging from behavioural and comfort-related metrics to strict timing requirements. Upon successful completion of this stage, the generated software, refined through iterative interactions with the LLM, achieves the robustness and reliability required for seamless integration, meeting all safety-critical and performance criteria.

IV. EVALUATION

To evaluate the proposed framework, we generate a test function designed to demonstrate its effectiveness in a realistic scenario. Specifically, we select the adaptive cruise control (ACC) system, a widely studied application in the automotive domain. The ACC system offers robust evaluation capabilities in simulation and aligns with the stringent ISO standards, providing a benchmark for the performance and reliability of our framework.

A. Simulation Environment and Hardware Setup

The primary objective of the ACC system is to control the longitudinal motion of a vehicle. As a critical component of the control subsystem in autonomous driving systems, the ACC takes as input the bounding box of the lead vehicle including distance information and the inertial measurement unit (IMU) data of the ego vehicle. Using these inputs, the system calculates the necessary longitudinal motion and generates throttle and brake commands as outputs. These interfaces are essential for maintaining the correct vehicle speed and spacing in dynamic driving conditions. To ensure seamless integration within the integration system, the AD system includes additional modules required for vehicle operation. These include object detection and segmentation to identify surrounding objects, as well as motion planning

and a lateral control module to ensure coordinated vehicle maneuvers. The simulation environment replicates real-world driving scenarios, providing a controlled yet dynamic setting to evaluate the ACC function. The system is tested under various conditions to validate its robustness, efficiency and compliance with ISO requirements. The hardware setup for this evaluation shown in Table II is critical to achieve high-performance execution and accurate real-time simulation.

Hardware Setup for the Generation and Testing Modules

CPU	AMD Ryzen 9 9950X (16x 4.3 GHz, 170W)
GPU	2 x NVIDIA GeForce RTX 4090 24GB
RAM	64GB DDR5-5600 Vengeance (2x 32GB)

TABLE II: To accelerate the development process, we leverage a high-performance hardware configuration to run the LLM locally and execute the algorithm within the simulation environment faster than real-time, ensuring efficient experimentation and iterative refinement.

To enhance computational efficiency, we use accelerated inference techniques to reduce latency and enable faster processing. Memory management optimizations are also implemented to minimize gaps in GPU memory allocation. By dynamically growing memory segments, the framework ensures efficient utilization of VRAM, reducing fragmentation and improving stability during prolonged simulations.

B. Requirements Engineering and System Specification

The requirements for the evaluation of the Adaptive Cruise Control (ACC) function are derived from a combination of multiple sources. The primary standard used is ISO 15622[36], which specifies the performance, safety and functional behaviour requirements for ACC systems. This standard serves as the foundation for defining the control strategy and minimum functional requirements of ACC systems, including parameters such as time gap (τ), clearance (c) and ego vehicle speed (v).

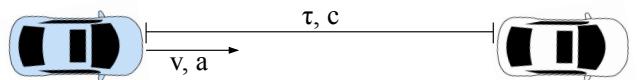


Fig. 5: ISO 15622 (Intelligent Transport Systems: Adaptive cruise control systems – Performance requirements and test procedures) defines the basic control strategy and minimum functional requirements for ACC systems. The time gap is introduced as τ , the clearance as c and the ego vehicle speed as v .

From this, we derive the requirement that the minimum clearance should satisfy:

$$\text{MAX}(c_{\min}, \tau_{\min} \cdot v) \quad (1)$$

In addition, we define a nominal following distance of 10 meters as an optimal balance between safety and driving

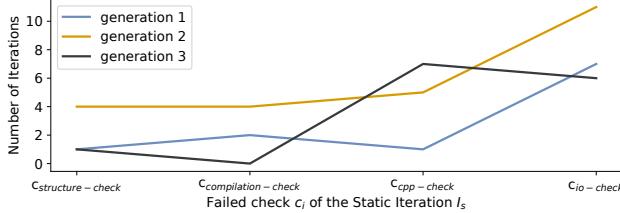


Fig. 6: Across three independent code generation runs using Qwen2.5-Coder-7B-I, an average of 16.3 iterations were required to pass all static tests. Structure and compilation checks failed the least (2 iterations on average). While the CPP check, including MISRA compliance, took an average of 4.3 iterations, the most demanding phase was unit testing, which required 8 iterations on average to ensure correctness.

comfort. In addition, to ensure physically possible driving manoeuvres, we set the requirement to reduce the maximum possible acceleration (positive and negative). The maximum acceleration is therefore defined as:

$$|a| < 5 \text{ m/s}^2 \quad (2)$$

if no emergency brake is applied. The architecture, interfaces and operating logic are carefully defined, following established best practice in control system development. Generative AI is given controlled access to throttle and brake APIs, complemented by real-time vehicle state data and object detection results, enabling seamless integration with the wider autonomous driving (AD) system. Software quality and reliability is ensured by MISRA-compliant static analysis, which facilitates early detection of potential problems. In addition, preconditions for integration monitoring, as outlined in ISO 15622, form the foundation for subsequent test phases, ensuring thorough evaluation and compliance with critical standards.

