Optimizing CI/CD Pipelines for Embedded Software: Addressing Constraints, Effectiveness, and Multi-Variant Complexity

University of Turku
Department of Computing
Master of Science Thesis
Software Engineering
June 2026
Moinul Laskar

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Moinul Laskar: Optimizing CI/CD Pipelines for Embedded Software: Addressing Constraints, Effectiveness, and Multi-Variant Complexity

Master of Science Thesis, 73 p., 8 app. p.
Software Engineering
June 2026

Continuous Integration and Continuous Delivery (CI/CD) practices remain difficult to adopt in embedded software development. This is due to hardware dependencies, heterogeneous test environments, lengthy validation cycles, and complex product variants. This thesis explores how these constraints manifest in an industrial GNSS firmware project and how changes to the pipeline can improve CI/CD performance under such conditions. The research uses a structured literature review, a survey of developer perceptions, and technical experiments on an industrial CI pipeline. The analysis considers both technical and organizational aspects of embedded CI/CD, such as build architecture, development practices, test strategies, and product-variants management. It also identifies several system constraints, such as critical dependency paths, pipeline orchestration, test execution latency, and the maintainability of pipeline scripts.
Based on these empirical findings, the thesis designs and evaluates targeted optimization strategies, including declarative CI templates, dynamic test selection, and the use of modern CI platform features. The empirical analysis found that CI/CD optimization is inherently a multidimensional challenge: technical, architectural, and organizational constraints are tightly coupled, and no single optimization technique is sufficient alone. Finally, the thesis discusses how AI-assisted development is likely to accelerate development throughput in large-scale projects, while not eliminating underlying hardware or architectural bottlenecks. As a result, scalable and maintainable CI/CD infrastructure will become increasingly important for large embedded software projects.

Keywords: CI/CD, Embedded Systems, Firmware, DevOps, GNSS, Optimization

Contents

1 Introduction **1**

1.1 Problem statement . . . 1
1.2 Research objectives . . . 3
1.3 Research questions . . . 4
1.4 Scope . . . 4
1.5 Thesis structure . . . 5

2 Background **7**

2.1 Systematic literature search . . . 7
2.2 DevOps and CI/CD in embedded systems . . . 10
2.3 CI/CD pipeline tools . . . 11
2.4 Embedded CI/CD constraints . . . 13
2.5 Organizational constraints . . . 18
2.6 CI/CD optimization techniques . . . 22
2.7 Metrics and observability in CI/CD pipelines . . . 25

3 Methodology **27**

3.1 Research design . . . 27
3.2 Evaluation method . . . 29

4 Case study **31**

4.1 System overview . . . 31
4.2 System constraints and requirements. . . . 35
5 EMFIS model survey **39**
5.1 Survey result . . . 39
5.2 Analysis of EMFIS assessment . . . 40
6 CI/CD optimization strategies **47**
6.1 Overview of optimization approach . . . 47
6.2 Technical challenges . . . 53
7 Results **55**
7.1 RQ1: embedded CI/CD: technical and organizational constraints . . . 55
7.2 RQ2: summary of expectation–reality gaps . . . 58
7.3 RQ3: pipeline improvements . . . 60
7.4 RQ4: architectural and technological impact . . . 66
8 Discussion **69**
8.1 CI/CD optimization constraints . . . 69
8.2 Implications of AI-assisted development . . . 70
8.3 Validity and limitations . . . 71
8.4 Conclusion . . . 72
References **74**
Appendices
A EMFIS survey questionnaire **A-1**
B AI usage declaration **B-1**

List of Figures

4.1 High-level architecture of the GNSS firmware system. 32
4.2 High-level CI workflow of the firmware project. 34

6.1 Unit-test job duration before optimization. 50

7.1 Impact of unit-test parallelization on master branch pipeline duration. 61
7.2 Completion time of parallelized unit tests. 62
7.3 Impact of build-check optimization on per-stage CPU execution time. 63

List of Tables

2.1 Database search results by Research Questions. 8

2.2 Thematic classification of the selected literature. 10

3.1 Mapping of research questions to data sources and methods. 28

5.1 EMFIS evaluation results. 40

7.1 Comparison of technical constraints in embedded and traditional CI/CD systems. 56

7.2 Comparison of organizational constraints in embedded and traditional CI/CD systems. 57

7.3 Quantified impact of native CI/CD templates. 64

7.4 Key factors affecting CI/CD pipeline performance. 67

B.1 Summary of AI tool usage in this thesis B-1

List of Acronyms

CI/CD Continuous Integration and Continuous Delivery.

DAG Directed Acyclic Graph.

EMFIS Enable More Frequent Integration of Software.

GNSS Global Navigation Satellite System.

HiL Hardware-in-the-Loop.

KPIs Key Performance Indicators.

LOC Lines of Code.

MR Merge-Request.

MTTD Mean Time to Detection.

MTTR Mean Time to Recovery.

RTOS Real-time Operating System.

SPL Software Product Lines.

V&V Verification and Validation.

1 Introduction

Continuous Integration and Continuous Delivery (CI/CD) pipelines for embedded software face fundamental technical, architectural, and organizational constraints that differentiate them from traditional software development pipelines [1]. These constraints create systemic inefficiencies that slow down developer productivity, extend feedback loops, and limit scalability in embedded projects. In cloud-native environments, resources are virtualized and can scale abundantly, whereas in embedded pipelines, some automated testing requires access to limited physical devices or test environments. As a result, test cycles are predominantly slower and less deterministic than in software projects, and a significant amount of developer time is spent fixing build and integration issues rather than building new features.

1.1 Problem statement

DevOps practices and Continuous Integration and Continuous Delivery pipelines have revolutionized software development in web and cloud environments by enabling rapid iteration cycles and continuous delivery [2]. However, embedded software development presents unique challenges that resist straightforward adaptation of these practices. The core challenge is that existing CI/CD tools and processes were designed for pure software systems and often fail to address the unique constraints of embedded development. Many embedded teams still operate with a waterfall mindset, resisting iterative approaches and CI/CD principles. In addition, there are

cultural gaps between developer expectations and what current pipeline tools can deliver.

Organizations recurrently face productivity bottlenecks resulting from product line management, fragmented toolchains, slow feedback loops, and hardware-dependent testing, which create deployment delays while consuming expensive hardware resources [1]. The effectiveness of CI/CD pipelines is challenged by the following core limitations:

- **Hardware Limitations:** Physical devices cannot be elastically provisioned like cloud resources, creating system testing bottlenecks. When both hardware and software are developed simultaneously for a new product. In such cases, the hardware may still be under design, available only as a prototype, or completely unavailable [3]. In short, development environments are inconsistent with runtime environments, and heterogeneous hardware platforms further complicate the environment [1].

- **Multi-Variant Complexity:** Embedded CI/CD pipelines deal with the challenge of multiple firmware configurations, hardware targets, and product lines within a single pipeline architecture. Unlike web applications that typically target a uniform cloud environment, embedded software development rarely produces a single, uniform product. There are:

  - Hardware Variants: Different microcontrollers, electronic control units (ECUs), memory sizes and vendor-specific components.
  - Product Variants: Families of products with customized software features tailored for different customers or markets (Software Product Lines).

  The pipeline complexity arises when hardware targets and product features are combined. Because an organization might support multiple software ver-

sions across multiple hardware generations, the number of possible execution environments results in a combinatorial explosion [4].

- **Real-Time Validation Requirements:** Hardware-in-the-Loop testing and timing-critical validation that cannot be fully virtualized and is inherently slow. Literature indicates that simulators inherently model "ideal" electronics and struggle to replicate complex interactions, signal noise, or real-time properties found in production environments [5]. Even if virtualization and emulation tools are available, they are often insufficient for heterogeneous hardware targets and must be complemented by tests on actual target platforms [6].

1.2 Research objectives

The primary objective of this research is to identify the unique constraints in embedded development, assess the effectiveness and complexity of embedded CI/CD pipelines, and improve CI/CD performance by addressing efficient multi-variant product management. The specific objectives include:

- Identify and measure technical and organizational constraints in embedded CI/CD.
- Identify gaps and prioritize factors that enable more continuous integration.
- Design and implement pipeline improvement strategies, including efficient management of multi-variant complexity in embedded CI/CD pipelines.
- Establish measurable metrics for pipeline performance, reliability, and efficiency.

1.3 Research questions

This study addresses four primary research questions:

- **RQ1:** What technical and organizational constraints distinguish CI/CD pipelines for embedded software from traditional software pipelines?
- **RQ2:** Where are the gaps between developer expectations and current experience in CI/CD pipelines and which factors should be prioritized to enable more continuous integration?
- **RQ3:** Which pipeline improvement strategies lead to measurable gains in CI/CD performance in the embedded case study?
- **RQ4:** How do architectural and technological changes impact the performance, scalability, and maintainability of embedded CI/CD pipelines?

1.4 Scope

This study focuses on the constraints, operation, and optimization of CI/CD pipelines for embedded firmware development. The u-blox firmware project [1] serves as the primary case study for this thesis. The scope is limited to the parts of the development lifecycle related to the CI/CD pipeline configuration and its supporting infrastructure. The research includes literature reviews, in-depth analysis of an industrial firmware, and the evaluation of strategies to optimize CI/CD pipelines. The scope of this study includes the following **activities and boundaries**:

1. **System boundary:** The study is primarily limited to the CI/CD pipeline, including pipeline definitions, build and test scripts, artifact handling and deployment steps related to embedded firmware.

[1] https://www.u-blox.com/en

2. **Process mapping:** The current CI/CD process for the selected firmware project is mapped end-to-end to identify all pipeline stages, build variants, tools, infrastructure and human touchpoints.

3. **Data collection:** The study adopts a mixed-methods approach to analyze the selected case. The study will assess both quantitative data (such as GitLab pipeline metadata, job and pipeline durations, and failure logs) and qualitative data (literature, interviews and surveys with u-blox developers).

4. **Implementation:** The study will develop proof-of-concept solutions for managing build variants and pipeline optimization strategies. The developed PoC will be tested for its feasibility and performance within the current memory and hardware limits.

1.5 Thesis structure

This thesis is organized into eight chapters. **Chapter 1** introduces the industrial context, the problem statement, and presents the research questions and scope of the study. **Background Chapter** provides the theoretical background derived from the literature review and presents both technical and organizational constraints for Embedded CI/CD. It also summarizes existing optimization techniques and metrics for observing CI/CD performance. **Chapter 3 (Methodology)** describes the research design and process.

Chapter 4 presents the embedded firmware case study used in this research. It presents an abstracted firmware architecture, the existing CI process, and some key system constraints. **Chapter 5** reports on the EMFIS model survey, identifying developer perception gaps and organizational impediments for Continuous integration. **Chapter 6** describes the implementation of the proposed CI/CD optimization strategies.

Chapter 7 provides the empirical results and addresses the research questions. **Chapter 8** discusses the findings, their implications for embedded CI/CD pipelines, and the study's limitations and concluding the thesis with directions for future work.

2 Background

2.1 Systematic literature search

The systematic literature search targeted three major digital libraries that cover software engineering and embedded systems:

- IEEE Xplore
- ACM Digital Library
- Scopus

For the review of the embedded CI/CD literature for this thesis, a search strategy was adopted following the PRISMA 2020 recommendations for reproducible reporting of information sources and search strings [7]. The base search pattern was adapted to RQ-specific concepts.

Base Search Pattern

All searches followed this hierarchical structure:

"Block 1 (CI/CD concepts) AND Block 2 (embedded domains) AND Block 3 (RQ-specific concepts)"

Block 1 (CI/CD concepts): "continuous integration OR continuous delivery OR continuous deployment OR CI/CD OR devops."

Block 2 (Embedded domains): "embedded OR embedded system OR embedded systems OR firmware OR Internet of Things OR IoT OR cyber-physical OR

hardware-in-the-loop OR HIL."

Block 3 (RQ-specific concepts):

- **RQ1 (Constraints and Challenges)**: "pipeline OR build system OR build time OR feedback time OR feedback loop OR test OR architecture OR constraint OR challenge OR impediment OR bottleneck."
- **RQ2 (Perceptions and Experience)**: "survey OR interview OR case study OR experience report OR perception OR expectation OR developer feedback OR developer satisfaction."
- **RQ3 (Test Selection and Optimization)**: "test selection OR test prioritization OR regression testing OR build optimization OR caching OR feedback time OR feedback loop OR pipeline optimization OR modular OR component-based OR subsystem OR software product line OR variant OR impact analysis OR test orchestration OR staged testing."
- **RQ4 (Architecture and Variants)**: "architecture OR modular* OR component-based OR software product line OR product line OR variant OR pipeline."

In all databases, queries were limited to the `Abstract` field using the advanced search interface, and four RQ-specific strings were executed by combining the generic CI/CD and embedded blocks with RQ-specific terms. This pattern was adapted to the syntax of individual databases and applied with limits to 2010-2026 and English-language publications. This produced several hundred candidate records from the three sources, with the per-RQ hit counts summarized in Table 2.1.

Source	RQ1	RQ2	RQ3	RQ4
IEEE Xplore	97	32	9	55
ACM Digital Library	140	31	12	92
Scopus	297	96	42	200

Table 2.1: Database search results by Research Questions.

After de-duplication, 90 unique records remained and were screened based on their titles and abstracts. Studies were included if they met all of the following criteria:

- The study focused primarily on CI/CD or DevOps.
- The study addressed embedded software systems, cyber-physical systems, firmware development, or other hardware-dependent domains.
- The study provided architectural approaches, industrial experience, methods, frameworks, or technical discussions relevant to CI/CD implementation.
- The study addressed at least one of the following aspects: constraints, architecture, software product variants, testing strategies, pipeline performance and feedback.

Studies outside of software systems, papers lacking technical details, and AI model-driven or exclusive cloud-native workflows were excluded. A full-text review of the remaining papers led to further exclusions. Although the search strategy was structured around the four research questions, the final literature synthesis was organized into six thematic areas. This thematic structure was adopted because many studies addressed multiple research questions simultaneously. Specifically, Sections 2.2 and 2.3 provide the conceptual background for all research questions. Technical and organizational constraints in Sections 2.4 and 2.5 primarily address RQ1 and RQ2 by identifying barriers to effective CI/CD adoption. Optimization Techniques (Section 2.6) and Section 2.7 mainly contribute to RQ3 by examining approaches for improving and evaluating pipeline performance. Table 2.2 summarizes the studies contributing to each theme, which form the basis of Sections 2.2–2.7.

Theme	Key References
DevOps and CI/CD Fundamentals in Embedded Systems	[1], [8], [9], [10], [11]
CI/CD Tools	[12], [13], [14], [15], [16]
Technical Constraints	[1], [4], [5], [6], [9], [10], [13], [17], [18], [19], [20], [21], [22], [23], [24]
Organizational Constraints	[8], [10], [19], [21], [25], [26]
Optimization Techniques	[5], [17], [27], [28], [29], [30], [31], [32], [33], [34], [35],
Metrics and Observability	[26], [36]

Table 2.2: Thematic classification of the selected literature.

2.2 DevOps and CI/CD in embedded systems

DevOps is a set of principles and practices that define, control, and continuously improve software life cycle processes. It is an interdisciplinary approach that integrates software development (Dev) and operations (Ops). Its key objectives are to improve stakeholder communication, software lifecycle management, including specification, development, and operation, and continuous improvement of the lifecycle [36]. In the embedded systems context, DevOps extends to hardware-software codesign and requires managing resource-constrained environments in real time. However, another core challenge involves the adoption of Agile and Lean ways of working into the traditionally long stage-gate development cycles in the embedded domain [1].

Continuous Integration and Continuous Delivery (CI/CD) forms the backbone of modern DevOps practices. Continuous Integration (CI) refers to the frequent integration of source code changes from multiple developers into a shared mainline [2]. CI/CD pipelines automate code compilation, multi-level testing (unit, integration, component, regression), and deployment [8], [9]. In embedded systems, however, CI/CD pipelines must orchestrate Hardware-in-the-Loop (HiL) testing, heterogeneous target architectures, and real-time validation. Automated HiL testing requires the pipeline to interact directly with the physical hardware through serial interfaces,

adapters, and low-level communication protocols. The dependency on physical test rigs may affect test outcomes due to external factors such as communication issues and environmental conditions, leading to flaky behavior. Consequently, the scope of automated testing in embedded CI/CD must expand to handle these physical interactions and hardware constraints before relying on cloud-native CI/CD practices.

Adopting CI/CD in embedded projects requires a systematic assessment of technical, architectural, and organizational constraints and bottlenecks. This challenge is extensively addressed in Torvald Mårtensson's Ph.D. thesis, which focuses on applying CI/CD to large-scale, software-intensive embedded systems [10]. To measure organizational readiness and identify specific bottlenecks, Mårtensson et al. developed the EMFIS (Enable More Frequent Integration of Software) model [11]. This model provides a systematic framework to identify what an organization must prioritize to enable more continuous integration of software. This study utilizes the EMFIS model to address Research Question 2 (RQ2) , where we investigate the gaps between developer expectations and the current state within CI/CD pipelines.

2.3 CI/CD pipeline tools

Modern embedded CI/CD pipelines are increasingly defined as *Pipeline-as-Code*, where the pipeline logic is expressed declaratively and stored in version control together with the source code. Modern CI engines (GitLab CI, GitHub Actions, Travis CI) rely heavily on YAML configuration files (e.g., .gitlab-ci.yml) to define pipelines [5]. Python, Groovy, Ruby, or Shell scripts are used as a scripting language, and even Domain Specific Language (DSL) has been developed to manage the complexity of the CI/CD pipeline [12].

The embedded CI/CD landscape is a hybrid of standard IT tools and domain-specific hardware interfaces. The main categories of tools are:

- **Orchestration and CI servers:** These tools act as the central coordinators of the pipeline, monitoring source code repositories for changes, triggering jobs, and reporting build statuses.

  Jenkins is a common CI orchestrator used in many older, complex embedded projects because of its large plugin ecosystem and flexibility in managing distributed build agents [13]. However, it lacks built-in version control and must be explicitly integrated with external Source Code Management (SCM) repositories.

  GitLab CI has gained popularity because it combines source code hosting and pipeline orchestration within a single platform, with support for Docker-based runners. GitLab CI supports configuration modularity via *include* and *extends*, allowing developers to build reusable templates. Furthermore, GitLab CI enables centralized logic via *!rules* and *!reference*, supports non-linear execution through Directed Acyclic Graph (DAG) pipelines using the *needs* keyword, and supports high-concurrency testing through *parallel:matrix* configurations [37].

- **Infrastructure as Code (IaC)** tools are used to configure and manage the environments in which pipelines run. Ansible is frequently used for the provisioning of CI/CD infrastructure and test environments, generating CI configuration files, and deploying artifacts [14]. Terraform complements this by provisioning cloud or virtualized infrastructure that hosts CI runners, simulators, and auxiliary services. Docker images provide lightweight and reproducible build environments that package cross-compilers and dependencies.

- **Virtualization and containerization**: Docker is the de facto standard for creating reproducible build images that bundle compilers, toolchains, and dependencies, although embedded targets rarely run Docker directly due to re-

source constraints [15]. QEMU is widely used as a machine emulator, enabling pipelines to execute and test firmware on emulated ARM or RISC-V architectures before physical hardware becomes available [38]. Kubernetes (K8S) is an industry-standard container orchestration system that automates the deployment, scaling, management, and networking of containers and facilitates scalable CI/CD infrastructures for embedded software development [16].

2.4 Embedded CI/CD constraints

CI is an indispensable part where a group of developers works concurrently on a large-scale software project. Part of the project's success depends on how effective and reliable the CI pipeline is. Embedded systems development introduces unique layers of complexity to CI/CD pipelines compared to general-purpose software. These complexities stem from the need to manage physical hardware, heterogeneous target architectures, and rigorous compliance standards while attempting to maintain the pace of Agile development. Hardware constraints, misconfigurations, and heterogeneous test environments can easily impact CI pipeline performance and delay the feedback loop for developers.

2.4.1 Hardware dependency

The key contrast with Software/Web development to Embedded development is strong dependency on hardware, and consequently Embedded CI/CD must validate firmware through HiL testing. Hardware-dependent validation has timing constraints and cannot be "scaled out" simply by adding computational resources.

- **HIL bottlenecks**: Telschig et al. (2018), reported that embedded systems often manage time-critical control loops, where sequential operations, such as system boot cycles or functionality tests, consistently takes fixed amount of

time that cannot be fast-forwarded with additional resources [24]. The long execution time of such processes blocks the hardware resources for an extended period. Subsequently, CI/CD pipelines may face long waiting times or limited availability of target devices [1]

- **Resource limitation**: Embedded systems' internal constraints, such as memory and processing power, are significantly limited compared to a typical internet host environment [6]. This limitation directly dictates firmware architecture, for example: the subject codebase achieves compile-time optimization for minimal memory footprint by conditionally including features via *#ifndef* directives, creating many distinct configuration combinations. Although this approach reduces binary size but explodes configuration combinations. This trade-off increases maintenance burden and testing complexity.

- **Target device discrepancy**: Separate Hardware and Software development cycles prolong overall product development. In some cases, the target device for which firmware is being developed may not be available if development occurs simultaneously. Even when target devices are available, end-to-end testing on a physical device may fail non-deterministically due to loose physical connections, environmental noise affecting sensor measurements, network glitches, or because a previous failed test run left the device inoperable or in an unresponsive state. As discussed in Section 2.2, this discrepancy between the development and execution environments inevitably leads to "flaky" tests and causes non-deterministic failures. Such instability reduces developer confidence in the mainline build and delays the discovery of defects [5], [9].

2.4.2 Combinatorial complexity of build variants

Organizations developing embedded software routinely manage complex Software Product Lines (SPL) where a shared codebase supports a large number of hardware variants, customer configurations, and older versions of the firmware. In industrial contexts, such as automotive powertrain control, a single product family may involve over 20,000 software variation points and 800,000 unique calibration parameters. This creates a very large configuration space that is difficult to verify and validate efficiently. "Clone and Own" approach works for a small number of variants, but it quickly becomes unmanageable at an industrial scale. Organizations must therefore manage variability systematically across both pre-compilation and post-compilation phases [4], [19].

The challenge of variability is most visible during the testing phase. In practice, many embedded systems resolve features at compile time using conditional compilation and other build-time mechanisms to minimize the binary size and meet real-time hardware constraints [23]. As a result, different product configurations can generate completely different binaries from the same shared codebase. These binaries can lead to significantly different execution paths, hidden dependencies, and timing behavior. A “test once” strategy is therefore infeasible: unit testing, integration testing, and system validation must be executed across many configurations. The problem becomes even larger in embedded environments because a single product family may need to generate and verify binaries for different microcontrollers (ARM vs. RISC-V), peripheral configurations, and execution environments (simulator vs. FPGA vs. prototypes vs. actual hardware). Each additional hardware or environment option multiplies the number of unique build jobs, test executions, and verification tasks that a CI/CD pipeline must orchestrate. This rapidly leads to a combinatorial explosion in large product lines. With so many hardware configurations and unique binaries, the traditional rule that “100% of tests must pass for

every build" becomes unrealistic and too expensive in terms of time and resources [10].

Architectural strategies based on modularity and abstraction have been applied to mitigate this complexity, for example, by introducing clear subsystem boundaries, reusable components, and standard interfaces. However, each additional abstraction layer adds extra cycles between hardware events and application response. Refactoring legacy systems is even more difficult and requires immense human and financial effort. Ultimately, this forces organizations to abandon exhaustive full-matrix execution and instead rely on advanced test selection, partial sampling, and hardware abstraction approaches to maintain continuous integration efficiency [4], [21], [22].

2.4.3 Monolithic firmware

The embedded systems development has historically converged toward the production of monolithic firmware images to optimize for the limited memory constraints and computational power. As functionality grows over time, weak encapsulation and cross-module dependencies naturally reinforce this monolithic design, creating tight coupling between software components and the underlying hardware platform. Monolithic structures increase build times and feedback latency because the CI system cannot use incremental compilation or parallel "module build" stages. While transitioning to a modular architecture or independent deployment units often requires extensive rework and is costly [20], [22].

Consequently, development teams are often forced into "lock-step evolution," a scenario where all software assets must move to the next release simultaneously. In some cases, a single defect in a non-critical component can stall the entire integration pipeline for multiple teams. Mårtensson et al (2017), further observe that monolithic structures hinder the breakdown of work into small, independently testable units, thereby limiting the effectiveness of continuous integration. In practice, managing

a single monolithic codebase creates a constant conflict between changes and stable integrations that prevents the CI pipeline from scaling effectively [10], [17], [18].

2.4.4 Build system and heterogeneous toolchains

In the embedded domain, build system complexity arises from cross-compilation, where software is developed on a host workstation but compiled into a binary image for a target hardware platform. Moreover, the system must coordinate linking, artifact generation, packaging, and target-specific configuration steps along with compilation. This complexity increases further because each hardware family may require its own combination of compilers, debuggers, flashing utilities, board support packages, and hardware interfaces such as JTAG or other vendor-specific tools. Many of these tools are proprietary, costly and platform-dependent. Differences in versions, operating system requirements, and restrictive vendor ecosystems make it exceedingly difficult to integrate such tools into automated CI/CD workflows [6], [13].

The long lifecycle of embedded products further amplifies this challenge. To maintain older devices and product variants, organizations often need to preserve legacy compilers, historical build scripts, and outdated development environments for compatibility reasons. As a result, CI/CD pipelines must manage highly heterogeneous build environments instead of relying on a single standardized toolchain. They often need to support both modern and legacy tools simultaneously, which increases operational and maintenance overhead [1], [4].

2.4.5 Hybrid validation - virtual and HiL

In the embedded domain, software validation is incomplete until the firmware is executed on the target hardware. However, accessing physical devices during development introduces major bottlenecks, as hardware resources are expensive, scarce,

and difficult to scale. This challenge is compounded by the fact that hardware quickly becomes outdated and new projects often require entirely new platforms or emerging technologies [4], [6], [26].

To address these constraints, embedded CI/CD pipelines adopt a hybrid validation strategy that combines virtualization with HiL testing. In this approach, the early stages of the pipeline use simulation and virtualization to enable rapid logic validation and high test coverage without requiring access to physical devices. By decoupling software logic from physical dependencies, it enables pipelines to run parallel and exhaustive tests, including resource-intensive tests, such as fuzzing and complex integration tests. This strategy also supports the "shift-left" testing principle, where defects are detected earlier, before the time-consuming hardware-dependent tests. Meanwhile, HiL testing can be utilized for final system-level validation and hardware-software interaction.

This approach introduces a trade-off between speed and fidelity. Virtual environments are scalable and fast, but simulators or early prototypes often fail to fully replicate the production hardware behavior. This is particularly true for sensor-driven IoT systems. Sensor behavior depends on real-world conditions, which are difficult to reproduce accurately in virtual environments. In contrast, Hardware-in-the-Loop (HiL) testing provides high fidelity but introduces significant latency. Processes like firmware flashing, device resetting, and real-time execution are much slower than software-only testing [5], [21], [22].

2.5 Organizational constraints

The success of CI/CD in an embedded context depends on organizational culture as much as it depends on technical infrastructure. As discussed in section 2.4, technical constraints involve hardware, architecture, and tools, while organizational constraints refer to the internal processes, structures, and cultural mindset. Orga-

nizational practices govern how software is developed and delivered. Often, established organizational workflows create friction towards automation and consequently CI/CD pipelines.

2.5.1 Conway's law and team structure

Industrial embedded projects face challenges arising from the way development teams and responsibilities are organized. A phenomenon famously described as Conway's Law: "Any organization thats design systems are constrained to produce systems which are copies of the communication structures of these organizations" [39]. In the embedded domain, where development is traditionally divided into isolated silos of module teams, the resulting software architectures naturally mirror these fragmented, historical team structures. Bosch(2010) reported that this led to increased effort as each business unit must balance prioritizing its own products against contributing to the shared platform [17].

This organizational coupling propagates into the CI/CD pipeline as systemic technical debt: tightly coupled subsystems, overlapping responsibilities, and unclear ownership make integration difficult. Consequently, the pipeline frequently encounters severe merge conflicts, checkout failures, and broken builds when multiple teams attempt to synchronize concurrent code changes [8], [21].

2.5.2 Legacy methodologies

The transition to CI/CD in embedded systems is often constrained by a traditional "waterfall" mindset. In the waterfall model, work proceeds in a step-by-step sequence, with requirements, development, and testing treated as separate phases rather than a continuous loop. This is particularly true for the hardware side of development, which ultimately affects firmware development as well. As a result, system integration, field testing, and final validation are often performed near the

end of the development cycle rather than continuously throughout development. This late integration and validation lead to incredibly slow feedback loops, meaning developers often do not discover critical defects or integration issues until much later in the project, and earlier assumptions may require substantial rework.

Similarly, organizational silos create further barriers in this transition. When specialized teams focus solely on their respective piece of hardware or software rather than the whole product, this prevents shared accountability and causes teams to lose end-to-end visibility into the overall product. Even when these organizations try to use modern tools, they often keep rigid "stage-gate" rules that require extensive documentation and manual approvals. Furthermore, Many managers follow a "don't fix it if it's not broken" rule and view software updates as risky events rather than routine automated tasks [1], [25]. This is in direct conflict with emerging European regulation. Software updates, especially security updates and patching vulnerabilities throughout the product lifecycle, are now legal requirements for companies according to the EU Cyber Resilience Act [40].

2.5.3 Compliance, safety standards, and traceability

One of the core values of the Agile Manifesto is “working software over comprehensive documentation” [41]. While this principle works well in many software domains, safety-critical embedded systems operate under very different conditions. Industries such as aerospace, automotive, and medical devices must comply with strict regulatory standards (e.g., DO-178B, ISO 26262) [10], [19]. In these environments, verifiable documentation for safety and compliance is as important as working software. This regulatory landscape is expanding to encompass rigorous data privacy and cybersecurity laws, such as the General Data Protection Regulation (GDPR) and the EU Cyber Resilience Act [40], [42].

These directives impose strict legal requirements on how systems handle data

and require organizations to provide verifiable evidence of security testing, vulnerability management, and encrypted communications throughout the product lifecycle. Moreover, safety-critical systems legally require development, and the V&V teams must be separate [5]. In such regulated environments, quality assurance, auditability, and traceability become critical requirements for ensuring compliance and system reliability. Hence, CI/CD must maintain a verifiable chain linking requirements, source code changes, and test executions. Although these practices improve safety, quality assurance, and accountability, enforcing such traceability and compliance makes CI/CD pipelines significantly more complex. It introduces mandatory compliance checks and approval stages, and results in slow feedback cycles.

2.5.4 Branching overhead and integration delay

In embedded development, organizations frequently attempt to manage complex hardware variants and strict stage-gate processes by maintaining multiple long-lived version control branches. Because these systems must support heterogeneous hardware platforms over extended lifecycles, relying on a singular "master-only" workflow is often practically infeasible [26]. Consequently, teams face a twofold scaling challenge: horizontal scaling (managing numerous active variants of a current product) and vertical scaling (maintaining legacy branches for decade-long support cycles). This practice induces significant "branch overhead," as each active branch requires its own dedicated CI/CD pipeline and testing environment, where each branch steals away limited and expensive hardware resources [25].

In addition, developers working on isolated branches delay merging their work to avoid breaking the shared build or dealing with complex merge conflicts. When these divergent branches are finally merged, teams face "integration delay"—where they must handle complex and time-consuming merge conflicts [10], [21]. As in the organizational coupling described in Section 2.5.1, this branching strategy further

accelerates the accumulation of technical debt across the organization.

2.6 CI/CD optimization techniques

Organizations apply multiple optimization strategies to address the complexity, performance and resource constraints of CI/CD pipelines, particularly in large-scale and embedded systems. This section discusses some of the optimization strategies appearing in the literature:

- **Build acceleration**: Long build times hinder rapid feedback and frequent integration. The most straightforward solution is to improve performance by upgrading hardware resources, such as CPU power and memory capacity, though this carries financial overhead. To optimize further, incremental builds can reduce overhead by recompiling only modified components; however, this requires proper dependency management and decoupled system architectures to accurately determine what needs recompilation. Pipelines also mitigate long build times by distributing build and test processes across multiple execution environments to increase throughput. Finally, caching build artifacts and dependencies further optimizes the process by reducing redundant computation and network overhead [21], [27].

- **Dynamic test selection**: To address the combinatorial explosion of variants described in Section 2.4.2, Test selection technique reduces the volume of executed tests by identifying and running only a specific subset of tests affected by the code change. There are many automated test selection techniques appeared in the literature:

  - **Code churn models:** These models recommend test cases by using historical correlations between source code changes and past test-case failures [28].

  - **Defect-driven selection:** This approach selects black-box tests by prioritizing tests associated with files that previously contained defects [29]. This is simpler than code churn models; it links tests directly to the specific defect without any code-change correlation.
  - **Trace-based selection:** This approach selects the system-level tests by identifying modified parts of the system rather than individual code changes based on system-level communication traces [30].
  - **Search-based test selection:** This method utilizes optimization algorithms and predefined fitness functions to dynamically identify an effective subset of tests [31].

- **Architectural refactoring**: The effectiveness of CI/CD pipelines depends on how well the architecture of the software is testable. To deal with monolithic firmware constraints discussed in Section 2.4.3, many authors advocate for transitioning from monolithic to modular architecture [13], [32]. Breaking the system into smaller, loosely coupled subsystems helps reduce the bottleneck of a single, massive build process. However, empirical studies indicate that systematic modularization is not always feasible under restricted resources. Transitioning to a modular structure requires significant organizational effort and rigorous upfront design and interface specification [4], [17], [33].

- **CI/CD pipeline restructuring**: To maintain CI/CD efficiency, the configuration files defining the pipeline (e.g., YAML scripts) must be optimized and refactored. Developers continually restructure pipelines by removing unneeded environments or scripts, cleanup the build matrix, extracting environment variables, and introducing reusable templates. Restructuring also involves redesigning dependency-driven job orders instead of just stage ordering. In tools like GitLab CI, this is achieved by using the *needs* keyword instead of tradi-

tional *stages*. This allows downstream jobs to trigger immediately as their specific requirements are met, rather than waiting for every unrelated job in a previous stage to finish, and decreases the total lead time [5].

- **Infrastructure tuning and tool migration**: Optimizing the underlying infrastructure is an important strategy for handling the high computational load of CI/CD pipelines. For example, replacing static virtual machines with containerized build agents managed by orchestration platforms such as Kubernetes allows build resources to scale more dynamically. Kubernetes improves resource utilization by distributing workloads across available worker nodes and reduces pipeline queue times by automatically scheduling new jobs on nodes with free resources [27].

  Organizations also improve infrastructure efficiency by introducing different runner classes optimized for specific workloads, such as high-memory or high-CPU tasks [34]. Furthermore, when existing proprietary solutions fail to meet performance, feature, or pricing requirements, organizations may switch to a different CI/CD platform entirely.

- **Pipeline quality and anti-patterns**: CI/CD pipeline scripts themselves are commonly prone to configuration smells, CI/CD anti-patterns, and fuzzy versioning introduced over time by developers. Use of *retry-failure* in flaky jobs or using *allow-failure* to make the pipeline green can mask real issues and compromise the effectiveness of the pipeline [35]. In the Ericsson Case, CI Pipeline Script Errors accounted for 36% of the total compilation failure, indicating that the pipeline scripts can be as fragile as the source code it compiles [8]. As discussed previously, embedded CI/CD is tightly coupled with its test environment and heterogeneous, proprietary toolchains (compilers, debuggers) characteristically restrict how pipelines can be configured.

2.7 Metrics and observability in CI/CD pipelines

To effectively monitor both the pipeline's efficiency and the software's quality, organizations track a specific set of Key Performance Indicators (KPIs) and metrics. The **ISO/IEC/IEEE 32675:2022** standard identifies core DevOps performance measures, including Deployment Frequency, Change Lead Time (the duration required for a code commit to transition from integration to successful production deployment), Mean Time to Recovery (MTTR), and Mean Time to Detection (MTTD), among other commonly adopted DevOps performance indicators [36].

However, many of these metrics are primarily designed for cloud-native or web-service environments where software can be deployed continuously to production. Meanwhile, the delivery process in embedded software projects differs substantially from cloud-native contexts. Firmware releases are often dependent on hardware validation, safety requirements, and product-specific configuration. Furthermore, the "production" is highly complex in embedded environments, as multiple software versions, including legacy variants, may coexist simultaneously due to different product profiles and customer-specific requirements.

As a result, metrics related to frequent production deployment provide limited practical value. Instead, teams are typically more concerned with how efficiently developers can validate changes, how stable the integration branch is, or how well the pipeline scales with an increasing number of hardware targets and variants. For this reason, the study focuses on metrics that are directly relevant to the daily development workflow and the maintainability of the CI/CD system itself. The following three categories present commonly used analysis-oriented metrics:

- **Performance-related metrics:** Metrics such as pipeline duration, individual job duration, queue time, pipeline success rate, and time-to-feedback for typical merge request commits. These measures reflect how efficiently the CI/CD system supports day-to-day development.

- **Scalability-related metrics:** These metrics capture how well the CI/CD infrastructure scales as workload complexity increases. Key indicators include concurrent execution capacity, ability to support an increasing number of hardware variants, resource efficiency, and scheduling delays under load.

- **Maintainability-related indicators:** Reduction in duplicated pipeline configuration, complexity of CI definitions, ease of extending the pipeline for new variants. These indicators help assess the long-term sustainability of the CI/CD architecture.

In addition, workflow-oriented indicators are highly relevant in embedded CI/CD environments. Examples include merge throughput, frequency of integration failures, and the stability of nightly or release pipelines. Pipeline resource consumption or execution cost is also an important consideration, particularly in large-scale CI/CD systems with extensive hardware validation requirements. These measures provide a more realistic view of development efficiency and integration quality than traditional deployment-centric DevOps metrics.

Observability and monitoring tools play a central role in collecting, correlating, and visualizing these metrics across CI/CD pipelines. Platforms such as Grafana and Prometheus are widely adopted for time-series data visualization and creating automated alerts for resource usage, latency, and pipeline failures [26].

3 Methodology

3.1 Research design

This thesis follows a mixed-methods research approach combining qualitative and quantitative analysis [43]. The study is conducted as an information-rich industrial case study based on the GNSS firmware project at *u-blox*, where the CI/CD pipeline infrastructure serves as the primary unit of analysis.

The research design is structured around four research questions (RQs) that progress from problem identification to solution evaluation. The study first establishes the domain-specific constraints of embedded CI/CD systems through a literature review and baseline analysis of the case project. The literature review process followed the PRISMA 2020 guidelines for identification and screening of relevant embedded CI/CD studies [7]. Detailed search strategies and selection criteria are presented in Section 2.1. The next step of the research was to investigate the human and organizational dimensions of CI/CD workflows through an EMFIS-based survey (Chapter 5)[11]. Based on these findings, a set of architectural and pipeline optimization strategies are designed and implemented within the case project, see chapter 6. Finally, the effects of these optimizations are evaluated using quantitative pipeline metrics and qualitative maintainability observations.

3.1.1 Data sources

The study uses the following data sources from the industrial case:

- Pipeline and build configurations (GitLab CI YAML, Makefiles, job templates, custom scripts), used to understand pipeline structure, build-variant handling, and toolchain usage.
- CI/CD pipeline logs and job metadata (timestamps, status, runner information) from an already developed Grafana dashboard, used to compute performance metrics such as total pipeline duration, queue times, and success rates.
- Internal observations, survey results and informal discussions with project members, used to validate interpretations of constraints and to identify feasible optimization options.

3.1.2 Linking methods to research questions

Table 3.1 summarizes the relationship between research questions, data sources, and analysis methods.

RQ	**Primary Data Sources**	**Methods**
RQ1	Literature review, project documentation, informal meetings with stakeholders	Literature synthesis, process mapping, documentation review, baseline pipeline analysis, thematic analysis of discussion notes
RQ2	EMFIS survey responses, pipeline metadata	EMFIS-based assessment, statistical and thematic analysis of survey responses, evaluate CI impediments
RQ3	Pipeline metadata, CI/CD configuration files, developer feedback from surveys	Before-and-after comparison of CI/CD metrics, qualitative assessment of maintainability and workflow impact
RQ4	Results from RQ1–RQ3	Comparative synthesis of findings, evaluate the impacts of changes on performance, scalability, and maintainability

Table 3.1: Mapping of research questions to data sources and methods.

This alignment ensures that the selected methods and data sources provide sufficient evidence to answer each research question and to support the conclusions drawn in Chapters 7 and 8.

3.2 Evaluation method

The evaluation of the optimization strategies was based on the metrics introduced in Section 2.7, with a particular focus on the performance, scalability and maintainability aspects relevant to embedded CI/CD environments. The analysis primarily considered metrics such as pipeline duration, job execution time, queue delays, pipeline success rate, number of concurrent jobs, and the structural complexity of CI definitions.

For each optimization strategy, a baseline configuration and an optimized configuration were compared using production pipeline data. Depending on the optimization scope, the evaluation also considered changes in total job count, execution parallelism, CI configuration size, and the effort required to maintain or extend pipeline definitions for new hardware variants. CI/CD platform metrics were collected through the existing observability infrastructure. Grafana dashboards were used for both real-time visualization and historical analysis of pipeline performance and runner activity. Additional pipeline execution details were obtained from the GitLab interface.

Before the metric-based evaluation, an EMFIS-based assessment captured the perspectives of developers and enablers on the CI/CD process. The survey targeted expectation–reality gaps, workflow bottlenecks, and factors affecting continuous integration practices. It was distributed internally to developers and enablers involved in the pipeline ecosystem, and participants were given approximately three weeks to respond. The EMFIS results were analyzed by computing mean scores and variances for each factor, separately for developers and enablers. Factors with mean

scores below the concern threshold, or with substantial differences between the two groups, were treated as priority areas. These findings informed both the selection of optimization strategies and the interpretation of their impact: improvements were considered more valuable when they directly addressed EMFIS-identified impediments.

The combined use of metrics, EMFIS scores, and developer feedback allowed triangulation: an optimization was considered successful when it produced measurable improvements in one or more key metrics and aligned with perceived needs and expectations in the organization. RQ4 synthesizes the outcomes of the earlier research questions by asking how the chosen architectural and technological changes affect the overall performance, scalability, and maintainability of the embedded CI/CD pipeline. A cumulative analysis was conducted by comparing the baseline state (as characterized in Chapters 4 and 5) with the post-optimization state (as reported in Chapter 7). The synthesis focused on whether the implemented changes:

- reduced or mitigated the most critical constraints identified in RQ1;
- addressed the high-priority gaps and perception mismatches highlighted by the EMFIS assessment in RQ2;
- delivered sustained improvements in the selected CI/CD metrics, rather than isolated gains in narrowly scoped scenarios.

By cumulatively analyzing evidence from all previous research questions, RQ4 provides an integrated view of how architectural and technological changes shape the embedded CI/CD pipeline and whether they move the organization closer to continuous integration in practice.

4 Case study

An industrial GNSS firmware project

This chapter presents the empirical case study conducted in an industrial embedded firmware project which works as a GNSS receiver. The case provides the context in which the research questions, methods, and optimization strategies are initiated and evaluated.

4.1 System overview

The case project develops embedded firmware for a family of GNSS receiver products deployed on ARM-based hardware platforms. The firmware supports several product profiles (e.g., Standard Precision SPG, High Precision HPG) derived from a shared code base. The common firmware platform is configured into multiple variants through build-time options and product-specific features.

Development teams are organized into modular sub-units, such as signal processing, navigation, measurement engine, embedded platform and drivers. All teams collaborate in a single Git-based version-controlled repository and share a common continuous integration pipeline. CI pipelines run cross-compilation, static analysis, unit tests, and integration tests for each change.

4.1.1 Firmware architecture

At a high level, the firmware follows a layered architecture composed of multiple functional subsystems built on top of a lightweight RTOS abstraction layer. Figure 4.1 presents an abstracted view of the firmware architecture derived from project documentation and architectural specifications. The figure is intended to provide a conceptual overview of the system structure and subsystem relationships rather than a detailed functional architecture.

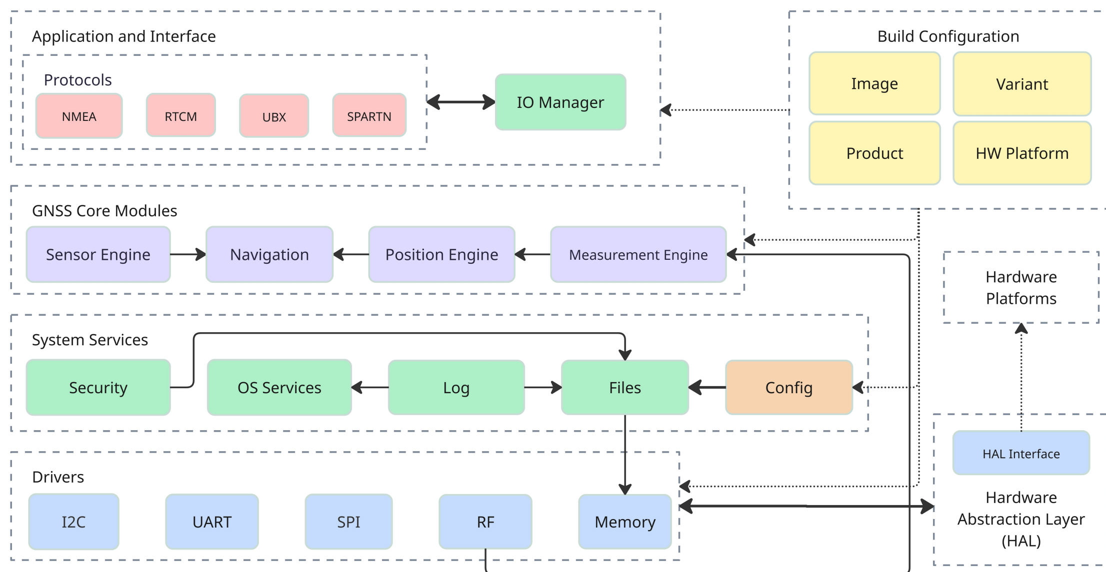

Figure 4.1: High-level architecture of the GNSS firmware system.

A System Service layer provides the underlying infrastructure and lifecycle support, handling all non-navigation functions such as storage, I/O, security, and logging. A global Configuration Manager (CFG) provides runtime configuration to all modules using a shared configuration key database. Above this, the core GNSS modules consist of a Measurement Engine (ME) for RF signal acquisition and tracking, a Positioning Engine (PE) to compute the position, and a Sensor Engine (SE) to process external sensor inputs. These core engines are coordinated by an application and interface layer. The application layer exposes configuration and product-specific behavior via standardized protocols (like UBX or NMEA). A hardware-abstraction

layer isolates these cores from concrete hardware platforms by providing uniform driver and interface APIs for RF, SPI, UART, I2C, and memory control.

Build-time configuration selects platform, image type, product, and feature set using a central project definition. It compiles relevant source files and configuration parameters from the project definition to be built for different hardware boards and product profiles.

4.1.2 CI process

The Continuous Integration (CI) process, shown in Figure 4.2, is implemented using GitLab CI and runs on a pool of containerized runners managed by Kubernetes. The process begins when a developer pushes code changes to the shared Git repository and opens a merge request. This event automatically triggers a merge-request pipeline that validates the code as if it were already merged into the target branch.

As illustrated in Figure 4.2, the pipeline is divided into multiple stages: *quick*, *prebuilt*, *build*, *validate*, *test*, *report*, and *registry*. The *quick* stage runs lightweight validation checks to provide early feedback, such as static analysis, code quality checks, and a few quick smoke tests. If these checks pass, the pipeline moves on to more complex tasks. But later stages are only available after a manual *select* stage. Developers choose a platform family to unlock the related build jobs and their downstream jobs, since the pipeline jobs are connected via a DAG dependency structure.

The *build* stage uses the GNU Make–based build system together with an ARM cross-compiler to produce firmware binaries for selected platform families. The *validate* and *test* stages then run more extensive tests, including additional unit tests, component tests, and integration tests that verify the firmware logic without requiring physical hardware. The pipeline also generates execution reports, stores the result summaries, and publishes firmware artifacts through the *report* and *registry*

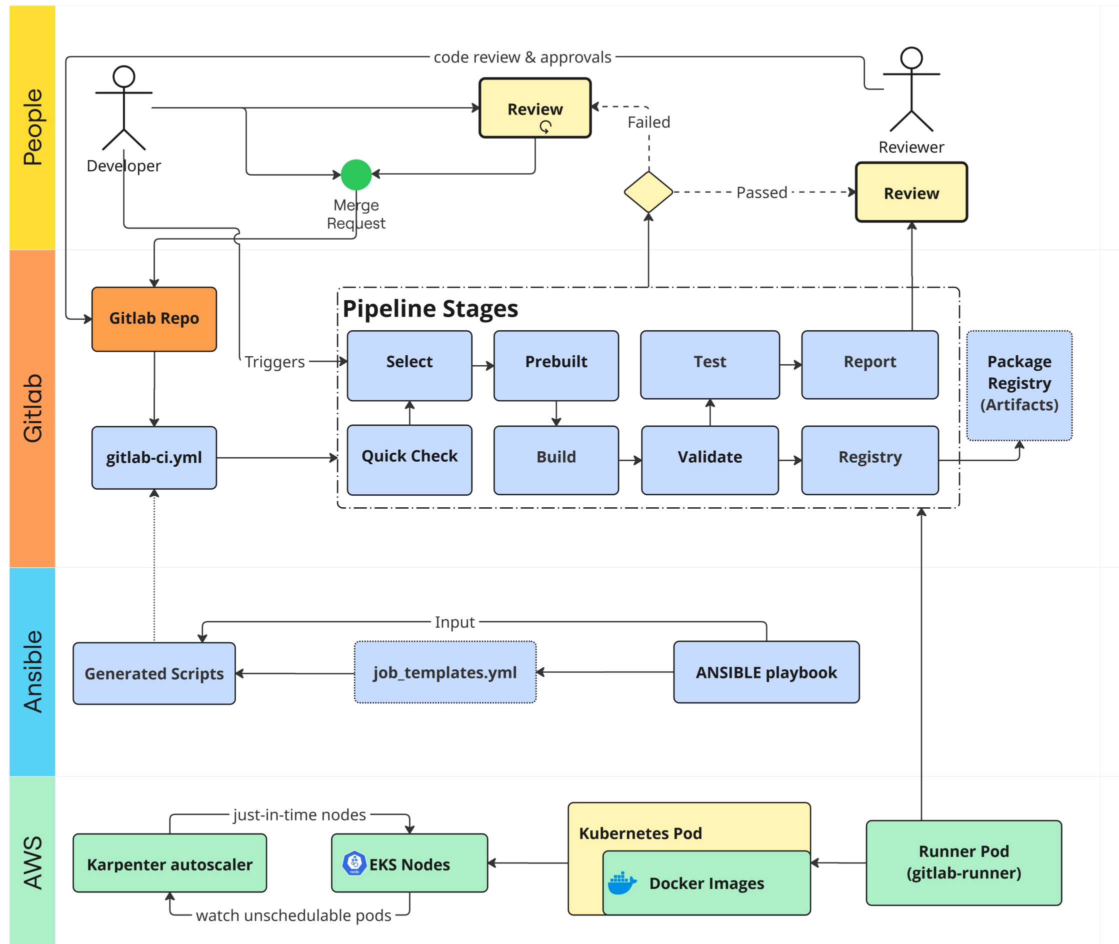

Figure 4.2: High-level CI workflow of the firmware project.

stages.

If any job fails, except those explicitly marked as allowed to fail, the system blocks the Merge-Request (MR). As long as MR remains blocked, the developer must investigate failed jobs, fix issues, and push updates. This cycle continues until a pipeline run completes successfully. Once the pipeline succeeds, code reviewers inspect the changes. They evaluate the code logic and the test results before approving. After the change is approved, it is added to an integration queue (merge train), which ensures that multiple merge requests queued for merging remain compatible with each other before finally merging into the master branch. Merging into the main branch triggers a Master pipeline that re-runs a selected subset of build and

test stages to confirm the stability of the codebase and to create artifacts and metrics. Besides these event-driven pipelines, the project also uses a scheduled nightly pipeline. This pipeline runs once per day, and it executes extensive tests that are too long for MR pipelines. These include system-level regression tests and HiL tests.

The CI configuration is partially generated using Ansible playbooks and reusable job templates that produce the final `gitlab-ci.yml` definitions. At the infrastructure side, GitLab runners run inside Kubernetes pods deployed on AWS Elastic Kubernetes Service (EKS) nodes. A Karpenter autoscaler dynamically provisions worker nodes as needed. This setup allows the system to scale dynamically during periods of high pipeline activity and improves resource utilization across parallel jobs.

4.2 System constraints and requirements.

Several constraints shape the design of CI/CD pipelines in this environment. These include build architecture, pipeline orchestration, and testing infrastructure:

- **Variant management:** The firmware build system organizes configurations through a fixed set of variant dimensions that are embedded into the build logic. The project follows a "4-Slot" variant management model consisting of `IMAGE`, `PLATFORM`, `VARIANT`, and `PRODUCT`. The variant resolve model is tightly coupled to the existing build-system structure, making it difficult to extend without updating every conditional branch that inspects it. As the number of supported products, platforms, and feature combinations increased, the number of build targets and CI jobs grew significantly. This creates a form of combinatorial growth in configuration complexity, reflecting the “complexity barrier” pattern described by Becker et al. [4]. Adding a new product profile further multiplies this complexity across all supported platform configurations.

- **Pipeline orchestration bottlenecks:** The current pipeline structure introduces DAG dependency structure, which improved parallel execution across stages. However, in some cases, DAG extended the critical path and consequently, the overall pipeline duration. For example, validation jobs that could run in parallel with compilation are instead executed afterward. Some post-build jobs are also blocked due to rigid dependency constraints, even though they could safely be executed in parallel.

  Redundant work further increases pipeline overhead. Build checks are executed separately for each build target, and they create many duplicated job definitions that follow nearly identical patterns. This creates maintenance overhead, since even small structural changes — such as adding a new rule or introducing an additional validation step — must be propagated across many repeated job definitions and then revalidated throughout the generated pipeline configurations.

- **Test execution constraints:** The test pipeline spans multiple execution tiers with substantially different latency profiles. Fast checks, such as static analysis, are complete within minutes. Cross-compilation and analysis jobs extend this to the range of fifteen to thirty minutes. Virtual simulation-based tests require up to one hour. Acceptance tests are scheduled nightly and run for up to sixteen hours.

  Moreover, the test setup uses different types of heterogeneous environments. Each test category needs its own execution environment. For example, some tests use dedicated EKS runners, while others need a special tagged board runner. Some tests also trigger child pipelines in external repositories, which extends the failure surface beyond the firmware codebase and adds additional dependencies.

Across the pipeline tiers, a consistent constraint is that high-fidelity testing does not scale well with time and available resources. As a result, some hardware, simulation, and analysis tests are allowed to fail or run in a non-blocking mode.

- **Build system maintainability:** The existing Make-based build system is monolithic and difficult to evolve, which makes it challenging to introduce new variants or restructure the pipeline without risking regressions. It also relies on Regex matching in the build configuration file to find source files. As the system grows, these Regex patterns become "fragile and hard to read." The consequence for the CI pipeline is that build correctness depends on the stability of the directory structure, and it increases the risk of silent regressions when variants are added or modified.

- **Architectural Challenges:** Finally, there are also significant bottlenecks within the system architecture. Although the projects are meant to follow clean, modular, and layered architectural principles, the actual implementation exhibits strong coupling between subsystems. Much of this coupling is inherent to the problem domain; components share global configuration, depend on the same middleware, and interact through timing-sensitive hardware events. Because of this, it is harder to separate subsystems in practice than it looks in architecture diagrams. Consequently, a change in one area may create unintended effects in other parts of the system, increasing integration risk.

  High coupling complicates independent subsystem testing. Isolating a single component often requires extensive use of mocks, fakes, or simulation layers to replace operating system services, hardware interfaces, and other modules. While this test setup can improve test coverage, it might not match timing or hardware interactions on the real target platform.

Together, these constraints show that optimizing CI/CD in this context is not just about changing pipeline code. It is a socio-technical problem that involves both tools and people. The main goal is to reduce duplication, make feedback faster, and improve maintainability. At the same time, there is a need to simplify the configuration without changing how the system behaves. Any successful redesign requires cleaner architecture, while still handling the complexity of a large firmware system with many variants.

5 EMFIS model survey

This chapter addresses RQ2: ***Where are the gaps between developer expectations and current experience in CI/CD pipelines, and which factors should be prioritized to enable more continuous integration?***

The EMFIS model (Enable More Frequent Integration of Software) [11] was applied to assess the current state of the CI/CD pipeline across the project. The EMFIS questionnaire was transformed into an online form, and employees of u-blox who regularly use CI pipelines were invited to participate in the survey. The full questionnaire used in this study is listed in **Appendix:A**.

A total of 34 participants completed the survey: 26 developers and 8 enablers. Enablers comprised roles responsible for processes, tools, and test environments, including a small number of Team Leads and Product Owners. The twelve EMFIS factors are organized under four themes: Activity Planning and Execution, System Thinking, Speed, and Confidence Through Test Activities. Each factor was rated on a Likert scale from 1 (*Major impediment to frequent integration*) to 5 (*Works really well for frequent integration*) [11].

5.1 Survey result

Table 5.1 presents the mean scores for each factor, separated by developer and enabler groups. The EMFIS model is developer-centric; accordingly, developer ratings are treated as the primary signal for identifying impediments [11]. A mean score

≤ 2 is interpreted as a critical impediment, ≤ 3 as a notable concern , and a difference > 0.5 between enabler and developer means signals a perception gap requiring investigation.

Factor	**Developers' Assessment**	**Enablers' Assessment**
Activity Planning and Execution		
Work breakdown	3.54	3.13
Teams and responsibilities	2.92 !	2.88 !
Activity sequencing	2.88 !	2.75 !
System Thinking		
Modular and loosely coupled architecture	3.12	3.25
Developers must think about the complete system	3.00	3.25
Speed		
Tools and processes that are fast and simple ≠	3.12	3.88
Availability of test environments	2.80 !	3.00
Test selection	3.04	2.75 !
Fast feedback from the integration pipeline	3.62	3.38
Confidence Through Test Activities		
Test before merge ≠	3.12	3.63
Regression tests on the mainline	3.16	3.00
Reliability of test environments	3.00	3.00

Note: mean ≥ 3 (satisfactory); ! mean < 3 (concern);
≠ enabler–developer gap > 0.5 (perception mismatch)

Table 5.1: EMFIS evaluation results.

5.2 Analysis of EMFIS assessment

Before presenting the detailed EMFIS assessment, it is important to note that the responses are explicitly perception-based. Respondents rate statements like *"Tools*

and processes that are fast and simple", *"Modular and loosely coupled architecture"*, or *"Availability of test environments"* on a 1–5 Likert scale based on their own interpretation and daily experience.

As a result, the assessment is inherently subjective. Different roles with varying levels of experience may interpret the same question through different lenses. Not everyone interacts with the CI/CD pipeline in the same way or as frequently. Some respondents also mention testing activities outside the CI/CD pipeline, such as live or field tests, which influence how they judge factors like test selection and feedback speed. For these reasons, the EMFIS scores reported in this chapter should be seen as structured perceptions of current CI/CD impediments, rather than as absolute measurements of technical capability.

5.2.1 Activity planning and execution

This theme addresses how the organization structures work, assigns team responsibilities, and coordinates integration activities. Two factors fall below a mean of 3.0 for both groups, indicating that planning and execution are one of the major impediments to frequent integration in this project.

- **Work breakdown:** Work breakdown is the highest-rated factor within this theme (Developers: 3.54 | Enablers: 3.13). However, high Enabler variance (1.55) suggests there is strong disagreement between enablers, but the developer score reflects moderate satisfaction at the individual task level. One enabler, however, explicitly identified this factor as a systemic constraint: *"Work breakdown often leads to long-living feature branches that are not integrated in master fast — but stay alive for months. This is a challenge as teams contribute to their branches, but not to master for a 'very long time'."*

- **Teams and responsibilities:** Both groups agree that this is an impediment, as ownership and coordination gaps emerged in the survey. A DevOps engineer

stated directly: *"code ownership: there should be a specific person responsible for each repo/code"*, while another participant highlighted: *"Some Codeowners are very slow (= days)"*, which delays continuous integration.

A senior architect called for better alignment between teams. A developer shared this concern and noted that teams do not know what the other teams are working on. He added that there is limited visibility into upcoming features and that the lack of clear release notes and implementation details makes it harder for teams to integrate new features effectively.

- **Activity sequencing:** Activity Sequencing received the lowest scores in this theme, with both groups rating it under 3.0. Enablers rate it even lower than developers, suggesting those closest to the coordination infrastructure recognize its inadequacy most. One Software Engineer noted: *"Better organized Initiative concepting to allow for smaller and self-contained epics"*, and pointed out that this is something product owners and architects should address. This feedback suggests that most activity sequencing challenges originate mainly at the planning level rather than during execution. Poor activity sequencing and tight architectural coupling are co-dependent problems, and the lack of clear ownership is creating "integration silos" where long-lived branches accumulate risk.

5.2.2 System thinking

System-level understanding and architectural design play a critical role in determining test scope and the clarity of feedback in CI/CD pipelines. Tight coupling between components tends to slow down pipelines and complicate integration. Both factors scored marginally above 3.0 in both groups, but high enabler variance and qualitative evidence reveal significant bottlenecks and internal disagreement.

- **Modular and loosely coupled architecture:** The moderate mean scores (Developers: 3.12 | Enablers: 3.25) mask the divided assessment as it can be observed in high variance (Developers Var.: 1.31 | Enablers Var: 2.21). There is sharp disagreement within the enabler group about the current architecture. However, one enabler argued for *"A major improvement requires architectural changes in the Firmware and reduction of the complexity"*. While another respondent addresses the issues as *"the system is spread around... to change things you need to go across the project, touching areas that one has no experience of"*. The suggestion came from another senior manager to split the Monorepo and separate repositories for major subsystems for the separation of concerns.

- **Developers must think about the complete system:** The developer score sits exactly at the threshold of 3.0. An embedded software engineer with 5–10 years of experience rated this factor 1 and asked for: *"Good training that explains the complete file system ... what files are there in each folders and which area of the entire system does it belong to"*. An Embedded Software Engineer added: *"Proper Documentation and accessibility to it, from developers that do not work on a specific module."* Even if it might not a immediate priority the need and accessibility of proper internal documentation appeared in developer responses which could greatly improve this factor and coordination among teams.

5.2.3 Speed

Developers tend to commit to the mainline less frequently if it is time-consuming or complicated [11]. Speed evaluates the simplicity of tools, the availability of test resources, and feedback latency. This assessment found the largest expectation–experience gap in this theme, and 2 factors scored below the mean of 3.0.

- **Tools and processes that are fast and simple:** This factor represents the largest perception gap (0.76) in the entire assessment (Enabler 3.88 | Developer 3.12). The high rating and the low variance(0.41) among enablers indicate that they confidently perceive tooling as adequate. Although both groups rated this factor above the concern threshold, developers have a more divided experience (variance 0.99). Developers consistently indicated room for improvement, requesting clearer pass/fail criteria, AI-supported code reviews, faster integration tests, and shortcuts to skip pipelines for non-code changes.

- **Availability of test environments:** The developer's mean is 2.8, indicating a notable concern, while the enabler rated it at just the threshold level. Many respondents pointed out that Hardware-in-the-Loop (HiL) testing is missing from the MR pipeline. A Software Engineer stated: *"No Hardware in Loop tests in the pipeline are critically missing"*. An Embedded SW Engineer rated this factor 1 and commented: *"Increase HiL by increasing targets (FPGA & ASIC), invest in traceability of testing of requirements."*

- **Test selection:** Test Selection presents an inverted pattern: enablers rated it lower than developers. Except for one enabler, all gave it a low score (3 or below). The average developer score is just above 3.0, which points to moderate but not strong confidence in current testing activities. Some engineers also recommended test selection based on code changes and avoiding unnecessary or long-running tests. In summary, test selection should be prioritized and needs a targeted testing effort rather than just running a comprehensive test suite.

- **Fast feedback from the integration pipeline:** Fast Feedback is the highest-rated factor in this theme (Developers: 3.62 | Enablers: 3.38). The high developer score reflects general satisfaction, except one developer said the

CI pipeline should complete in less than 10 minutes. However qualitative responses suggested that developers would trade faster pipelines with more effective pipelines with more tests, as confidence in feedback depends on relevance, traceability, coverage quality, and clarity of validation outcomes rather than speed alone.

5.2.4 Confidence through test activities

Confidence is built through appropriate testing before merging and ensuring mainline reliability. All three factors cluster around a mean of 3.0–3.16 for developers, with a significant perception gap on Test Before Merge.

- **Test before merge:** Test Before Merge presents a gap of +0.51. Enablers rated higher for pre-merge testing, while developers are measurably less confident. The qualitative evidence identifies the merge train as the critical point. An Embedded Developer stated: *"Sometimes an MR is all ready and approved, but then only when you try to merge, it discovers additional issues, which then requires another whole round of review"*. An Embedded Developer corroborated this: *"It does the job well; most issues come from the merge train, which can be moved to an earlier stage in many cases."*.

- **Regression tests on the mainline:** Both groups rate Regression Tests similarly (Developers: 3.16 | Enablers: 3.00). Broadly, the response indicates the pipeline is good for regressions, however there is a strong call for increasing code coverage and adding new tests in the pipeline. Additionally, a software engineer proposed *"would be great if the CI pipeline would run (more) component tests"*.

- **Reliability of test environments:** Both groups' average is 3.0, which means the test environment is functional but not robust. One key enabler noted "Re-

quirement traceability and test management" as the primary needed improvement, and this is agreed upon by several other engineers. This links reliability and confidence to the ability to interpret failures and establish credible evidence chains.

A summary of the findings from the EMFIS assessment is presented in Result Chapter (Section 7.2). These findings also influence the selection and design of the CI/CD optimization strategies presented in Chapter 6.

6 CI/CD optimization strategies

This chapter examines the optimization strategies applied to improve CI/CD efficiency in a large embedded software project. In line with RQ3, it evaluates how concrete CI/CD restructuring strategies can produce measurable improvements without requiring a full redesign of the software product or its test framework.

6.1 Overview of optimization approach

To address RQ3, several optimization strategies were implemented on the project's CI/CD pipeline. Each strategy targeted a specific bottleneck identified through prior analysis of pipeline execution data and developer feedback collected from the RQ2-related survey (see Chapter 5). The strategies were applied incrementally to production pipelines in two separate branches. The branches collectively handled around 100 build targets across 20 platform families. The evaluation compared the restructured pipeline against the baseline configuration before optimization using pipeline duration, job counts, and resource utilization as primary indicators.

6.1.1 Gitlab native templates

The baseline pipeline used Ansible playbook with Jinja2 templates to generate CI job definitions for each platform family. Each platform had a variable file listing targets and their job types, and a Jinja2 template iterated over these to emit individual YAML job blocks. While the Ansible variable files were the intended source of truth,

the generated YAML was also committed to the repository. This meant a developer could modify the generated output directly — adding or adjusting jobs without touching the variables — causing configuration drift silently over time. While the resulting jobs ran in parallel, they produced a "flat" and cluttered pipeline graph. With the total pipeline exceeding 250 jobs, this structure became a visual burden.

The optimization replaced this generation pipeline with GitLab's native *spec/inputs* templating mechanism combined with *matrix-based* jobs. A single parameterized template defined all job types — build, lint, and test jobs as composable blocks controlled by boolean input parameters. Listing 6.1 shows the abstracted matrix-based build template used in the optimized design.

```
platform-build:
  extends: .build
  parallel:
    matrix: $[[ inputs.build_matrix ]]
  variable:
    Build_Options: "$[[ inputs.build_options ]]"
  needs:
    - job: $[[ inputs.platform ]]-select
```

Listing 6.1: GitLab parallel matrix template.

A platform definition now passes only the data needed for a shared template, such as the list of targets, optional checks, or build options. At runtime, GitLab expands this definition into N concurrent instances, N being the number of targets. Listing 6.2 illustrates how a platform definition consumes the shared build template. In this structure, one platform definition can expand into several parallel build jobs while still reusing a single shared template.

```
include:
  - local: '/ci/templates/platform_build.yml'
```

```
inputs:
  platform: Platform-A
  Build_Options: "OPTION=1"
  build_matrix:
    - TARGET: [Target_A, Target_B, Target_C, ...]
```

Listing 6.2: Platform definition using the shared build template.

The optimized design established the Platform Definition as the single source of truth, while consolidating execution logic into reusable shared templates. This transformed the pipeline architecture from an imperative generation model into a declarative and functional templating approach. Additionally, the visual structure of the pipeline became substantially cleaner and more hierarchical, improving readability and developer experience.

6.1.2 Test suite parallelization

This optimization strategy focused on decomposing and parallelizing tasks to reduce pipeline duration. In the case study, we found an ideal candidate to apply this change. The project's unittest framework contained 400 package-level test modules with approximately 3000 registered test cases. In the baseline configuration, these tests were run as one monolithic job *unittests*, which sequentially ran all of the tests and generated a code coverage report within the same context. The average job duration for this job was approximately 13 minutes, as shown in the figure 6.1

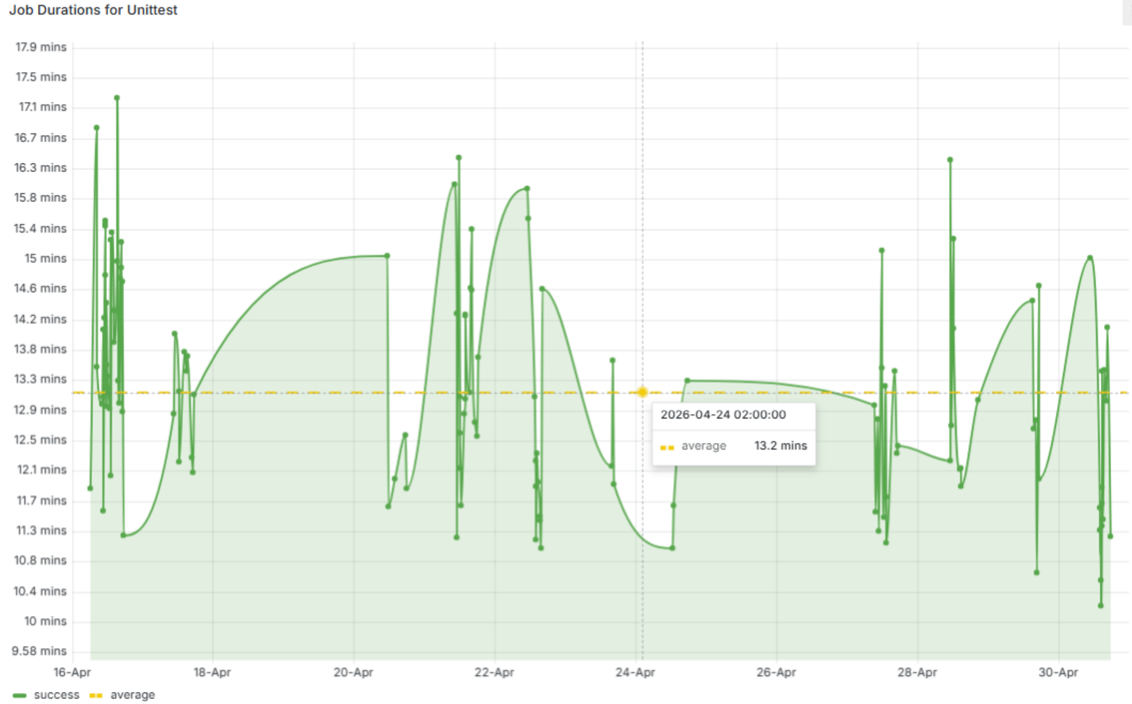

Figure 6.1: Unit-test job duration before optimization.

The monolithic job was replaced with a 27-partition parallel matrix job using GitLab's *parallel: matrix* feature. Each matrix instance executed one logical partition of the suite. This partition boundary followed the existing test-tree structure, so no changes to the test framework were needed. Each partition produced an intermediate coverage report and a downstream job (**unittests-coverage-report**) collects the 27 trace files and merges them for the final report. This preserved the original reporting behavior while distributing tests across independent runners. Conceptually, theoretical runtime improvement can be expressed as the transition from a summation of all test durations to the duration of the longest-running partition:

$$T_{\text{before}} \approx \sum_{i=1}^{n} T_i + T_{\text{report}}$$

after parallelization, it became,

$$T_{\text{after}} \approx \max(T_1, T_2, \ldots, T_n) + T_{\text{aggregate}} + T_{\text{queue}}$$

The gain depends on workload balance across partitions and runner availability. After parallelization, the runtime moved closer to the duration of the slowest partition, plus the cost of report aggregation and any queuing overhead due to runner availability.

6.1.3 Refactoring pre-build workflows

Similar to previous optimization, this strategy also focused on reducing pipeline duration by eliminating redundant jobs, and it targeted the way pre-compilation validation was structured in the CI/CD pipeline. In the baseline workflow, each build target passed through a dedicated pre-build stage before compilation. That stage executed target-specific static validation tasks, such as header-consistency checks, development-only features detection, and then aggregated the outcome into a report. These checks were useful because they could stop faulty targets before full compilation, and it was resource-efficient for on-prem servers. However, this design introduced additional orchestration overhead and complexity. Each target required an extra scheduled job, which increased the total number of pipeline jobs and added another step to the dependency chain. In addition, different targets required different combinations of checks, and these variations had to be explicitly maintained in the CI configuration. As the number of targets grew, the jobs were losing uniformity over time.

As an optimization, the pre-build checks were integrated into the build system, and dedicated build check jobs for each target were removed from the pipeline execution path. The pipeline now executes both activities within a single job. Now the CI layer only triggers a standard build entry point, and the build system decides what to execute. The build step begins by running the relevant validation checks

and proceeds to compilation only if those checks succeed. This preserves the original fail-fast behavior, since static issues are still detected before compilation begins, but it removes the need for a separate scheduling boundary between the checks and build execution.

The main benefit of this change is therefore both operational and structural. Structurally, the CI definition becomes cleaner as there are no separate build-checks jobs per target and it removes the repetitive job definitions. Operationally, less scheduling overhead and a shorter execution graph, which improves pipeline execution time at scale. The detailed improvements are discussed in Result chapter (see Section 7.3.2)

6.1.4 Dynamic tests selection

One gap identified in the EMFIS assessment(See Section 5.2.3) was the lack of effective test selection based on code changes. A custom script called "Test-the-fix" existed to selectively run a subset of unit tests based on relevant code changes. The remaining jobs were gated behind manual selection steps to avoid unnecessary execution. Triggering one of these manual gates would unlock a family of platform builds, validation, and system-level jobs. This approach introduced two key inefficiencies. First, developers had to wait for the initial stage to complete before manually triggering the selection gates, and then rely on their own judgment to choose the relevant jobs. Second, the gating operated at the platform-family level and often triggers a wide set of builds even when the code change affected only a small part of the system.

The optimization strategy was to replace this manual decision step with an automated test selection approach. Similar to the existing “Test-the-fix” script, a new job was introduced to map changed files to the corresponding build configurations. This was implemented using an existing file-filtering script that identified which

source files belonged to each build target. Once the build targets were identified, a separate job would trigger a dynamic child pipeline containing only the relevant targets and their required downstream jobs.

6.2 Technical challenges

In the unittests case, if sufficient runners were not available, some of the expected gain could be lost to queueing delays. Similarly, because the logical partitions were uneven in size, the slowest partition still dominated the completion time of the full validation phase. Third, the partition configuration is statically defined in the CI job (hardcoded matrix). This creates structural coupling between the CI configuration and the test topology, introducing a risk of configuration drift when the test suite evolves, but the partition definitions are not updated accordingly.

In the dynamic test selection case, the manual selection jobs could be overridden to run the affected jobs without creating an additional child pipeline. However, this approach introduced several challenges. The selection was still done at the platform-family level, so all jobs in the affected family were executed instead of only the required ones. Another constraint was that overriding jobs through the GitLab API caused permission errors in downstream jobs. Many of these jobs needed to download container images from another repository, but the API triggered jobs did not have the required access permissions.

We also performed additional experiments in which the results were not as effective as expected. For instance, increasing CPU allocation for build acceleration as discussed in Section 2.6. While the jobs themselves became faster, this introduced scheduling delays in the Kubernetes cluster. For example, when a job requested only one CPU, Kubernetes could usually schedule it immediately. After increasing the request to three CPUs, the scheduler had to wait until enough resources became available on a node. Since many jobs were started in parallel in the same pipeline

stage, this often caused the job to remain queued before execution. As a result, part of the performance gain from faster execution was lost due to queued time. Since the scope of this work primarily focused on CI/CD pipeline and orchestration, further investigation of infrastructure scaling or resource management was not explored in detail.

7 Results

7.1 RQ1: embedded CI/CD: technical and organizational constraints

Our first research question investigates,

> ***"What technical and organizational constraints distinguish CI/CD pipelines for embedded software from traditional software pipelines?"***.

This section answers **RQ1** by identifying the constraints in embedded CI/CD pipelines.

In addressing RQ1, this analysis frames CI/CD challenges not as isolated tooling problems but as arising from a hierarchical constraint structure. In embedded systems, hardware constraints shape firmware architecture, and consequently, the architecture bounds pipeline design and organizational flexibility. Chapter 2 identified a set of recurring technical constraints specific to embedded CI/CD pipelines. To make these distinctions explicit, Table 7.1 was derived by consolidating the constraint categories introduced in Sections 2.4 and contrasting them with characteristics of CI/CD in web and cloud systems. For each dimension, the "embedded" column summarizes the synthesis from the earlier Background (Section 2.4) chapter, while the "traditional" column summarizes the dominant patterns in software-only environments.

Constraint	Embedded Software	Traditional Software	CI/CD Differentiator
Execution Target (Sec. 2.4.1)	Customized or product-specific hardware platform	Virtualized environments (containers/VMs in cloud infrastructure)	Physical hardware cannot be scaled on demand.
Build Variants (Sec. 2.4.2)	Combinatorial explosion of binary configurations.	Typically, a single build artifact is promoted across environments	Testing complexity increases proportionally with variants
Architecture (Sec. 2.4.3)	Monolithic systems with tight hardware-software coupling.	Microservices or loosely coupled modular architectures.	Harder to parallelize or use incremental builds.
Toolchain (Sec. 2.4.4)	Heterogeneous toolchains with cross-compilers and proprietary SDKs.	Standardized cloud-native toolchains (e.g., Docker, Kubernetes)	Vendor-locked and platform-dependent tools.
Validation (Sec. 2.4.5)	Hybrid validation: simulation and physical HiL testing.	Automated testing focusing on software correctness and deployment.	Hybrid Validation is expensive and HiL is limited

Table 7.1: Comparison of technical constraints in embedded and traditional CI/CD systems.

Before presenting the organizational constraints in Table 7.2, it is important to note that many of these challenges are not exclusive to embedded software development. Traditional software systems can face similar issues, including team silos, compliance requirements and resistance to process change. However, in web and cloud environments, many of these constraints have been reduced through widely

adopted engineering practices such as cross-functional DevOps teams and higher levels of automation [1]. The comparison, therefore, does not suggest that traditional software systems are free from organizational constraints, but rather that these challenges are often less severe or better mitigated. In embedded environments, hardware dependencies, legacy product lifecycles, and regulatory demands tend to reinforce the same constraints [4].

Constraint	**Embedded Software**	**Traditional Software**	**CI/CD Differentiator**
Team Structure (Sec. 2.5.1)	Fragmented hardware and software teams.	Cross-functional teams organized around services/products.	Team silos creating coordination and communication gaps and slow integration
Methodology (Sec. 2.5.2)	Stage-gate, waterfall legacy mindset.	Continuous agile cycles.	Limited continuous integration practices
Compliance Gate (Sec. 2.5.3)	Safety-critical standards, traceability, security, and regulatory compliance (e.g., ISO 26262)	Primarily security and regulatory compliance (GDPR/-SOC2).	Embedded systems have different regulations and rigid process gates.
Branching (Sec. 2.5.4)	Long-lived variant and legacy support branches.	Short-lived branches with frequent integration and easy rollback.	Older variants require long-term support.

Table 7.2: Comparison of organizational constraints in embedded and traditional CI/CD systems.

7.2 RQ2: summary of expectation–reality gaps

A more detailed analysis of the EMFIS assessment is presented in Chapter 5. This section provides a summary of the main findings relevant to RQ2:

> ***Where are the gaps between developer expectations and current experience in CI/CD pipelines, and which factors should be prioritized to enable more continuous integration?***

The EMFIS survey results (Table 5.1) indicate that overall CI/CD process was generally perceived as functional but still improvable, with most average scores clustering around 3 on a 5-point scale. However, few recurring constraints and perception gaps emerged across both quantitative and qualitative responses.

7.2.1 Expectation–reality gaps

The clearest expectation–reality gaps were observed in **Tools and processes that are fast and simple** and **Test before merge**, where the difference between developer and enabler assessments exceeded > 0.5 points. For **Tools and processes**, enablers rated the current workflow considerably higher (3.88) than developers (3.12), indicating that developers experience more friction in the day-to-day CI/CD process than enablers perceive.

Similarly, **Test before merge** showed a notable perception gap **(Developers: 3.12, Enablers: 3.63)**, suggesting different expectations regarding the effectiveness of pre-merge tests and processes. The survey also captured developer frustration with the current merge process, where integration-related issues may not be detected early in the pipeline, resulting in additional review cycles. Although these factors exhibited the largest perception gaps, their overall scores remained above the threshold for notable concern. This indicates that the primary issue is not the absence of these capabilities, but rather differences in how different roles perceive

the effectiveness and usability of the system.

7.2.2 Primary constraints: planning and environment

The most prominent constraints were related to Activity Planning and Execution. The lowest-rated factors shared by both developers and enablers were **Teams and responsibilities** and **Activity sequencing**, both scoring below 3.0. These results indicate coordination challenges within the development process, particularly around ownership, work breakdown, and synchronization between teams.

Developers also identified **Availability of test environments** as a notable concern (2.80), reflecting difficulties in accessing reliable HiL during pre-merge development and testing. In contrast, enablers rated **Test selection** lower than developers, suggesting concerns regarding the efficiency or relevance of the current testing strategy.

7.2.3 Prioritized improvement areas

Overall, the EMFIS results for RQ2 suggest that the following areas should be prioritized to enable more continuous integration in the studied project:

- Clarifying teams and responsibilities and improving activity sequencing, to reduce long-lived branches and integration silos.
- Expanding and stabilizing test environments, especially HiL in or near the Merge-Request pipeline.
- Improving test selection and classification, focusing on relevance and code change-based selection rather than volume, to balance fast feedback with confidence.
- Reducing developer-perceived friction in tools and processes, through clearer

pass/fail criteria, skip pipelines for non-code changes, and possibly additional automation support, thereby closing the perception gap with enabler.

7.3 RQ3: pipeline improvements

Several optimization strategies were evaluated to address RQ3:

> ***Which pipeline improvement strategies lead to measurable gains in CI/CD performance in the embedded case study?***

This section discusses the outcomes, observed improvements, and practical limitations of the implemented strategies.

7.3.1 Unit test optimization improvement

As strategies adopted in Section 6.1.2, the overall pipeline execution time, feedback latency, and maintainability improved in multiple dimensions. The monolithic `unittests-full` job, which previously executed the entire test suite sequentially, was replaced by 27 parallel matrix instances, each responsible for executing a single partition. The partitions improve failure visibility, evolving from a single pass/fail outcome into 27 independent results. As a result, failures within a specific package became immediately visible without requiring the full test suite to complete execution.

The achievable performance improvement, however, remained bounded by the heaviest partition, which had the longest execution time, and by runner scheduling overhead caused by the additional 27 execution pods. Overall, the optimization reduced the total pipeline execution time. Figure 7.1 illustrates the improvement observed in Branch pipelines after introducing the parallel unit-test matrix.

The red point (●) in the figure marks the point at which the changes were introduced. It can be observed that the average duration of the Branch pipeline

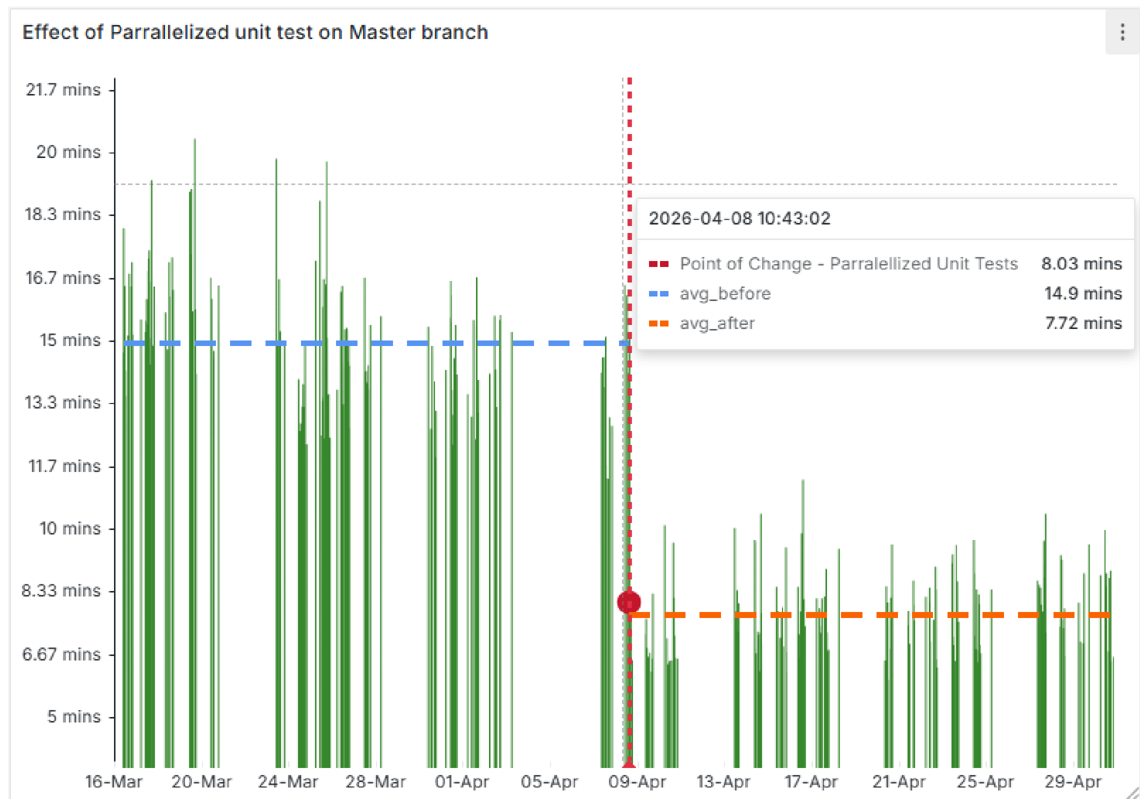

Figure 7.1: Impact of unit-test parallelization on master branch pipeline duration.

decreased from approximately 15 minutes to around 8 minutes. The improvement at the job level can be compared against Figure 6.1, where the original monolithic unit test job had an average completion time of 13 minutes. After parallelization, the partitioned unit test jobs completed within approximately 6 minutes, as shown in Figure 7.2.

It should be noted that this observation applies only to the Branch pipeline. The reduction in overall duration in the Merge Request and Merge Train pipelines was minimal, as other long-running jobs continued to dominate the total pipeline duration. Furthermore, the measurement presented in Figure 7.2 represents a single pipeline execution. In practice, the average duration varies by approximately ±2 minutes depending on runner availability.

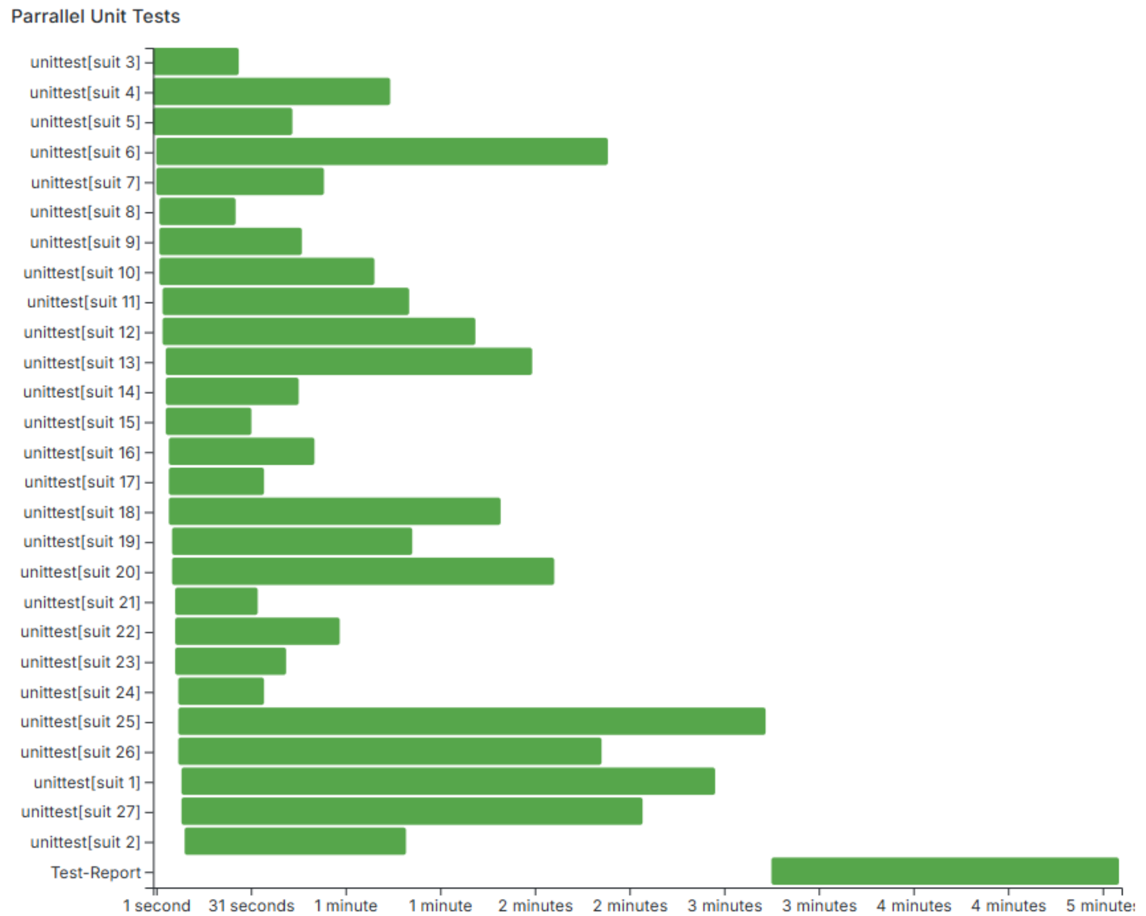

Figure 7.2: Completion time of parallelized unit tests.

7.3.2 Pre-build workflows improvement

The primary benefit of refactoring pre-build workflows are reduction in pipeline complexity and scheduling overhead. The number of pre-build jobs was proportional to the number of build targets in the pipelines and the strategy removed a large number of standalone validation jobs. In one production branch where the optimization was applied, the total number of jobs was reduced by approximately 16%. Running validation and compilation in the same job eliminated repeated environment setup and teardown between stages, reducing redundant initialization overhead. It also impacted resource availability on shared runners. From a maintainability perspective, large amounts of duplicated CI configuration were removed and replaced with

a centralized build-system-based approach.

Figure 7.3 illustrates the shift in cumulative CPU execution time across pipeline stages following the pre-build refactoring. As discussed in Section 6.1.3, the checks were consolidated into the build jobs and predictably CPU execution time increased in the build stages. However, the most significant improvement is observed in the prebuild stage, where execution time is reduced by nearly half due to the removal of standalone validation jobs.

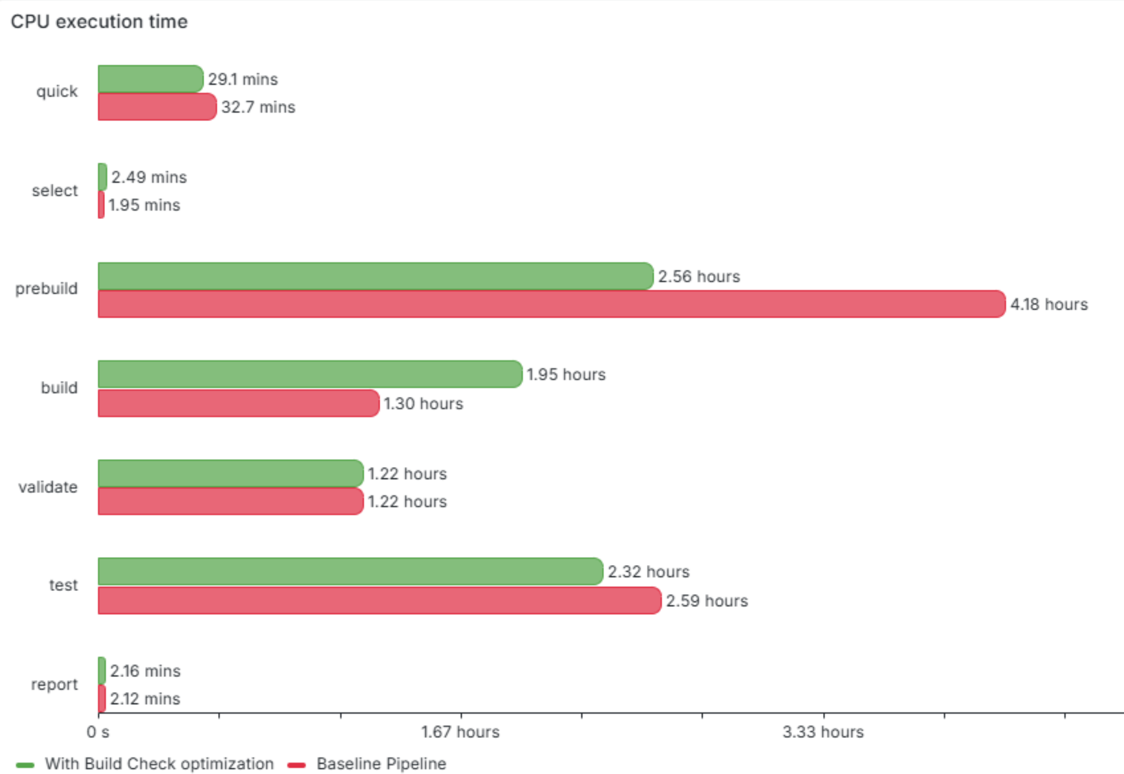

Figure 7.3: Impact of build-check optimization on per-stage CPU execution time.

Note that Figure 7.3 displays the cumulative CPU execution time per stage, rather than the actual duration of the pipeline. In practice, the difference in total pipeline duration between the two pipelines is negligible. The structural reasons for this are discussed in Discussion chapter (See Section 8.1). Despite no overall runtime gain, the pipeline is more efficient. It achieves the same results with fewer jobs and lower total CPU usage while simplifying the CI configuration.

7.3.3 Benefit of templating strategy

The transition from a generated pipeline model to native templating fundamentally reduces one of functional abstraction. In the generated approach, every platform independently declared identical job structures—repeating the same extends:, needs:, and artifacts: blocks for every target—which violated the DRY (Don't Repeat Yourself) principle. With native templating, the job structure is defined once. Platforms differ only in their inputs: which targets to build, which optional steps to enable, and what resource allocation to request. Adding a new platform is a single include: block with input parameters. This modularity ensures structural updates are inherited instantly by all targets without any regeneration step.

Metric	**Baseline (Ansible/Jinja2)**	**Optimized (Native Templates)**
Platform configuration files	11 generated files 9 Ansible variable files	1 shared template and 1 composition file
Pipeline YAML lines	∼2400 generated lines	∼500 authored lines
Structural pipeline changes	Modify Jinja2 templates → regenerate yaml script	Edit shared template once

Table 7.3: Quantified impact of native CI/CD templates.

Moreover, incorporating GitLab matrix parallelization together with native templates created a more maintainable solution compared to the previous Ansible-based generation approach. By consolidating pipeline configuration directly within GitLab CI, the system removed the additional overhead of maintaining an external generator tool. From a maintainability perspective, the total amount of authored CI configuration was reduced substantially, as shown in Table 7.3. In addition, the resulting pipeline structure became more organized, since targets belonging to the same platform family were visually grouped together within the pipeline graph.

7.3.4 Dynamic test selection

To evaluate the Dynamic Test Selection strategy, the approach was tested on five merge requests created by other developers that modified at least one firmware build-related file. The merge requests were selected randomly to represent different scopes of changes. For each merge request, the selection script generated a dynamic child pipeline containing only the affected build targets and their required downstream jobs, while the parent pipeline remained responsible for lightweight pre-compilation checks such as static analysis and configuration validation. Across the evaluated merge requests, the generated child pipelines executed 7, 12, 37, 40, and 59 jobs respectively. In total, the experimental setup was executed 15 times to validate the consistency of the selection mechanism.

The results demonstrate that the approach successfully reduced unnecessary pipeline execution by limiting builds and validation stages to only the affected targets. This shifted the pipeline from manual platform-family selection toward impact-based execution, reducing both manual intervention and redundant CI workload. The same idea works for non-code changes as well, and the pipeline can skip unnecessary builds. This will enable quick feedback and run only the relevant validation test. However, the accuracy and completeness of the job-selection mechanism need more testing and developer approval before production adoption. The results indicate the approach works, but the selection algorithm still has room for improvement, particularly in ensuring that all required downstream jobs are identified for complex cross-component changes.

7.3.5 RQ3 summary

This study demonstrates that restructuring the CI/CD pipeline architecture yields immediate improvements in maintainability and local pipeline performance. Transitioning to data-driven native templates simplified the pipeline structure and reduced

configuration complexity. As discussed in Section 4.2 and further elaborated in Section 8.1, the study found that micro-optimizing individual jobs has a limited impact on overall pipeline runtime unless those jobs are located on the critical path. Instead, meaningful improvements can be achieved through:

- **Optimizing pipeline flow**: Implement DAG Execution, Merge small orchestration jobs, Clean Up the Build Matrix, Compress the End-to-End Critical Path.
- **Configuration changes**: Adopt Native Templating, Refactor CI Definition, Remove CI Anti-Patterns, Migrate tools and platform.
- **Testing Strategies**: Dynamic Test Selection, Tiered test strategy, Leverage Incremental Builds and Caching.
- **Infrastructure Tuning and Observability**: Standard Container Images, Scale Runners and Resource Classes, Add Observability and Metrics.

7.4 RQ4: architectural and technological impact

This section presents a holistic summary of the thesis findings in order to answer RQ4.

> ***How do architectural and technological changes impact the performance, scalability, and maintainability of embedded CI/CD pipelines?***

The findings show that CI/CD pipeline effectiveness cannot be improved through isolated fixes alone. Developers indicate they would actually trade a faster pipeline for more effective pipelines with higher-quality tests, better traceability, and clearer pass/fail criteria (see Section 5.2.3). Instead, CI/CD performance is intrinsically tied to a combination of technical and organizational factors, as shown in Table 7.4.

Factor	Supporting Sections
Compute and Hardware resources	2.4.1, 2.4.5
Platform Tools	2.3, 2.4.4, 5.2.3
System Constraints	2.4.2, 2.4.3, 4.1.1, 4.2, 5.2.2
Pipeline configuration complexity	4.2, 4.1.2, 5.2.4 , 6.2
Organizational Integration Practices	2.5, 5.2.1

Table 7.4: Key factors affecting CI/CD pipeline performance.

Mårtensson et al. argue that breaking down large systems – software systems and organizational structure – into smaller pieces is a key enabler for continuous integration at scale [10]. However, refactoring the core firmware architecture requires a multi-year effort, and it requires substantial budget and human effort to mitigate years of technical debt [4]. As discussed in Sections 2.4.3 and 4.2, it is not always feasible in embedded contexts, particularly when the system has limited memory and computing resources. Although recent AI-assisted workflows can significantly reduce the cost and human effort of exploring modularization options for monolithic architectures (See Section 8.2).

Given these constraints, the architectural and technological changes investigated in this thesis focus on the CI/CD pipeline rather than the firmware architecture itself. As shown in Section 7.3.5, optimizing CI/CD configuration is comparatively low effort and there are multiple ways to approach it. In this project, the organization had already standardized on GitLab and AWS as a consolidated toolchain for source control, orchestration, and compute resources. This consolidation eliminated cross-tool dependencies and synchronization overhead and provided a stable foundation for further pipeline-level optimizations. The impact of the implemented changes can be discussed along three primary dimensions:

- **Performance**: Technological changes such as test-suite partitioning and parallelization produced clear local performance gains. For example, the branch pipeline duration was reduced by roughly half after optimizing unit tests (Sec-

tion 7.3.1). However, the DAG-based execution model means that the pipeline time is dominated by long-running hardware or simulation-based tests (Section 8.1). This thesis shows ways to shorten the critical path by removing redundant build-check jobs and identifying jobs that can run in parallel (Section 4.2, 6.1.3).

- **Scalability:** Changes like Dynamic Test Selection and Native Templating strategy directly address the combinatorial complexity of build variants. Native templating makes it much easier to add or remove variants than before(Section 7.3.3). Experiments with dynamic child pipelines show that only the build targets and downstream jobs affected by a change need to be executed, which reduces redundant workload while preserving coverage (Section 7.3.4). At the same time, simply infrastructure scaling alone is not sufficient to guarantee performance improvements (Section 6.2).

- **Maintainability:** The strongest observable impact is on the maintainability of the CI/CD configuration. Moving to native GitLab templating from external Ansible playbook generation is another step towards consolidating tools and processes. This transition effectively reduces configuration drift, minimizes total lines of code, and eliminates the need for manual regeneration steps. Additionally, utilizing the parallel matrix feature allows for the condensed grouping of jobs within the same family. However, implementing this strategy can be a challenging effort if the jobs are not uniform (Section 7.3.3).

8 Discussion

Before the optimization work in this thesis started, the baseline pipeline had already been modernized. Moving from an on-premise Jenkins setup to an AWS cloud-native environment with Kubernetes solved most infrastructure scaling issues. Shared caching helped reuse artifacts, and using standard Docker image definitions made environments more consistent across different targets.

Despite these updates, challenges remained in pipeline orchestration, configuration redundancy, managing heterogeneous hardware variants, test accuracy, and maintaining CI definitions over time. Therefore, the main focus of the optimization work in this thesis was to improve pipeline maintainability, reduce unnecessary execution, and faster validation workflows on top of an already elastic cloud-native infrastructure.

8.1 CI/CD optimization constraints

In this project, job-level optimization has a limited impact unless evaluated in the context of the pipeline's dependency graph. In traditional stage-ordered models, each stage acts as a barrier; downstream work must wait for every job in the current stage to finish. In this case, accelerating individual jobs directly reduces stage completion time and consequently, total pipeline duration.

However, the studied pipeline is primarily DAG-driven rather than traditional stage-gated. Most jobs use explicit needs relationships and are allowed to start im-

mediately after their required dependencies are completed. This design enables high cross-stage parallelism and reduces unnecessary waiting between stages. Because of this execution model, the total pipeline duration was determined mainly by the longest critical execution path rather than by the average duration of all jobs. The case project shows that this critical path originates from a combination of build and test chains inherent to the system domain. The path includes the slowest test partitions, particularly long-running on-device and integration tests whose durations are constrained by physical hardware behavior and the need to execute timing-sensitive, sequential scenarios such as GNSS acquisition. As a result, micro-optimizing individual jobs produces weak global gains. If an optimized job was not part of the critical path, the effect on total execution time remained limited.

This became one of the main optimization constraints observed during the study. Pipeline performance was ultimately bounded by the longest-running dependency chain and runner scheduling latency. The pipeline already exploits DAG parallelism effectively; future performance improvements should focus on the job dependency structure and critical-path compression rather than on isolated job optimization.

8.2 Implications of AI-assisted development

In this work, various AI models supported coding, refactoring, and technical writing tasks (see Appendix B). The experience suggests that AI-assisted development is beginning to change the cost, speed, and accessibility of large-scale software modification. However, current Large Language Models (LLMs) are still constrained by context window limits and may misinterpret domain-specific terminology or implicit system behavior without sufficient project context.

The architectural refactoring challenge discussed in Section 7.4 remains substantial, but the nature of the approach is beginning to change. Historically, restructuring monolithic firmware or refactoring complex CI/CD configurations required

extensive manual analysis by experts with deep system knowledge and months of coding effort. With AI-assisted workflows - such as iterative prompting and rapid prototyping- a less experienced developer can now explore alternatives, generate proof-of-concept, and summarize large parts of a codebase much more quickly than before, provided that review practices are in place. Essentially, the cost of producing and iterating on meaningful code changes has decreased, even though architectural decisions and verification responsibility remain with humans.

However, AI does not remove the fundamental bottlenecks identified in this study. The key constraints identified in RQ1 (Section 7.1) - HiL dependent validation, organizational integration practices, and build variant complexity - still remain as structural challenges. AI models do not eliminate the need for engineering judgment, particularly in safety-critical or constrained embedded environments. Instead, the effort and time shift toward verification and architectural decision-making rather than manual implementation. In the context of this thesis, AI-assisted development is an enabler and accelerator for the heavy lifting required to modernize legacy CI/CD infrastructure, provided it is supported by rigorous human review and testing.

8.3 Validity and limitations

The study has several limitations that affect the validity of its findings. First, the scope is limited to a single company and one embedded firmware product line. This restricts external validity because the observed constraints and optimization effects may not fully apply to other domains or organizations.

Second, the industrial environment imposes confidentiality constraints that restrict the level of detail that can be reported about specific configurations, variants, or hardware setups.

8.4 Conclusion

This thesis investigated how CI/CD pipelines can be optimized for large-scale embedded firmware projects operating under hardware constraints, multi-variant build complexity, and organizational impediments. Even though the studied system already employed modern DevOps practices and state-of-the-art CI/CD technologies, substantial friction remained in both developer experience and execution efficiency. The study examined both the technical and human factors that limit continuous integration in embedded environments.

The first research question established how embedded CI/CD pipelines differ from traditional software pipelines. The primary constraint is that validation depends on physical hardware, which cannot be scaled like cloud resources, and numerous product variants increase configuration complexity. Monolithic firmware architectures, heterogeneous toolchains, and lengthy validation cycles also reinforce these limitations. Organizational factors such as team silos, waterfall-style workflows, and long-lived branches further slow down integration cycles.

According to the EMFIS survey, developers generally perceived the studied pipeline as functional, but identified several pain points. Key impediments were found in processes and tools, test-before-merge practices, and test selection. Guided by these constraints, the implemented optimizations showed meaningful improvements through targeted pipeline-level changes. Native templating reduced configuration complexity, dynamic test selection reduced redundant execution, and pipeline restructuring improved local performance. However, micro-optimizing individual jobs had a limited effect on overall pipeline duration due to dependency chain structures.

Finally, the last research question showed that embedded CI/CD optimization is a multidimensional challenge, as summarized in Table 7.4. The findings demonstrate that meaningful improvements cannot be achieved through isolated fixes alone, since

pipeline behavior is tightly connected to the identified factors. In large-scale embedded environments, CI/CD pipelines evolve together with the product architecture, testing strategy, and organizational processes. Consequently, CI/CD optimization should not be viewed as a one-time migration effort, but rather as a continuous process of measurement, refinement, and architectural adaptation.

Overall, this thesis contributes both analytical and practical insights into embedded CI/CD engineering. The study demonstrates that while embedded systems have constraints that cannot be completely eliminated, careful architectural and organizational improvements can significantly improve pipeline scalability, maintainability, and performance.

References

[1] L. E. Lwakatare et al., “Towards DevOps in the Embedded Systems Domain: Why is It So Hard?”, in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, IEEE, Jan. 2016, pp. 5437–5446. DOI: 10.1109/HICSS.2016.671. [Online]. Available: http://ieeexplore.ieee.org/document/7427859/.

[2] M. Fowler, *Continuous integration*, Thoughtworks Blog, Available at https://www.martinfowler.com/articles/continuousIntegration.html, accessed 12 November 2025, 2024.

[3] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, “Testing embedded software: A survey of the literature”, *Information and Software Technology*, vol. 104, pp. 14–45, Dec. 2018, ISSN: 09505849. DOI: 10.1016/j.infsof.2018.06.016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584918301265.

[4] M. Becker, R. Rabiser, and G. Botterweck, “Not quite there yet: Remaining challenges in systems and software product line engineering as perceived by industry practitioners”, in *Proceedings of the 28th ACM International Systems and Software Product Line Conference*, ser. SPLC ’24, New York, NY, USA: Association for Computing Machinery, Sep. 2, 2024, pp. 179–190, ISBN: 979-8-4007-0593-9. DOI: 10.1145/3646548.3672587. [Online]. Available: https://dl.acm.org/doi/10.1145/3646548.3672587.

[5] F. Zampetti, D. Tamburri, S. Panichella, A. Panichella, G. Canfora, and M. Di Penta, "Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study", *ACM Trans. Softw. Eng. Methodol.*, vol. 32, 73:1–73:44, Apr. 26, 2023, ISSN: 1049-331X. DOI: 10.1145/3571854. [Online]. Available: https://dl.acm.org/doi/10.1145/3571854.

[6] P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, "A Distributed Test System Architecture for Open-source IoT Software", in *Proceedings of the 2015 Workshop on IoT Challenges in Mobile and Industrial Systems*, Florence Italy: ACM, May 18, 2015, pp. 43–48. DOI: 10.1145/2753476.2753481. [Online]. Available: https://dl.acm.org/doi/10.1145/2753476.2753481.

[7] M. J. Page et al., "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews", *BMJ*, vol. 372, n71, Mar. 29, 2021, ISSN: 1756-1833. DOI: 10.1136/bmj.n71. PMID: 33782057. [Online]. Available: https://www.bmj.com/content/372/bmj.n71.

[8] H. Fu, S. Eldh, K. Wiklund, A. Ermedahl, and C. Artho, "Prevalence of continuous integration failures in industrial systems with hardware-in-the-loop testing", in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2022, pp. 61–66. DOI: 10.1109/ISSREW55968.2022.00040. [Online]. Available: https://ieeexplore.ieee.org/document/9985087.

[9] Í. Gautsson and Þ. Hafsteinsdóttir, "Continuous Integration in Component-Based Embedded Software Development: Problems and Causes", Jun. 2017.

[10] T. Martensson, "Continuous integration and delivery applied to large-scale software-intensive embedded systems", English, Ph.D. dissertation, University of Groningen, 2019, ISBN: 978-94-034-1399-0.

[11] T. Mårtensson, D. Ståhl, and J. Bosch, "The EMFIS Model — Enable More Frequent Integration of Software", in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2017, pp. 10–17. DOI: 10.1109/SEAA.2017.31. [Online]. Available: https://ieeexplore.ieee.org/document/8051321.

[12] N. Fonseca, J. Paulo Fernandes, M. Pires, and S. Melo de Sousa, "PACE: A DSL-based Approach to Manage Complex Build Pipelines", in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 43–50. DOI: 10.1109/SEAA51224.2020.00018. [Online]. Available: https://ieeexplore.ieee.org/document/9226318/.

[13] F. Segatz, "Continuous integration for embedded software with modular firmware architecture", M.S. thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2023, p. 87.

[14] E. Agorogiannis, K. Petrakis, G. Antonopoulos, T. Anagnostopoulos, N. Grigoropoulos, and Z. Benomar, "MAGNET: An optimized DevOps framework for efficient software integration and orchestration in the IoT-Edge-Cloud Continuum", in *2025 6th International Conference in Electronic Engineering & Information Technology (EEITE)*, Jun. 2025, pp. 1–6. DOI: 10.1109/EEITE65381.2025.11166038. [Online]. Available: https://ieeexplore.ieee.org/document/11166038.

[15] H. Yasar and S. E. Teplov, "DevSecOps In Embedded Systems: An Empirical Study Of Past Literature", in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ser. ARES '22, New York, NY, USA: Association for Computing Machinery, Aug. 23, 2022, pp. 1–6, ISBN: 978-1-4503-9670-7. DOI: 10.1145/3538969.3544451. [Online]. Available: https://dl.acm.org/doi/10.1145/3538969.3544451.

[16] P. Patchkaew, H. Kirdpakdee, C. Kidwat, and A. Sukstrienwong, "Adaptive CI/CD: A Flexible Architecture for Software Development", in *2024 8th International Conference on Information Technology (InCIT)*, Nov. 2024, pp. 763–768. DOI: 10.1109/InCIT63192.2024.10810646. [Online]. Available: https://ieeexplore.ieee.org/document/10810646/.

[17] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems", *Journal of Systems and Software*, SI: Top Scholars, vol. 83, no. 1, pp. 67–76, Jan. 1, 2010, ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.051. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121209001617.

[18] T. Chatterjee, "Test Lead Time And Cost Improvement Of Automotive Embedded Project - A New Perspective With Automation And CI/CD", in *2024 International Conference on Vehicular Technology and Transportation Systems (ICVTTS)*, vol. 1, Sep. 2024, pp. 1–6. DOI: 10.1109/ICVTTS62812.2024.10763911.

[19] M. Eggert, K. Günther, J. Maletschek, A. Maxiniuc, and A. Mann-Wahrenberg, "In three steps to software product lines: A practical example from the automotive industry", in *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '22, vol. A, New York, NY, USA: Association for Computing Machinery, Sep. 12, 2022, pp. 170–177, ISBN: 978-1-4503-9443-7. DOI: 10.1145/3546932.3547003. [Online]. Available: https://dl.acm.org/doi/10.1145/3546932.3547003.

[20] M. D. O. Farina et al., "Hardware-Independent Embedded Firmware Architecture Framework", en, *Journal of Internet Services and Applications*, vol. 15, no. 1, pp. 14–24, Apr. 2024, ISSN: 1869-0238. DOI: 10.5753/jisa.2024.3634.

Accessed: Jan. 12, 2026. [Online]. Available: https://journals-sol.sbc.org.br/index.php/jisa/article/view/3634.

[21] J. Koivuniemi, "Shortening feedback time in continuous integration environment in large-scale embedded software development with test selection", Pro Gradu -Työ, Univeristy of Oulu, Apr. 2017. [Online]. Available: https://oulurepo.oulu.fi/handle/10024/9370.

[22] T. Mårtensson, D. Ståhl, and J. Bosch, "Continuous Integration Impediments in Large-Scale Industry Projects", in *2017 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, pp. 169–178. DOI: 10.1109/ICSA.2017.11. [Online]. Available: https://ieeexplore.ieee.org/document/7930214.

[23] J. Perdek and V. Vranić, "Fully Automated Software Product Line Evolution With Diverse Artifacts", en, *IEEE Access*, vol. 13, pp. 27 325–27 358, 2025, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2025.3539868. Accessed: Jan. 8, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/10877839/.

[24] K. Telschig, A. Schönberger, and A. Knapp, "A real-time container architecture for dependable distributed embedded applications", in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, Aug. 2018, pp. 1367–1374. DOI: 10.1109/COASE.2018.8560546.

[25] E. Laukkanen, T. O. Lehtinen, J. Itkonen, M. Paasivaara, and C. Lassenius, "Bottom-up Adoption of Continuous Delivery in a Stage-Gate Managed Software Organization", in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16, New York, NY, USA: Association for Computing Machinery, Sep. 2016, pp. 1–10, ISBN: 978-1-4503-4427-2. DOI: 10.1145/2961111.2962608. Accessed:

Jan. 7, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/2961111.2962608.

[26] B. Lima and R. Pinto, "Current Challenges and Future Perspectives in Testing IoT Systems: A Comprehensive Review", *IEEE Sensors Reviews*, vol. 3, pp. 22–47, 2026, ISSN: 2995-7478. DOI: 10.1109/SR.2025.3628264. [Online]. Available: https://ieeexplore.ieee.org/document/11224440/.

[27] V. Debroy, S. Miller, and L. Brimble, "Building lean continuous integration and delivery pipelines by applying DevOps principles: A case study at Varidesk", in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, New York, NY, USA: Association for Computing Machinery, Oct. 26, 2018, pp. 851–856, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3275528. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3275528.

[28] E. Knauss, M. Staron, W. Meding, O. Soder, A. Nilsson, and M. Castell, "Supporting Continuous Integration by Code-Churn Based Test Selection", in *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, Florence: IEEE, May 2015, pp. 19–25, ISBN: 978-1-4673-7067-7. DOI: 10.1109/RCoSE.2015.11. [Online]. Available: https://ieeexplore.ieee.org/document/7167168/.

[29] T. Çıngıl and H. Sözer, "Black-box Test Case Selection by Relating Code Changes with Previously Fixed Defects", in *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '22, New York, NY, USA: Association for Computing Machinery, Jun. 13, 2022, pp. 30–39, ISBN: 978-1-4503-9613-4. DOI: 10.1145/3530019.3530023. [Online]. Available: https://dl.acm.org/doi/10.1145/3530019.3530023.

[30] S. Vöst and S. Wagner, "Trace-based test selection to support continuous integration in the automotive industry", in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, ser. CSED '16, New York, NY, USA: Association for Computing Machinery, May 14, 2016, pp. 34–40, ISBN: 978-1-4503-4157-8. DOI: 10.1145/2896941.2896951. [Online]. Available: https://dl.acm.org/doi/10.1145/2896941.2896951.

[31] A. