Friedrich-Alexander-Universität
Technische Fakultät

Master's Thesis

AI Usage in CI/CD/CT Pipelines for Compute Platforms in Automotive

by

Morris Darren Babu

Matrikel-Nr.: 08150815

Supervision:
University Supervisor:
Prof. Dr.-Ing. Jürgen Teich
Dr.-Ing. Stefan Wildermann

Industrial Supervisor (CARIAD SE):
Dr.-Ing. Bernd Schaller
Dr. Andreas Weichslgartner

January 16, 2026

This document was produced with the typesetting system LaTeX2e.

Aufgabenstellung zur Masterarbeit

Hier kommt die Aufgabenstellung hin.

Abstract

Modern automotive software now exceeds 100 million lines of code per vehicle, making thorough security testing essential under ISO 26262 and UNECE WP.29 regulations. Fuzz testing is good at finding bugs, but writing fuzz drivers requires deep API knowledge. This skill many teams lack.

This work explores whether Large Language Models (LLMs) can automate fuzz driver generation for C++ automotive libraries and whether they work well enough for enterprise CI/CD pipelines.

We tested 14 open-source LLMs from 1.5 billion to 46.7 billion parameters on `yaml-cpp`. Our results surprised us: larger models performed worse. General-purpose models like Mixtral (46.7B) and CodeLlama (34B) failed completely with 0% code coverage. In contrast, the specialized Qwen 2.5-Coder (32B) consistently achieved 45% line coverage. This shows that domain-specific training matters more than raw model size.

To address resource constraints, we fine-tuned a small 1.5B model using Low Rank Adaptation (LoRA). This reduced generation time by 33% and token usage by 55%, while maintaining driver quality. This makes deployment in resource-constrained environments practical.

The work also revealed real infrastructure challenges. Integrating our system into CARIAD's CI/CD exposed network barriers. Corporate firewalls blocked external API calls. We solved this using Azure Private Link and self-hosted runners. This experience showed that production deployment depends as much on infrastructure planning as on model quality.

Cost analysis shows an enterprise deployment would cost about 1,500 euros annually. This is minimal compared to hiring dedicated security engineers. Overall, LLM-based fuzzing is both technically viable and cost-effective for automotive security, provided the model is chosen carefully and infrastructure is adapted as needed.

Keywords: Large Language Models, Fuzz Testing, Automotive Security, CI/CD Integration, LoRA, ISO 26262.

Contents

List of Figures

List of Tables

List of Abbreviations

ADAS	Advanced Driver-Assistance Systems
AFL	American Fuzzy Lop
AI	Artificial Intelligence
API	Application Programming Interface
ASan	AddressSanitizer
AST	Abstract Syntax Tree
BRS	Business Requirements Specification
CI	Continuous Integration
CT	Continuous Testing
ECU	Electronic Control Unit
HLD	High-Level Design
JSON	JavaScript Object Notation
LLD	Low-Level Design
LLM	Large Language Model
LoRA	Low-Rank Adaptation
SRS	System Requirements Specification
SUT	Software Under Test
UBSan	UndefinedBehaviorSanitizer

1 Introduction

1.1 Background and Motivation

Automotive software has changed fundamentally in the last 15 years. When I first read about vehicle cybersecurity, I was surprised to learn that a modern car runs over 100 million lines of code. Not thousands. Not millions. Over 100 million. Distributed across more than 100 Electronic Control Units (ECUs). This number, which appears in industry reports [5, 4], illustrates a complete shift in how we think about vehicles.

This shift came from somewhere. Vehicles used to be mechanical systems. An engine controller managed fuel injection. An ABS system prevented wheel lockup. These systems mostly operated independently. A skilled engineer could understand the complete electronic architecture within weeks. That is gone now.

Today, vehicles are software platforms that happen to have wheels. Tesla demonstrated this most clearly. When Tesla updates a vehicle over the air, customers receive new features, better performance, and improved safety. All without visiting a service center. Traditional automakers watched this happen and realized something fundamental had changed. Software now determines competitive advantage. It determines how fast a car accelerates. It determines what features customers receive. Mechanical engineering no longer drives differentiation.

This transformation created a problem. Software is complex. A single vehicle contains code spanning multiple programming paradigms, different safety criticality levels, and various execution environments. Code that is safety-critical on AUTOSAR platforms follows entirely different development rules than infotainment applications running on Linux. Real-time control systems operating at microsecond intervals coexist with cloud-connected services. This diversity makes testing exponentially harder.

Consider a Level 2 automated driving system. Perception algorithms process camera, radar, and lidar data. Planning modules compute safe trajectories. Control systems translate trajectories into steering and throttle commands. Each layer requires different technical expertise: computer vision, machine learning, control theory, real-time systems. Integration alone consumes enormous effort. When you add legacy code to this, code written years ago with minimal documentation, understanding what the software actually does becomes detective work.

1.1.1 Security Challenges in Modern Systems

This complexity creates security risks that are genuinely dangerous.

When a web server gets compromised, the consequences are serious. Data theft. Service disruption. Financial loss. When a vehicle control system is compromised, people die. This is not theoretical. In 2015, researchers demonstrated remote exploitation of a Jeep Cherokee. They showed scenarios where attackers could disable brakes or seize steering at highway speeds [19]. Chrysler recalled 1.4 million vehicles. The infotainment system, connected via cellular networks, had provided a pathway to safety-critical networks.