C. Evaluating the Performance of the generated ACC System

To analyze the behavior of the LLM across both software states (Figure 3), we evaluate its ability to generate code that meets the requirements of the verified state (static checks) and the safe state (combined static and integration checks). We apply the defined requirement checks and initiate code generation, running three independent iterations to analyse and compare the number of attempts it takes the LLM framework to produce code that satisfies the static requirements of a given check (Figure 6).

To ensure effective testing, the behaviour of the leading vehicle is randomised during integration monitoring. This approach prevents the LLM from tailoring the generated function to a specific scenario, thereby improving generalization. The results, shown in figures 7 and 8, demonstrate consistent compliance with the defined requirements. The generated functions maintain the expected vehicle behaviour, and even under emergency conditions the deceleration remains within the prescribed safety threshold.

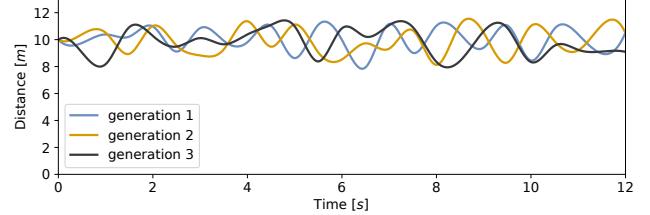


Fig. 7: The distance between the ego vehicle and the leading vehicle over time shows that all three generated functions maintain a consistent following distance of approximately 10 metres. The behaviour shows only minor differences across all implementations, with adjustments occurring mainly in the 8m to 12m range.

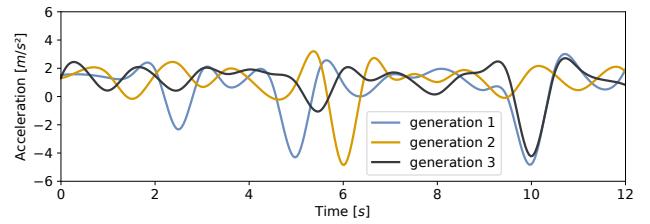


Fig. 8: The acceleration profiles over time show the typical oscillatory behaviour of all the functions evaluated. Notably, each function adheres to the predefined safety requirement, ensuring that the braking does not exceed the threshold of 5 m/s².

V. CONCLUSION AND FUTURE WORK

This work addresses the problem of integrating generative AI into safety-critical automotive software development. Specifically, this work aims to (1) develop an LLM-driven software development framework that ensures compliance with functional safety requirements and (2) establish a structured pipeline for validation through static code analysis and integration monitoring. By using structured specifications, automated refinement, and iterative validation, the proposed framework enables efficient and reliable software generation for safety-critical applications. Using a case study on adaptive cruise control (ACC) case study, our experiments demonstrate that the generated functions meet predefined safety and performance constraints, maintaining safe following distances and adhering to acceleration limits even in emergency scenarios.

Future work will extend the framework by adapting mathematical guarantees to prove software correctness. Additional extensions could involve increasing the number and complexity of requirements to better assess the robustness of various LLMs, as well as evaluating their performance across different automotive functions. Furthermore, the exploration of system-level AI-driven software design methodologies could facilitate the development of a structured model that improves the integration of generative AI into the automotive

software development lifecycle, building upon and refining the SPICE standard. Further evaluation in real-world testing will also be critical to validate the effectiveness of the framework in practical deployment scenarios.

REFERENCES

- [1] Roger Pressman. *Software Engineering: A Practitioner's Approach*. 7th ed. USA: McGraw-Hill, Inc., 2009. ISBN: 0073375977.
- [2] Alexandru Constantin Serban et al. "A Standard Driven Software Architecture for Fully Autonomous Vehicles". In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2018, pp. 120–127. DOI: 10.1109/ICSA-C.2018.00040.
- [3] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: <https://arxiv.org/abs/2005.14165>.
- [4] Alexey Dosovitskiy et al. *CARLA: An Open Urban Driving Simulator*. 2017. arXiv: 1711.03938 [cs.LG]. URL: <https://arxiv.org/abs/1711.03938>.
- [5] Ian Sommerville. *Software Engineering*. 9th. Boston, MA, USA: Addison-Wesley, 2011.
- [6] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. 8th. New York, NY, USA: McGraw-Hill, 2014.
- [7] Abraham Silberschatz et al. *Database System Concepts*. 7th. New York, NY, USA: McGraw-Hill, 2020.
- [8] Len Bass et al. *Software Architecture in Practice*. 3rd. Boston, MA, USA: Addison-Wesley, 2012.
- [9] Kent Beck. *Test-Driven Development by Example*. Boston, MA, USA: Addison-Wesley, 2003.
- [10] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 2002.
- [11] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2004.
- [12] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA, USA: Addison-Wesley, 2001.
- [13] Nir Shavit and Maurice Herlihy. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2012.
- [14] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1st. USA: Prentice Hall PTR, 1997. ISBN: 013215871X.
- [15] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840.
- [16] Daniel Marjamäki. *Cppcheck: A static analysis tool for C++*. 2024. URL: <https://cppcheck.sourceforge.io/>.
- [17] Google. *Google Test: C++ Testing Framework*. Accessed: 2023-12-10. 2023. URL: <https://github.com/google/googletest>.
- [18] International Organization for Standardization. *ISO 26262: Road vehicles – Functional safety*. Geneva, Switzerland: ISO, 2018.
- [19] VDA QMC Working Group 13. *Automotive SPICE Process Assessment Model*. Berlin, Germany: German Association of the Automotive Industry (VDA), 2022.
- [20] Motor Industry Software Reliability Association. *MISRA C: Guidelines for the use of the C language in critical systems*. 3rd. UK: MISRA, 2012.
- [21] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [22] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: <https://arxiv.org/abs/2407.21783>.
- [23] Binyuan Hui et al. "Qwen2. 5-Coder Technical Report". In: *arXiv preprint arXiv:2409.12186* (2024).
- [24] Anna Scius-Bertrand et al. "Zero-Shot Prompting and Few-Shot Fine-Tuning: Revisiting Document Image Classification Using Large Language Models". In: *Pattern Recognition*. Springer Nature Switzerland, Dec. 2024, pp. 152–166. ISBN: 9783031784958. DOI: 10.1007/978-3-031-78495-8_10.
- [25] Zhuosheng Zhang et al. *Multimodal Chain-of-Thought Reasoning in Language Models*. 2024. arXiv: 2302.00923 [cs.CL].
- [26] Aobo Kong et al. *Better Zero-Shot Reasoning with Role-Play Prompting*. 2024. arXiv: 2308.07702 [cs.CL]. URL: <https://arxiv.org/abs/2308.07702>.
- [27] Merlijn Sevenhuijsen et al. *VeCoGen: Automating Generation of Formally Verified C Code with Large Language Models*. 2025. arXiv: 2411.19275 [cs.SE]. URL: <https://arxiv.org/abs/2411.19275>.
- [28] Malin Eriksson and Victor Hallberg. "Comparison between JSON and YAML for Data Serialization." PhD thesis. 2011. URL: <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-130815>.
- [29] Linzheng Chai et al. *McEval: Massively Multilingual Code Evaluation*. 2024. arXiv: 2406.07436 [cs.PL].
- [30] Aider-AI. *Aider Code Editing Benchmark*. <https://github.com/Aider-AI/aider/blob/main/benchmark/README.md>. Accessed: 2025-01-22. 2024.
- [31] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].
- [32] DeepSeek-AI et al. *DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence*. 2024. arXiv: 2406.11931 [cs.SE].
- [33] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE].
- [34] Mistral AI. *Codestral: Hello, World!* Accessed: January 27, 2025. 2024. URL: <https://mistral.ai/news/codestral/>.
- [35] Dong Wu et al. "YOLOP: You Only Look Once for Panoptic Driving Perception". In: *Machine Intelligence Research* 19.6 (Nov. 2022), pp. 550–562. ISSN: 2731-5398. DOI: 10.1007/s11633-022-1339-y. URL: <http://dx.doi.org/10.1007/s11633-022-1339-y>.
- [36] International Organization for Standardization. *Intelligent transport systems — Adaptive Cruise Control systems — Performance requirements and test procedures*. 2018. URL: <https://www.iso.org/standard/71515.html>.