

Qualification of AI for Embedded Systems Testing: Accelerating DevSecOps Workflows

Rainer Poisel

DevSecOps

Embedded Focus / honeytreesLabs

St. Poelten, Austria

rainer@embedded-focus.com

Abstract—Embedded systems require rigorous testing to ensure functionality, security, and compliance with industry standards such as IEC 62443 and the EU Cyber Resilience Act (CRA). However, traditional test development is time-consuming and heavily manual. This paper presents a test generation framework based on Large Language Models (LLMs) that automates the work preparation phase of test implementation, reducing development effort while ensuring high-quality test cases. The framework generates pytest/labgrid-based test cases from structured specifications and integrates with CI/CD pipelines, enabling automated validation. Tests generated by the system have been successfully executed on QEMU-based virtual machines and physical hardware, demonstrating robustness across different environments. Results show that 70% of the test preparation can be automated, allowing developers to focus on final refinements. This work demonstrates that LLM-assisted test generation can be effectively applied to embedded software testing, improving productivity by automating labor-intensive aspects of quality assurance while also enabling the efficient validation of security requirements.

Keywords—embedded testing; DevSecOps; AI; IEC62443; CRA

I. INTRODUCTION

Embedded systems are vital to critical applications, from industrial automation to consumer electronics, but their growing complexity presents significant challenges for ensuring safety, security, and compliance with international standards. Rigorous testing is essential, yet traditional workflows for test case creation and execution are burdened by inefficiencies, demanding significant manual effort and expertise that often create bottlenecks in the software development life cycle (SDLC). These challenges are further amplified by the stringent requirements set forth by standards such as IEC 62443 [10], [11] and regulations like the EU Cyber Resilience Act (CRA) [6], which require systematic testing and validation to ensure system safety, security, and resilience. Achieving compliance under these frameworks necessitates structured processes that thoroughly verify every aspect of the system against defined requirements.

DevOps emerged as a response to the inefficiencies of traditional software development and IT operations, aiming to bridge the gap between development (Dev) and operations (Ops) through automation, collaboration, and continuous delivery practices. According to Ebert and Serrano, DevOps emphasizes rapid, reliable software delivery by integrating agile development, infrastructure as code (IaC), automated testing, and continuous monitoring [7]. It supports a culture of cross-functional collaboration, reducing deployment cycles and improving system resilience. By incorporating automation pipelines and feedback loops, DevOps accelerates software iterations while ensuring quality and stability, making it a cornerstone for modern software engineering.

While DevOps optimizes software delivery through automation and collaboration, security concerns are often addressed late in the process. DevSecOps extends DevOps by embedding security practices throughout the software life cycle, ensuring that security testing, vulnerability scanning, and compliance checks are integrated into CI/CD pipelines without slowing down development. The requirements defined by aforementioned standards align with the motivations behind DevSecOps. Traditional software development life cycles (SDLCs) struggle to balance rapid development with security and compliance, often leading to bottlenecks in testing and validation. DevSecOps, as outlined in Prates and Pereira [20], extends DevOps principles by embedding security into every phase of development, rather than treating it as an afterthought. It introduces security automation, continuous validation, and compliance-driven workflows, which address the inefficiencies of manual test case creation and execution. Furthermore, the DevSecOps Best Practices Guide [1] emphasizes the role of collaboration, automation, and security testing as fundamental pillars of a secure software development life cycle. A key advantage of applying DevSecOps principles to this test generation approach is its ability to “shift security left”, ensuring that security concerns are addressed at the test preparation stage rather than being detected late in the process. This aligns with the system-

atic testing mandates of IEC 62443, which require structured, repeatable validation steps.

The major contribution of this research paper lies in presenting an AI-driven framework that automates the test preparation phase for embedded systems, significantly reducing manual effort while maintaining high-quality outputs. As product requirements and related test cases are typically expressed in natural language, the proposed system automates the generation of foundational test structures from human-defined specifications, such as step-by-step descriptions formatted in Markdown, reducing manual effort by approximately 70% while ensuring high-quality outputs that can be easily refined by engineers. In order to integrate with existing workflows and tools, the approach utilizes frameworks for device interaction, virtualization platforms for executing tests in virtual environments, and issue-tracking and continuous integration pipelines to automate validation and deployment processes. Additionally, built-in post-processing capabilities such as automatic source-code formatting and linting ensure that generated outputs adhere to stringent coding standards, promoting consistency and alignment with best practices in software development, thereby enabling a more efficient and scalable solution to test preparation challenges.

In this context, we assume that software development is frequently guided by the V-Model, a well-established framework that pairs each development phase on the left side of the "V" with a corresponding testing phase on the right. The V-Model's systematic approach ensures traceability and thorough validation, making it a suitable methodology for addressing the high demands of compliance and reliability. For example, integration testing ensures that individual components work together as intended, while system testing validates the complete system against its functional and non-functional requirements.

However, the V-Model is not the only approach that teams might follow. Alternative methodologies such as agile development or continuous delivery focus on iterative cycles and faster feedback loops. These methods emphasize adaptability and rapid prototyping, benefiting dynamic environments. In many cases, modern development workflows combine the rigor of the V-Model with the flexibility of agile practices, forming a hybrid Agile V-Model approach [17]. As shown in Fig. 1, this hybrid model allows teams to maintain the structured validation of traditional methodologies while incorporating the iterative, feedback-driven principles of agile development.

The approach presented in this paper specifically supports the System Analysis ↔ System Test as well as the Software Design ↔ Integration Testing phases of the V-Model, which are horizontally linked by the validation process. The validation link represent the structured validation process, where system tests are derived from the system analysis phase, ensuring that requirements translate into functionality verified by system tests, while integration tests originate from the software design phase, validating interactions between components. As visualized in Fig. 1, we highlight these validation procedures by marking the relevant phases with a star symbol. By automating the test preparation process for these phases, our framework enhances

efficiency and consistency in embedded software validation, ensuring that test cases align with the specified requirements while reducing the manual effort typically associated with these verification steps.

II. STATE OF THE ART

This section provides an overview of existing research relevant to this work. It first examines the integration of IEC 62443 with DevOps and DevSecOps practices, highlighting challenges and advancements in automating compliance within modern development workflows. It then explores the application of Large Language Models (LLMs) in code generation, particularly in test automation, analyzing their capabilities, limitations, and potential improvements.

A. IEC62443 and DevOps/DevSecOps

Industrial automation and control systems (IACS) face increasing cybersecurity risks due to the convergence of Operational Technology (OT) and Information Technology (IT). IEC 62443 provides a structured approach to managing these risks, but compliance processes remain manual and slow, creating bottlenecks in development [12]. Traditional security validation often happens too late in the life cycle, making early-stage automation crucial [9]. Embedding IEC 62443 compliance checks into DevSecOps workflows ensures security validation throughout development. Göttel et al. [9] highlight the benefits of automated security testing in CI/CD pipelines, reducing risks and improving efficiency. The VeriDevOps framework proposed by Sadovykh et al. [22] employs model-based security verification and NLP to translate security requirements into testable policies. These approaches demonstrate that automation can make compliance more proactive, aligning security enforcement with modern DevOps practices.

Despite progress, fully automating IEC 62443 validation remains difficult. Security testing tools lack complete coverage, particularly for IEC 62443-4-2 component requirements, which require both white-box and black-box testing [15]. The ISuTest framework, though modular, still involves manual setup and evaluation, while automated compliance checks remain expensive due to their poor integration with certification processes [9].

Both Rajapakse et al. [21] and Yasar and Teplov [24] highlight the challenge of integrating IEC 62443 compliance into DevSecOps. They emphasize that traditional security assurance conflicts with agile workflows, making compliance inefficient. Both propose automating security validation in CI/CD pipelines, with Rajapakse et al. focusing on formal verification and Yasar and Teplov on real-time monitoring. However, existing tools lack full IEC 62443 coverage, and compliance automation remains difficult.

B. Using LLMs for Code Generation

Recent advancements in Large Language Models (LLMs) have significantly impacted software testing, particularly in automated test generation. Research demonstrates that LLMs such as GPT-4, Llama, and Codex can generate test cases from

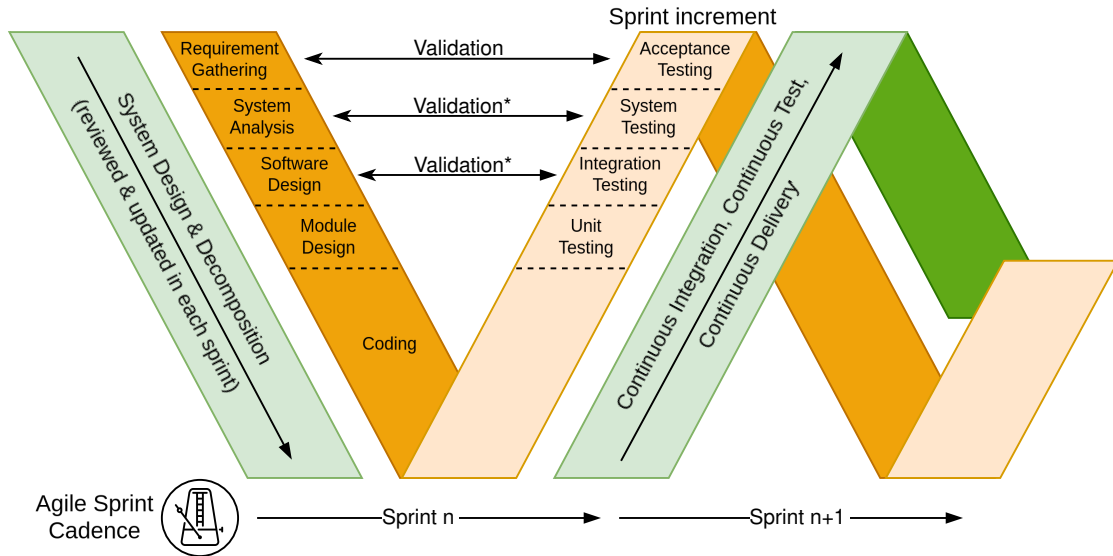


Fig. 1. The Agile V-Model [17]

natural language descriptions, effectively assisting in software quality assurance [14]. However, studies emphasize that LLM-generated tests often rely on inferred requirements from the source code, which can lead to confirmation bias—validating faulty implementations instead of detecting defects [16].

LLM-assisted test generation has been explored in the context of Test-Driven Development (TDD). The LLM4TDD framework proposes an interactive workflow where LLMs generate test cases before implementation, aiming to enhance software correctness and minimize logical errors [16]. Similarly, TICODER, an intent-driven test generation system, demonstrates how interactive test refinement can improve code correctness by matching AI-generated code with developer intent [8]. However, these studies highlight a critical limitation—while LLMs can automate test writing, they still require human intervention to refine tests and correct misinterpretations of requirements.

Empirical studies evaluating LLM-generated tests reveal inconsistent reliability. Research on unit test generation found that test quality depends heavily on contextual information provided in prompts, the complexity of the system under test, and the size of test requests per LLM query [23]. More detailed specifications result in higher-quality test cases, but a lack of explicit test requirements often leads to irrelevant, redundant, or overly simplistic tests. Additionally, frameworks such as PYTHONESS attempt to mitigate these issues by incorporating behavioral specifications and automated test validation, ensuring that generated tests align more closely with software requirements [13].

Based on this literature research, we identified four limitations of current approaches: LLMs tend to generate tests that confirm existing functionality rather than uncover defects. In addition to that, current models need project-specific fine-tuning to be suitable for domain-specific test generation. Most

approaches also still rely on human intervention for reviewing and refining test cases. Large-scale test generation remains computationally expensive. We will focus on these challenges in the main part of this publication.

III. IMPROVING THE EMBEDDED SYSTEMS TESTING WORKFLOW

The main part of this paper provides an overview of the proposed framework, describing its architecture, key components, and critical implementation details. Along with presenting the system’s architecture, it discusses the LLM prompt design, the use of testing frameworks such as pytest and labgrid, and other essential elements that ensure its effectiveness. Additionally, it highlights the results achieved with this approach, showcasing improvements in test preparation efficiency, and scalability in embedded systems testing.

A. System Architecture Overview

This section outlines the core components, their interactions, and the workflows they enable. Fig. 2 illustrates the core components of the LLM test generator framework and Fig. 3 visualizes the steps needed to generate a test from a given test specification (“Issue”).

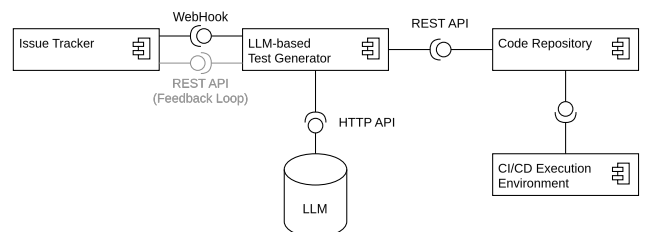


Fig. 2. LLM test generator framework

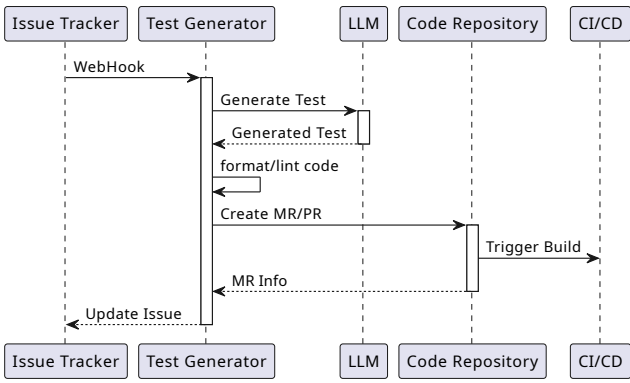


Fig. 3. LLM test generation sequence

The procedure begins with an *Issue Tracker*, such as GitLab or Atlassian Jira, that manages product requirements and test specifications. The *Issue Tracker* provides a structured environment for engineers to define test case specifications in natural language, with test steps typically formatted in Markdown, making them ideal input for Large Language Models (LLMs). These specifications serve as the starting point for the automated test generation process.

The *LLM-Based Test Generator*, the primary innovation of this research, is activated through a *Webhook* interface, which is provided by the generator and triggered by the *Issue Tracker* when a new test case issue is created. This component processes the test case specifications and instructs the *LLM* to generate the foundational test code. The *LLM* itself can be hosted either locally or remotely as a cloud-based service, depending on organizational requirements. Due to privacy and data security concerns, many businesses prefer on-premises deployment to ensure that proprietary test data and intellectual property remain in-house and under full control. The generator interacts with the *LLM* via an HTTP API, requesting test code based on the structured prompt.

Once the test code is generated, it passes through a post-processing pipeline to enforce coding standards and improve quality. This pipeline includes mechanisms such as formatters and linters (e.g., Ruff [3]), which check for issues like line length, unused variables, and adherence to best practices, ensuring that the output aligns with human-written standards. The *LLM-Based Test Generator* then submits the refined test code to the *Code Repository*, represented in this implementation by GitLab, using a *REST API* provided by the repository. Future work will explore using the *Issue Tracker's REST API* for enhanced automation (see Sec. III-D).

The *Code Repository* manages the automated creation and handling of Merge Requests (MRs), serving as a central hub for collaboration and review. The platform's API allows the system to create MRs containing the generated test code, where human engineers validate and refine the work-prepared test code during the review process. Throughout this process, *CI/CD* pipelines execute the generated tests at multiple stages to ensure they function correctly and meet project requirements.

Tests are first executed in virtualized environments using QEMU, which provides a scalable and resource-efficient method for early validation. Running tests in QEMU-based virtual machines reduces hardware dependencies and enables parallel execution on commodity hardware, improving performance while minimizing infrastructure costs. By using virtualization, the system can efficiently identify defects before running the tests on physical devices, ensuring that only properly validated test cases proceed to hardware testing.

In a subsequent phase, tests are executed on physical devices to confirm that functional and security requirements are also met under real-world conditions. This step is particularly crucial for certification processes, such as IEC 62443 compliance, where system behavior must be assessed on the actual hardware. The combination of virtualized and hardware-based testing ensures a thorough verification process, balancing efficiency with the rigor required for embedded system validation.

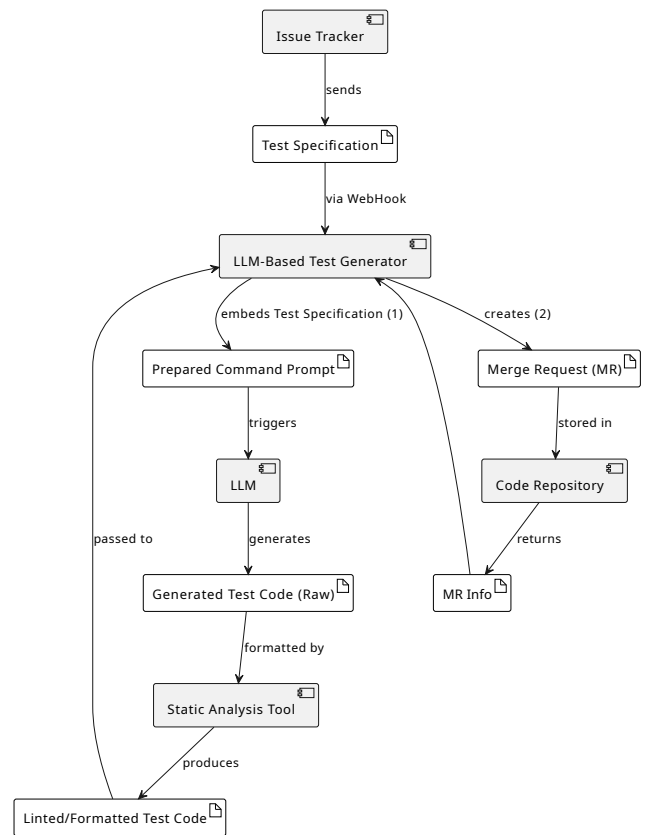


Fig. 4. Artifacts of the test generation process

Fig. 4 illustrates the relationships between the system components and the associated artifacts. The *Issue Tracker* manages issues such as product requirements and test cases. Each test case contains a test specification, where the test steps replace a placeholder in the *Command Prompt*. The *Command Prompt* is a static component within the execution context of the *LLM-based Test Generator*. Once the placeholder is filled with the test steps, the result is the *Prepared Command Prompt*, which is then passed to the *LLM* for generating the raw test

code. After the generated code is linted and formatted, the LLM-based Test Generator creates a Merge Request (MR) in the Code Repository. The corresponding MR Info artifact, containing details such as the Merge Request number, allows the LLM-based Test Generator to update the Issue Tracker, establishing a traceable link between the generated test code and the corresponding test case issue.

B. The Test Framework

Users creating test case issues in the issue tracker trigger the LLM-based test generation sequence. To structure the generated tests effectively, we define two distinct roles in our test system: *test developers* and *test library developers*. Test developers primarily implement and refine test cases, often having less experience in software development. Their contributions have limited system-wide impact, as non-ideal test implementations affect only individual test cases. In contrast, test library developers define reusable, high-level APIs that interact with test devices in a consistent way, ensuring ease of use and maintainability across all tests. Their work has a broader influence, as modifications to the test library affect multiple test implementations. We released the high-level device interaction APIs our experiments are based on in a public GitHub repository [18].

Our test generation prototype is built on pytest [2], a widely used Python testing framework. We chose pytest because LLMs produce high-quality Python code, making it well-suited for automated test generation. Pytest offers flexible setup/tear down mechanisms through its fixture system, extensive test execution hooks, and an ecosystem with hundreds of plugins. Among these, the Labgrid framework [4] allows for orchestrating hardware access for embedded testing. Labgrid provides structured access to physical interfaces, such as serial consoles, power management units, network connections, and SD-Muxer devices. It also facilitates remote device access over networks such as VPNs, allowing test execution across distributed environments. Furthermore, Labgrid’s resource pooling mechanism enables multiple users to synchronize access to shared DUTs, ensuring efficient parallel testing in shared test racks [19].

Many complex test scenarios require DUTs to interact with external services, such as web servers, authentication backends, or VPN concentrators. To minimize dependencies on the test environment, we developed an automated Docker Compose integration that enables ad-hoc service provisioning. This approach ensures that essential infrastructure components can be dynamically instantiated and disposed of as needed, maintaining test isolation and reducing setup complexity.

C. The Test Generation Process

In this subsection, we demonstrate the high-level workflow of the test generation process using a practical example. Specifically, we use the test code generator to produce Python code that implements a test to check the open network ports on the device under test.

The process begins when a requirements engineer creates a test case issue in the issue tracker, which in our implementation

is GitLab. A crucial part of the test case issue is the description, which is formatted in Markdown. This description outlines the steps required to perform the test, as shown in Fig. 5.

Once the test case issue is created, GitLab triggers a configured WebHook for “Issue events.” This WebHook activates the LLM-based test generator, which embeds the test case specification into a prepared prompt. Fig. 6 shows the structured prompt, which ensures test code adheres to project requirements. For example, the *Terminology* section defines key terms such as Device Under Test (DUT) to provide clarity. The *Important Considerations* section includes constraints, such as instructing the LLM to generate only the test’s source code without additional explanations and ensuring that all generated functions include a Python docstring for documentation. The *Code Guidelines* section specifies the API of the underlying testing framework, describing relevant pytest fixtures that provide access to hardware interfaces such as the serial console of the DUT.

The test code generated by the LLM is automatically analyzed by static code analysis which is triggered by the LLM-based test generator. For this purpose, the generator utilizes Ruff [3], a Python linter that includes an auto-fix mode capable of resolving many common issues automatically. This automated step ensures that the generated code adheres to coding standards and best practices, enhancing its quality by identifying and correcting potential errors, stylistic inconsistencies, and deviations from established guidelines. Once the static analysis is complete, the LLM-based test generator proceeds to create a Merge Request (MR) in GitLab, enabling human engineers to review the code for correctness, completeness, and alignment with the test requirements.

An example for the generated code is shown in Fig. 7. While the generated code in our illustrative example is close to being functionally operational, it exhibits several limitations. First, the code does not explicitly verify whether services are listening on all network interfaces (i.e., 0.0.0.0). Additionally, the `grep` command used to filter UDP ports fails because OpenWrt, by default, does not listen on any UDP ports across all interfaces. This failure causes the run function of the test framework to terminate prematurely.

These issues can be mitigated by modifying the code to validate the first field of the column value after splitting it on the `:` delimiter and by appending “`|| true`” to the `grep` command to suppress its exit code. Furthermore, the reliance on `grep` executed directly on the DUT to filter command output is suboptimal, as test implementations should minimize trust in the DUT. A more robust approach would involve parsing and processing all output from the DUT on the trusted host executing the Python test code, thereby reducing dependency on the DUT’s internal state and improving test reliability.

Recent studies have shown that LLMs can complete approximately 70% of developers’ tasks [5]. This finding aligns with our goal of using LLMs to prepare tests for test developers, who then complete the remaining 30%. To achieve this, as outlined in section III-B, we propose maintaining a clear

```

Perform the following steps using the serial console:
1. Log all listening TCP ports using the `netstat` command.
2. Based on the output of the `netstat` command, ensure that services only on these
   → ports are listening on all interfaces:
   - 443
   - 80
   - 22
3. Based on the output of the `netstat` command: make sure there are no services
   → listening on UDP ports on all interfaces.

```

Fig. 5. An example test case in Markdown format

```

Generate a pytest from the following
→ description in markdown format:
```markdown
{description}
```

Terminology:

< ... >

Important:

< ... >

Code Guidelines:

< ... >

```

Fig. 6. The structure of the LLM Prompt we were using

separation between the API design, which guarantees easy-to-use DUT access, and the processes of test code generation and development. This approach makes generating and developing test code more straightforward and efficient while upholding high standards of maintainability.

Our prototype successfully conducts the work preparation step of implementing pytest/labgrid test cases from structured test specifications, and these tests execute correctly out-of-the-box in many cases on both QEMU virtual machines and physical devices. This confirms that the generated tests are functionally valid across different test environments. However, while the current evaluation demonstrates feasibility on a small scale, it does not yet confirm whether the approach can handle large numbers of requirements, such as those necessary for verifying full IEC 62443 compliance for a product. The limited number of test cases generated so far prevents us from drawing statistically significant conclusions about the framework’s scalability in real-world certification scenarios.

D. Ongoing Research

Currently, our framework generates test code using an LLM system prompt that contains all necessary instructions for the work preparation step, i.e., producing an initial test case version. However, we do not yet employ fine-tuned LLMs. To integrate fine-tuning, the system architecture must be extended to support an iterative learning mechanism that incorporates human feedback.

To enhance scalability and adaptability, we propose integrating a feedback loop that supports the fine-tuning process of the LLM based on test specifications (input) and the corresponding human-modified test code (output) that forms the final implementation. The interface required for this feedback loop is illustrated in Fig. 2. To ensure that the original test specification remains accessible not only at its creation but also at the time of fine-tuning, the *LLM-Based Test Generator* must interact with the *Issue Tracker* component.

A key advantage of this approach is that the fine-tuning data originates directly from real project use cases, ensuring that the generated test cases remain closely aligned with actual testing needs. Each test case item in the Issue Tracker is linked to its finalized, human-refined version, providing a structured dataset for training the LLM on how test cases evolve from specification to implementation. As the number of refined test cases grows, the feedback loop continuously strengthens, making the model more context-aware and improving its ability to generate project-specific tests. This adaptation results in continuous improvement, reducing manual effort while increasing the reliability and applicability of AI-generated test cases in real-world scenarios.

Another promising application of this architecture is “No-Code Security”, inspired by “No-Code Development.” This approach enables security audits and system analyses without manual coding. By defining security requirements and verbal descriptions of system parameters, the architecture generates test code to validate security assumptions. For example, requirements like “ensure no unauthorized access to the admin interface” or “validate all network ports are closed by default” can be automatically translated into executable tests. This empowers non-technical stakeholders, such as security analysts, to define and validate security requirements without writing code, while the generated tests are refined and executed within

the existing workflow.

This capability aligns with the goal of making security testing more accessible and scalable. By automating the translation of high-level security requirements into tests, the framework reduces dependency on specialized expertise and integrates security validation into the development process. This accelerates vulnerability identification and supports establishing a proactive security culture.

CONCLUSION AND OUTLOOK

In conclusion, our study demonstrates the effectiveness of an AI-driven framework for automating test preparation in embedded systems development. Using Large Language Models (LLMs) to generate initial test structures from natural language specifications, we significantly reduce manual effort while maintaining high-quality outputs. The integration of static code analysis, automated linting, and continuous integration pipelines ensures that the generated test code adheres to stringent coding standards and is ready for refinement even by less experienced test engineers. Our approach not only accelerates the test creation process but also enhances scalability, enabling execution across both virtual and physical environments. The results indicate a reduction in manual effort by approximately 70%, along with improved product quality and faster feedback loops during development. These outcomes validate the feasibility of integrating AI into DevSecOps workflows, supporting the verification process with rigorous standards such as IEC 62443.

Looking ahead, our future research will focus on enhancing the framework's adaptability and scalability through the introduction of a feedback loop for LLM fine-tuning. By iteratively training the LLM on finalized test cases refined by human engineers, the system can progressively improve its ability to generate contextually relevant and high-quality tests. Additionally, we aim to explore the potential of "No-Code Security," enabling non-technical stakeholders to define and validate security requirements without writing code. Together, these advancements will strengthen the framework's ability to support the evolving needs of modern embedded software development.

REFERENCES

- [1] DevSecOps Best Practices: A Practical Guide. <https://www.kroll.com/en/insights/publications/cyber/devsecops-best-practices-a-practical-guide>. Accessed: 2025-01-23.
- [2] pytest: helps you write better programs. <https://pytest.org>. Accessed: 2025-01-26.
- [3] Ruff – An extremely fast Python linter and code formatter, written in Rust. <https://docs.astral.sh/ruff/>. Accessed: 2025-02-06.
- [4] Welcome to labgrid. <https://labgrid.org>. Accessed: 2025-01-26.
- [5] Addy Osmany. The 70% problem: Hard truths about AI-assisted coding. <https://addy.substack.com/p/the-70-problem-hard-truths-about>, December 2024. Accessed: 2025-01-26.
- [6] European Commission. Regulation of the European Parliament and of the Council on horizontal cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020. <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:52022PC0454>, 2022.
- [7] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE software*, 33(3):94–100, 2016.

- [8] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268, September 2024.
- [9] Christian Göttel, Maëlle Kabir-Querrec, David Kozhaya, Thanikesavan Sivanthi, and Ognjen Vuković. Qualitative analysis for validating iec 62443-4-2 requirements in devsecops. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, page 1–8. IEEE, September 2023.
- [10] International Electrotechnical Commission (IEC). IEC 62443 Series: Industrial communication networks - Network and system security. Standard, 2018. Available from <https://webstore.iec.ch/>.
- [11] ISA Global Cybersecurity Alliance. Quick Start Guide: An Overview of ISA/IEC 62443 Standards Global Cybersecurity Alliance. <https://gca.isa.org/hubfs/ISAGCA%20Quick%20Start%20Guide%20FINAL.pdf>.
- [12] Björn Leander, Aida Čaušević, and Hans A. Hansson. Applicability of the IEC 62443 standard in Industry 4.0 / IIoT. *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019.
- [13] Kyla H. Levin, Kyle Gwilt, Emery D. Berger, and Stephen N. Freund. Effective LLM-Driven Code Generation with Pythoness, 2025.
- [14] Noble Saji Mathews. Code Generation and Testing in the Era of AI-Native Software Engineering. Master's thesis, University of Waterloo, 2024.
- [15] Steffen Pfrang, David Meier, and Valentin Kautz. Towards a modular security testing framework for industrial automation and control systems: Isutest. *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–5, 2017.
- [16] Sanyogita Piya and Allison Sullivan. LLM4TDD: Best Practices for Test Driven Development Using Large Language Models. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024.
- [17] The Digital Playbook. Agile V-Model. https://aiotplaybook.org/index.php?title=Agile_V-Model, 2022. Accessed: 2025-01-15.
- [18] Rainer Poisel. labgrid QEMU Sample. <https://github.com/honeytreelabs/labgrid-qemu-sample>, 2025. Accessed: 2025-01-28.
- [19] Rainer Poisel and Stefan Riegler. Boosting Embedded System Development: A Case for Rapid Testing. In *embedded world Conference 2024*, pages 403–409, 2024.
- [20] Luís Prates and Rúben Pereira. DevSecOps practices and tools. *International Journal of Information Security*, 24(1):11, 2024.
- [21] Roshan N. Rajapakse, Mansoor Zahedi, M. Ali Babar, and Haifeng Shen. Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology*, 141:106700, 2022.
- [22] Andrey Sadovych, Gunnar Widfors, Dragos Truscan, Eduard Paul Enoiu, Wissam Mallouli, Rosa Iglesias, Alessandra Bagnato, and Olga Hendel. Veridevops: Automated protection and prevention to meet security requirements in devops. *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1330–1333, 2021.
- [23] Florian Stifter. Automated Unit Test Generation and Improvement with Large Language Models (LLMs). Master's thesis, Johannes Kepler Universität Linz, 2024.
- [24] Hasan Yasar and Sam E. Teplov. Devsecops in embedded systems: An empirical study of past literature. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–6, 2022.

AUTHORS AND AFFILIATIONS

Rainer Poisel is the founder of Embedded Focus and honeytreeLabs, specializing in DevSecOps for embedded systems. His expertise lies in designing scalable, high-performance development processes, integrating automation, security, and compliance frameworks into modern engineering workflows. With a strong interdisciplinary approach, he bridges the gap between software engineering, embedded systems, and cybersecurity, enabling organizations to modernize legacy infrastructures, optimize CI/CD pipelines, and ensure regulatory compliance. His research-driven methodology and practical focus contribute to advancing secure and efficient development practices in embedded software engineering.

```

import logging

from labgrid.driver import ShellDriver
from process import run

def log_listening_tcp_ports(shell_command: ShellDriver) -> list[str]:
    """Log all listening TCP ports using the `netstat` command.

    :param shell_command: ShellDriver instance
    :return: List of lines from the `netstat` command output
    """
    logging.info("Logging listening TCP ports")
    output = run(shell_command, "netstat -tlnp | grep LISTEN")
    return output.splitlines()

def get_listening_tcp_ports(output: list[str]) -> list[int]:
    """Extract listening TCP ports from the `netstat` command output.

    :param output: List of lines from the `netstat` command output
    :return: List of listening TCP ports
    """
    listening_ports = []
    for line in output:
        columns = line.split()
        if len(columns) > 3 and columns[0] == "tcp":
            port = int(columns[3].split(":")[-1])
            listening_ports.append(port)
    return listening_ports

def check_no_udp_services(shell_command: ShellDriver) -> None:
    """Check that there are no services listening on UDP ports on all interfaces.

    :param shell_command: ShellDriver instance
    """
    logging.info("Checking for UDP services")
    output = run(shell_command, "netstat -ulnp | grep LISTEN")
    if output:
        logging.error(f"UDP services found: {output}")

def test_listening_ports(shell_command: ShellDriver) -> None:
    """Test that services are only listening on allowed TCP ports and no UDP ports.

    :param shell_command: ShellDriver instance
    """
    logging.info("Testing listening ports")
    output = log_listening_tcp_ports(shell_command)
    listening_ports = get_listening_tcp_ports(output)
    allowed_ports = [443, 80, 22]
    assert all(port in allowed_ports for port in listening_ports), (
        f"Disallowed ports found: {listening_ports}")
    )
    check_no_udp_services(shell_command)

```

Fig. 7. Example for generated test code