Regulators noticed. UNECE Regulation 155 now requires cybersecurity management systems for all new vehicle types [25]. ISO/SAE 21434 sets cybersecurity engineering requirements across the entire vehicle lifecycle [15]. These are not suggestions. Vehicles that do not comply cannot be sold in regulated markets. Manufacturers must demonstrate that security was designed into systems from the beginning, not bolted on afterward.

This regulatory pressure collides with a fundamental tension in automotive security testing. Safety-critical systems undergo exhaustive verification against precisely defined requirements, like ISO 26262 [14]. Every possible failure mode is analyzed. Fixes are tested. This works for safety, where hazards are known and physics are deterministic.

Security is different. The adversary is intelligent, creative, and unconstrained by specifications. Attackers do not limit themselves to documented interfaces. They probe for implementation weaknesses, logic errors, overlooked edge cases. Testing for security means exploring the unexpected, not verifying the expected. This requires fundamentally different approaches.

1.1.2 CI/CD Pipeline Context

Modern software development has embraced a philosophy: integrate code frequently, test automatically, deploy rapidly. The reasoning is sound. Small, incremental changes are easier to understand and debug than large infrequent releases. Automated testing catches regressions before they spread. Fast feedback loops let developers address problems while context is fresh [13, 9].

The automotive industry has adopted these principles, adapting them to domain-specific constraints. You do not casually push untested code to vehicles traveling at highway speeds. Automotive continuous integration pipelines typically feed into formal release processes with human review and additional validation before software reaches production vehicles.

But the core principles hold: automate what you can, test early and often, maintain a continuously releasable codebase [17].

This creates a tension. A well-optimized pipeline might complete build and basic

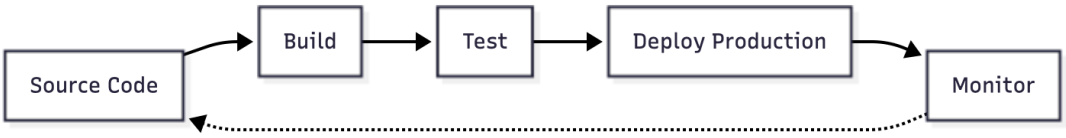

Figure 1.1: The Automotive CI/CD Pipeline. The diagram illustrates the stages from code commit through deployment, highlighting where security validation typically occurs.

testing in 15 to 30 minutes. Developers expect rapid feedback. But security testing, especially fuzzing, often requires extended execution. A fuzzing campaign might run for hours or days to discover subtle vulnerabilities [8]. This timescale is incompatible with developer expectations for fast feedback.

Enterprise environments add additional complexity. Large organizations balance computational demands against security requirements. Build runners may execute in public cloud for scalability while sensitive operations stay in private networks isolated from external connections [2]. Network policies restrict communication between zones. A simple task like calling an LLM API from a build runner encounters firewalls, proxy servers, and access control systems.

At CARIAD, this became a concrete problem during our work.

1.2 Problem Statement

The core problem is straightforward: code production vastly outpaces security analysis capacity.

At CARIAD, we have hundreds of developers producing thousands of changes daily. Even well-staffed security teams cannot review each change for vulnerabilities. Automated tools help, but static analysis generates false positives requiring human triage, and dynamic analysis requires test harnesses that must be created manually.

Fuzzing demonstrates this bottleneck clearly. Fuzzing works. Google's OSS Fuzz project has found over 10,000 vulnerabilities in open source software through continuous fuzzing [8, 23]. The technique is proven. The constraint is fuzz driver creation.

A fuzz driver is a small program that feeds generated inputs to target functionality. Writing effective drivers requires understanding API contracts, identifying interesting code paths, and constructing inputs that explore edge cases. This is specialized work. At CARIAD, we have people who can do this, but not enough to keep pace with code production.

The mismatch creates a security gap. Code ships with inadequate fuzzing coverage. Vulnerabilities that could have been discovered through systematic fuzzing remain hidden until attackers find them or security incidents force discovery.

Large Language Models offer a potential solution. These models, trained on vast

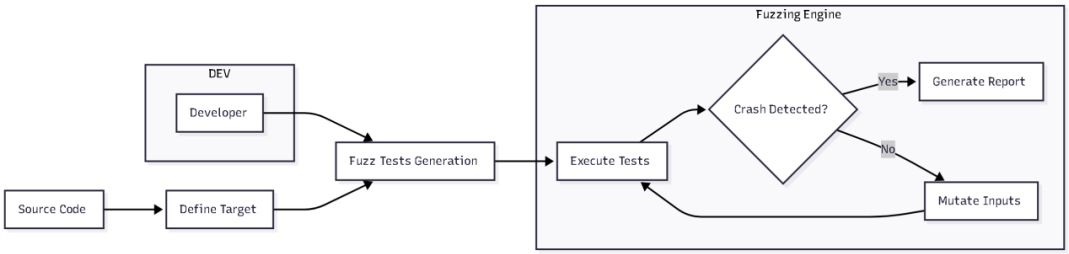

Figure 1.2: The Fuzzing Process. The manual creation of "Fuzz Drivers" (center) is the bottleneck this thesis aims to automate.

amounts of source code, can generate syntactically correct code following common patterns [24, 3]. Recent research shows LLM capabilities in various software engineering tasks: code completion, bug detection, test generation, documentation [7, 29].

The question we asked was simple: can LLMs generate fuzz drivers? Specifically, can they generate drivers that compile, execute without crashing, and achieve meaningful code coverage?

This question mattered to us at CARIAD because if the answer was yes, it could change how we approach security testing at scale.

1.3 Research Questions and Objectives

We had three main research questions driving this work.

1.3.1 Primary Research Questions

RQ1 (Effectiveness): Can Large Language Models generate fuzz drivers for C++ code that compile successfully, execute without errors, and achieve meaningful code coverage?

This is the fundamental question. A driver that fails to compile is useless. One that compiles but crashes immediately provides no security value. A driver that runs but only exercises trivial code paths does not justify its existence. Meaningful coverage means reaching error handling paths, boundary conditions, and complex interaction sequences [32].

RQ2 (Optimization): Does model size determine fuzz driver quality, or can smaller models with domain-specific training match or exceed larger general purpose models?

Everyone in machine learning hears that bigger models are better. Models with more parameters consistently outperform smaller ones on standard benchmarks [16]. The largest models show impressive capabilities across diverse tasks. They are also

expensive, require substantial compute, and often have rate limits. But domain-specific tuning offers an alternative path [12]. Instead of relying on the general knowledge of massive models, we adapt smaller models to specific tasks.

We wanted to know: which approach wins for fuzz driver generation?

RQ3 (Feasibility): Can LLM-assisted fuzz driver generation be integrated into secure enterprise CI/CD pipelines that are isolated from external networks while meeting performance, cost, and security requirements?

Academic research often evaluates techniques in isolation, abstracting away deployment concerns. Production matters. Can the solution run within CI/CD time constraints? Does the cost fit budget limits? Can it operate under network security policies that restrict external communication? Is operational complexity acceptable?

These questions drove our entire investigation. We were not just asking whether LLMs could generate code. We were asking whether this approach could work in a real company, dealing with real constraints.

1.4 Thesis Organization

The rest of this thesis is structured as follows.

Chapter 2: Literature Review surveys relevant research. It examines traditional AI approaches in software testing, traces the evolution of Large Language Models and their application to code generation, reviews fuzzing techniques from foundational methods through AI-enhanced approaches, and examines prior work on security integration in CI/CD pipelines.

Chapter 3: Methodology and Framework presents our approach. It introduces the conceptual framework for AI-driven fuzzing using white-box techniques, describes our technical architecture design, and outlines the three research phases: local LLM evaluation, model optimization with LoRA, and enterprise CI/CD integration. We specify evaluation methodology and metrics.

Chapter 4: Implementation covers the concrete realization. It addresses toolchain selection, describes experimental setup for each phase, documents architectural challenges encountered during enterprise integration, and explains how we solved them.

Chapter 5: Experimental Results presents empirical findings: results of fuzz driver generation across evaluated models, performance variations across target repositories, model optimization outcomes, and economic analysis.

Chapter 6: Discussion and Conclusion interprets results against our research questions. It examines what the data reveals about LLM capabilities, addresses unexpected findings, acknowledges limitations, explores implications for automotive industry practice, synthesizes contributions, and identifies future research directions.

2 Literature Review

2.1 Traditional AI Approaches in Software Testing

Automated software testing has been a research area for decades, and AI techniques have played various roles in its evolution. Understanding the historical context helps explain why LLM-based approaches represent a different class of methods.

Genetic algorithms were among the earliest AI applications to testing. The basic idea was straightforward: treat test case generation as an optimization problem. Maintain a population of test cases, apply selection and mutation operations across generations, and let evolution guide toward more effective tests. Wang et al. documented how evolutionary testing dominated research from the late 1990s through the early 2010s [27].

This approach showed initial promise. Coverage could be formulated as a fitness function. Generate test cases that maximize coverage. Multi-objective optimization helped balance competing goals. Over time, limitations became apparent. Fitness function design proved challenging. Maximizing code coverage did not necessarily correlate with bug detection. Most fundamentally, genetic algorithms lacked semantic understanding. They could shuffle bytes and hope for improvements, but they could not understand what made certain inputs meaningful for a particular API.

Symbolic execution offered a different approach based on constraint solving rather than random search. The technique treats program inputs as symbolic values and accumulates path constraints during execution. Solving these constraints yields concrete inputs guaranteed to exercise specific program paths. Microsoft's SAGE, deployed internally for security testing, demonstrated practical viability by finding vulnerabilities that conventional testing had missed [10].

Symbolic execution offers theoretical rigor. But it faces serious scalability problems. The number of paths through a program grows exponentially with the number of branch points. This is the path explosion problem. Yavuz's recent work on DESTINA addresses this through targeted execution strategies, but complete path coverage remains infeasible for realistic programs [31].

Machine learning entered testing in supporting roles around 2015. A Sandia National Laboratories report surveyed early applications: defect prediction, test case prioritization, coverage estimation [22]. These applications used ML models to inform human decisions rather than generate tests directly. They were practical and useful but did not fundamentally change how tests were created.

Neural networks enabled more ambitious applications. Pei et al. introduced DeepXplore for testing deep learning systems, defining neuron coverage as a testing adequacy criterion [21]. Odena et al. extended coverage-guided fuzzing to neural networks with TensorFuzz [20]. These works showed that neural networks could participate actively in testing rather than merely supporting it.

This context shows why LLM-based approaches represent a different class of methods. Previous AI techniques either optimized based on metrics (genetic algorithms, coverage-guided fuzzing) or reasoned about program structure (symbolic execution). LLMs do something different: they learn patterns from vast code repositories and can generate syntactically correct code following those patterns.

2.2 Large Language Models: Evolution and Code Generation

The Transformer architecture, introduced in 2017, enabled training on vastly larger datasets than previous approaches permitted. Scaling revealed that models trained on next-token prediction could perform diverse tasks.

GPT-3 (2020) demonstrated this principle. With 175 billion parameters, it could write coherent text and syntactically valid code. The specific finding that mattered for software engineering: the model had learned patterns from code in its training data and could apply them to generate new code.

Code-specific models followed. Codex, which powers GitHub Copilot, achieved 28.8% success on the HumanEval benchmark initially and 70.2% with multiple sampling attempts. GPT-4 further improved code generation with better contextual understanding and reduced subtle errors. Context windows expanded from 2K to 128K tokens, enabling inclusion of multiple source files, headers, and documentation in prompts.

Open-source alternatives emerged as important counterparts. Meta's LLaMA demonstrated that competitive performance was achievable with smaller, publicly available models [24]. Code-specific variants like StarCoder and CodeLlama targeted programming tasks specifically. This mattered for automotive applications where intellectual property constraints make sending code to external services problematic.

Researchers have systematically tested LLM code generation. Deng et al. found that models generated syntactically valid code at high rates but frequently contained semantic issues: code that compiled but exercised APIs incorrectly [7]. Wang et al. observed similar patterns in mutation testing experiments, noting high success on simple cases with degrading performance as complexity increased [26].

C presents particular challenges for LLM-based code generation. Training data skews heavily toward Python, JavaScript, and other high-level languages. C's distinctive features, such as pointer arithmetic, manual memory management, preprocessor

macros, and undefined behavior, are less well represented in training data. Empirical evaluations consistently show worse performance on C compared to Python [32]. This has direct implications for automotive applications where C remains the dominant language for embedded systems.

2.3 Foundations of AI-Guided Fuzzing Techniques

Fuzzing originated with Barton Miller's 1988 experiments at the University of Wisconsin, where students subjected UNIX utilities to streams of random input and found that approximately 25% crashed or hung [18]. This established fuzzing as a viable technique for uncovering reliability issues. Early fuzzing was entirely random, requiring no knowledge of target program structure. But simplicity had a cost: random inputs rarely satisfy validation checks and prevent exploration of deeper logic.

Coverage-guided fuzzing represented a fundamental advance. AFL (American Fuzzy Lop), released in 2013, introduced lightweight instrumentation to track which code paths each input explored. Inputs discovering new coverage were retained and mutated further. Inputs revealing no new coverage were discarded. This feedback loop dramatically improved efficiency by directing search toward unexplored behavior [1].

AFL spawned an ecosystem. AFL++ added performance optimizations and additional mutation strategies. LibFuzzer integrated coverage-guided fuzzing into the LLVM toolchain [23]. Syzkaller adapted coverage-guided techniques for kernel testing [11]. Google's OSS-Fuzz project deployed these tools at scale, continuously fuzzing hundreds of open-source projects.

Ding and Le Goues analyzed bugs found by OSS-Fuzz, documenting vulnerability classes that other testing methods had missed [8]. The technique works. The bottleneck is harness creation.

Grammar-based fuzzing addressed structural validity problems. Random byte mutations rarely produce syntactically valid inputs for complex formats. A randomly mutated JSON file is almost certainly invalid JSON. Grammar-based fuzzers generate inputs according to specified grammars, ensuring structural validity while exploring variations. This proved particularly effective for protocol parsers and file format handlers.

Despite these advances, challenges persist in fuzzing practice. Writing fuzz harnesses connecting fuzzers to target libraries remains manual and labor-intensive. Coverage plateaus occur when random mutation cannot discover paths beyond certain checkpoints. These limitations motivate AI-enhanced approaches that can reason about program structure.

2.4 AI-Enhanced Fuzzing: State of the Art

Combining machine learning and fuzzing has led to a growing body of research. This section surveys the current state.

LLM-Based Test Case Generation represents a fundamentally different approach. rather than mutating existing inputs, LLMs generate tests from specifications, documentation, or source code.

Deng et al. introduced TitanFuzz, demonstrating that LLMs could generate effective fuzz targets for deep learning library APIs by prompting models with API documentation [7]. TitanFuzz produced test cases achieving higher coverage than manually written tests for several libraries and discovered previously unknown bugs. This demonstrated that tests produced by LLMs can be applied in real projects.

FuzzGPT extended this approach with focus on edge cases, observing that LLMs possess implicit knowledge of typical API usage patterns from training on vast code repositories [6]. By prompting specifically for atypical or boundary case inputs, FuzzGPT discovered bugs that conventional fuzzers overlooked.

Fuzz4All pursued a more general approach, targeting multiple programming languages and application domains while using LLMs to generate both test inputs and harness code [29].

Zhang et al. directly evaluated LLM-generated fuzz drivers compared to manually written alternatives, finding that results varied significantly across libraries [32]. Some LLM drivers achieved 80 to 90 percent of manual driver coverage. Others reached only 50%. Performance correlated with documentation quality and API complexity.

Previous work worth noting includes HyLLfuzz, which combines LLM-generated seeds with mutation-based fuzzing [28], and WhiteFox, which uses LLMs for targeted compiler fuzzing [30]. These are hybrid approaches attempting to combine LLM strengths while mitigating limitations.

Across these studies, authors repeatedly note a gap between code that compiles and code that does something meaningful. LLMs produce code that compiles at high rates but may not exercise meaningful behavior.

2.5 CI/CD Pipeline Security Integration

Continuous Integration and Continuous Testing have become foundational practices for managing complexity while maintaining development velocity. The philosophy is straightforward: integrate code changes frequently, test automatically, deploy validated changes rapidly.

The automotive industry has adapted these principles to domain-specific constraints. Formal release processes with human review precede deployment to production vehicles. Nevertheless, core principles apply: automate what can be automated, test early

and often, maintain a continuously releasable codebase [17].

This creates tension with thorough security testing. A well-optimized pipeline might complete build and basic testing in 15 to 30 minutes. Developers expect rapid feedback. Security testing, particularly fuzzing, requires extended execution to achieve meaningful coverage. A fuzzing campaign might run for hours or days. This timescale is incompatible with developer expectations [17].

Enterprise environments add infrastructure complexity. Large organizations deploy systems for CI/CD across hybrid cloud architectures, balancing computational demands against security requirements. Build runners may execute in public cloud environments for scalability while sensitive operations stay in private networks isolated from external connections. Network policies restrict communication between zones.

I should note that when we started this work, we underestimated how much this infrastructure complexity would matter. We thought the hard problem was generating good fuzz drivers with LLMs. The hard problem turned out to be deploying that solution in CARIAD's real network environment.

2.6 Automotive Software Development and Safety Standards

Automotive software operates under regulatory constraints unlike other domains. Safety standards like ISO 26262 require systematic identification and mitigation of safety risks throughout the vehicle lifecycle [14]. Every failure mode must be analyzed. Fixes must be verified through testing. This fits safety engineering, where hazards are defined and the physical behavior is predictable.

Security engineering faces a different problem. Attackers are intelligent, creative, and unconstrained by specifications. UNECE Regulation 155 and ISO/SAE 21434 require cybersecurity management systems and risk assessment, but implementation requires different approaches than safety engineering [15]. The expected behavior must be verified, but also the unexpected behavior must be tested for weaknesses.

As illustrated in the V-Model (Figure 2.1), which assumes design and testing are perfectly matched. But security breaks this assumption. Safety requires comprehensive coverage of specified behaviors. But, security requires exploration of unspecified attack surfaces. Both demand rigorous testing. Development cycles must accelerate while testing becomes more thorough. Under these conditions, AI-assisted fuzzing is no longer a theoretical idea but a practical need.

2.7 Research Gaps and Thesis Positioning

Based on this review, several specific gaps can be identified.

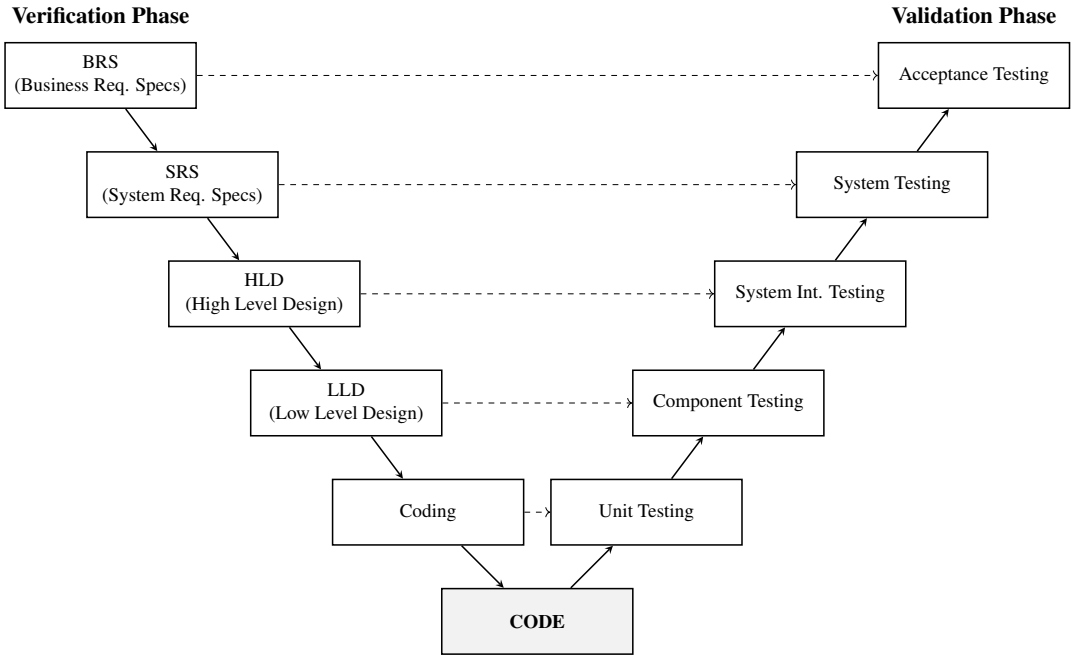

Figure 2.1: The V-Model Development Lifecycle. The diagram illustrates the relationship between each development phase (left) and its corresponding testing phase (right), converging at the Code implementation.

First, while previous work has evaluated LLM-based test generation on individual target libraries, comprehensive evaluation across diverse targets with different models and with explicit analysis of model scale versus specialization is limited. We address this through systematic evaluation of 14 models across multiple C++ libraries.

Second, while optimization techniques like LoRA are well-established in general LLM domains, their application to fuzzing is underexplored [12]. We investigate whether domain-specific fine-tuning of small models can match or exceed larger general purpose models.

Third, academic research typically evaluates techniques in isolation. Production deployment in secure enterprise environments faces constraints such as network isolation, cost limits, and compliance requirements, which academic research often abstracts away. We focus on deployment challenges.

Finally, while prior work examines success cases, systematic analysis of where generated drivers fail and detection mechanisms is limited.

This thesis addresses these gaps through systematic evaluation and explicit attention to enterprise deployment challenges.

3 Methodology and Framework

In this chapter, I explain how the research was carried out, from early experiments to the final deployment at CARIAD SE between May and September 2025. We explain the conceptual foundations of our framework, the technical design that supports it, and the three-phase research strategy that guided our work from initial experiments to enterprise deployment.

The methodology was shaped by several factors. We needed to balance academic goals with industrial needs, since the research took place within a real automotive software development environment. It also needed to adapt to the fast-changing field of large language models, where new models appear regularly. Most importantly, the research had to produce results that could guide both immediate deployment decisions and future research directions.

3.1 Conceptual Framework: AI-Driven White-Box Fuzzing

The conceptual framework for this research combines established ideas from software security testing with new possibilities created by large language models. To explain our approach clearly, we first review the core problem and our proposed solution.

3.1.1 The Fuzz Driver Problem

Fuzzing is an effective technique for finding vulnerabilities, but it relies on a critical component: the fuzz driver. A fuzzer like libFuzzer generates raw inputs (byte streams). A fuzz driver is a C++ program that translates those inputs into meaningful API calls to the target library. It acts as the bridge between the random mutations of the fuzzer and the logic of the library.

Writing effective drivers requires deep understanding of the API. An engineer must know what functions exist, what parameters they accept, what preconditions must be satisfied, and what edge cases might cause problems. This understanding is specialized knowledge. At CARIAD, we have security engineers who can do this, but not enough to keep pace with the volume of code produced by development teams. This manual bottleneck limits the scalability of fuzz testing.

3.1.2 Our Approach: Automated Generation

We investigated whether Large Language Models (LLMs) could automate this process. The conceptual idea is straightforward: feed an LLM a description of a target library API (header files, documentation) and ask it to generate a valid fuzz driver.

If the model produces compilable, executable code that achieves meaningful coverage, we have partially automated what previously required human effort. This addresses the bottleneck directly. Instead of assigning a specialist to write each driver manually, we run an automated pipeline, review the output, and deploy it.

We know LLMs have constraints. LLMs sometimes generate code that compiles but does not exercise interesting behavior. They sometimes misunderstand API contracts. They sometimes generate code with undefined behavior that crashes the fuzzer immediately. These are not theoretical concerns; we encountered all of them during our work.

We recognized that imperfect automation still helps. If an LLM generates a driver that achieves 40% coverage and saves 8 hours of expert time, that is progress. We do not need perfection. We need a solution that is "good enough" to save time overall.

3.2 Technical Architecture Design

Moving from concept to working system required a robust technical design. Our system uses a layered architecture with three main components: Input Analysis, Generation, and Execution.

[Technical Architecture Diagram]

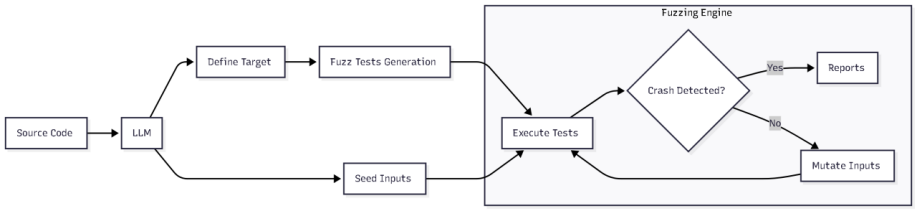

Figure 3.1: Technical Architecture Implementation. The workflow integrates the LLM as an automated frontend to the standard **Fuzzing Engine**, replacing manual driver development while maintaining the established execution pipeline.

As shown in Figure 3.1, the data flows from source code through the LLM to the final execution engine.

3.2.1 Input Layer (Automated Context Extraction)

The first step is preparing the context. Instead of writing custom parsers, we used the analysis capabilities of **cifuzz spark**. This tool automatically parses the project structure, identifying public headers and API surfaces relevant for testing. It handles the complex task of traversing the abstract syntax tree (AST) to extract function signatures and class definitions. This ensures the LLM receives accurate context about the code it needs to test, without requiring manual intervention or maintenance of regex scripts.

3.2.2 Generation Layer (LLM Interface)

This component handles the interaction with the model. It includes two critical subsystems:

- **Prompt Engineering:** We combine the extracted API context with specific fuzzing instructions for the fuzz test case structure.
- **Backend Abstraction:** We configured the environment to swap between different model backends. While ‘cifuzz‘ manages the workflow, we routed the generation requests to both local models (via Ollama) and cloud models (via Azure OpenAI) to compare performance across different architectures.

3.2.3 Execution Layer (Evaluation Pipeline)

Generated drivers are useless if they do not run. This component handles the compilation and execution pipeline.

1. **Validation:** We try to compile the driver against the target library using Clang.
2. **Sanitization:** We compile with AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan) to detect memory errors.
3. **Execution:** If compilation succeeds, the driver runs under `libFuzzer`. The system monitors coverage metrics and records any crashes.

3.3 Research Strategy: Three-Phase Approach

Our research unfolded in three distinct phases, each addressing different aspects of the research questions defined in Chapter 1.

3.3.1 Phase 1: Local LLM Evaluation

We gathered 14 open-source LLM models ranging from 1.5B to 46.7B parameters and evaluated them on multiple C++ target libraries. This phase directly addresses **RQ1** and **RQ2**. Can LLMs generate useful fuzz drivers? Does model size matter? Which models work best?

We chose open-source models specifically because automotive applications have intellectual property constraints. Sending proprietary code to external APIs is often problematic. Local models, run on available hardware, avoid this concern entirely.

3.3.2 Phase 2: Model Optimization with LoRA

Based on Phase 1 results, we selected a promising small model and applied Low Rank Adaptation (LoRA) fine-tuning. We curated a dataset of high-quality fuzz drivers from the Google OSS-Fuzz project to teach the model the specific structure of a good test harness.

The goal was to adapt the model specifically for the fuzzing task while reducing computational cost. This phase addresses **RQ2** more directly: can smaller, fine-tuned models outperform larger general purpose models?

3.3.3 Phase 3: Enterprise CI/CD Integration

The final phase moved from controlled experiments to the real world. We integrated the solution into CARIAD's actual CI/CD infrastructure. This phase addresses **RQ3**: Can this approach work in a real corporate environment with real network security policies, cost constraints, and operational complexity?

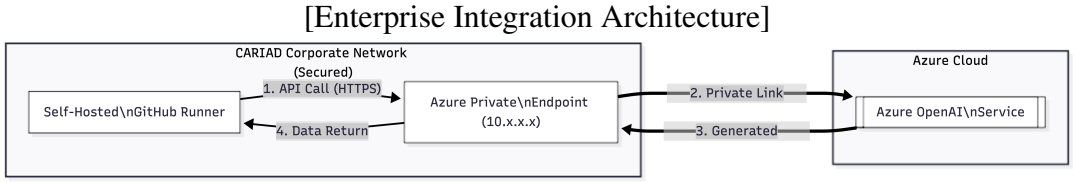

Figure 3.2: Enterprise Integration Strategy. This diagram illustrates the solution to the network isolation challenge using Azure Private Link to connect self-hosted runners to cloud LLM services.

As illustrated in Figure 3.2, this phase revealed challenges we had not anticipated during earlier work. The corporate firewall blocked external API calls, requiring us to re-architect the solution using Azure Private Link.

3.4 Evaluation Methodology

We evaluated generated fuzz drivers using quantitative metrics to ensure objectivity, focusing on both code quality and pipeline impact.

- **Compilation Success:** Does the driver compile against the target library? This is the first and most critical filter.
- **Runtime Behavior:** Does the driver execute without crashing immediately? We checked if the driver actually runs or just fails instantly.
- **Code Coverage:** What percentage of the target library does the driver test? We measured this using standard tools (‘llvm-cov‘).
- **Vulnerability Discovery (MTTV):** We measured the "Mean Time To Vulnerability"—simply put, how long the fuzzer needs to run before it finds a bug.
- **Pipeline Latency (CI Overhead):** We measured how much extra time the AI adds to the build process. The system must stay within the 15-30 minute window developers expect.
- **Cost and Efficiency:** We calculated the exact price of generating a driver. By tracking the number of tokens (words) sent to the AI, we determined the cost in euros to prove it is cheaper than human engineering.

For each target library, we also manually inspected a sample of generated drivers to understand failure modes. We documented cases where drivers compiled but did not work as intended. Finally, we conducted a cost analysis for enterprise deployment, modeling different scenarios to estimate annual expenses.

4 Implementation

4.1 Toolchain Selection and Rationale

4.2 Phase 1: Local LLM Evaluation Setup

4.2.1 Benchmarked Models

4.2.2 Target Repositories

4.2.3 Evaluation Environment

4.3 Phase 2: Model Optimization with LoRA Fine-Tuning

4.4 Phase 3: Enterprise CI/CD Integration

4.4.1 Architectural Challenges

4.4.2 Self-Hosted Runner Solution

5 Experimental Results

5.1 Experimental Setup

5.1.1 Target Selection and Criteria

5.1.2 Hardware and Software Configuration

5.2 LLM Fuzz Driver Generation Results

5.2.1 Successful Models: Performance Data

5.2.2 Unsuccessful Models: Critical Findings

5.2.3 Performance Across Repositories

5.3 Model Optimization Results

5.3.1 LoRA Fine-Tuning Efficiency

5.3.2 Comparative Analysis

5.4 Economic Analysis and Resource Metrics

6 Discussion and Conclusion

6.1 Interpretation of Results

6.1.1 What the Data Reveals

6.1.2 Unexpected Findings: Network Architecture

6.2 Addressing Research Questions

6.2.1 Primary Research Question Analysis

6.2.2 Secondary Research Questions Analysis

6.3 Limitations and Constraints

6.3.1 Technical Limitations

6.3.2 Methodological Boundaries

6.4 Implications for Practice

6.4.1 Automotive Industry Impact

6.4.2 Enterprise CI/CD Challenges

6.5 Future Research Directions

6.6 Summary of Findings and Contributions

6.7 Conclusion

A Appendix

A.1 Pipeline Configuration

```
name: Fuzzing Pipeline
on: [push]
jobs:
  fuzz:
    runs-on: self-hosted
    steps:
      - uses: actions/checkout@v3
      # ... implementation details ...
```

Listing A.1: GitHub Actions Workflow Snippet

References

[1] AFLplusplus Team. *AFL++: afl-fuzz Approach*. 2024. URL: https://aflplus.plus/.

[2] *Azure DevOps Documentation*. Microsoft. 2024. URL: https://learn.microsoft.com/en-us/azure/devops/.

[3] Jinze Bai et al. *Qwen Technical Report*. arXiv:2309.16609. 2023.

[4] Manfred Broy. "Challenges in Automotive Software Engineering". In: *ICSE '06*. 2006.

[5] Robert N. Charette. "This Car Runs on Code". In: *IEEE Spectrum* (Feb. 2009).

[6] Yinlin Deng et al. *Large Language Models Are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT*. arXiv:2304.02014. 2023.

[7] Yinlin Deng et al. *Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models*. arXiv:2212.14834. 2022.

[8] Zhen Yu Ding and Claire Le Goues. *An Empirical Study of OSS-Fuzz Bugs*. arXiv:2103.11518. 2021.

[9] Martin Fowler. "Continuous Integration". In: *martinfowler.com* (2006). URL: https://martinfowler.com/articles/continuousIntegration.html.

[10] Patrice Godefroid, Michael Y. Levin, and David Molnar. "Automated Whitebox Fuzz Testing". In: *NDSS*. 2008.

[11] Google. *Syzkaller: Coverage-Guided Kernel Fuzzing*. 2020. URL: https://github.com/google/syzkaller.

[12] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685. 2021.

[13] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[14] *ISO 26262: Road Vehicles – Functional Safety*. International Organization for Standardization. Geneva, Switzerland, 2018.

[15] *ISO/SAE 21434: Road Vehicles – Cybersecurity Engineering*. International Organization for Standardization. Geneva, Switzerland, 2021.

[16] Albert Q. Jiang et al. *Mistral 7B*. arXiv:2310.06825. 2023.

[17] Thijs Klooster et al. *Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines*. arXiv:2205.14964. 2022.

[18] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Communications of the ACM*. Vol. 33. 12. 1990.

[19] Charlie Miller and Chris Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. Black Hat USA. 2015.

[20] Augustus Odena et al. "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing". In: *ICML*. 2019.

[21] Kexin Pei et al. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems". In: *SOSP*. 2017.

[22] Gary J Saavedra et al. *A Review of Machine Learning Applications in Fuzzing*. Tech. rep. Sandia National Laboratories, 2019.

[23] Kostya Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *IEEE Cybersecurity Development (SecDev)*. 2016.

[24] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. arXiv:2307.09288. 2023.

[25] UNECE. *UN Regulation No. 155: Uniform provisions concerning the approval of vehicles with regards to cyber security and cyber security management system*. Official Journal of the European Union. 2021.

[26] Bo Wang et al. *A Comprehensive Study on Large Language Models for Mutation Testing*. arXiv:2406.09843. 2024.

[27] Yan Wang, Peng Jia, and Luping Liu. "A systematic review of fuzzing based on machine learning techniques". In: *Applied Sciences* 10.16 (2020).

[28] Yue Wang et al. *Large Language Model assisted Hybrid Fuzzing (HyLLfuzz)*. arXiv:2401.12345. 2024.

[29] Chunqiu Steven Xia et al. *Fuzz4All: Universal Fuzzing with Large Language Models*. arXiv:2308.04748. 2023.

[30] Chenyuan Yang et al. *WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models*. arXiv:2310.15991. 2023.

[31] Tuba Yavuz. *Multi-Pass Targeted Dynamic Symbolic Execution (DESTINA)*. 2024.

[32] Cen Zhang et al. *How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation*. arXiv:2307.12469. 2023.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen,

_____

Morris Darren Babu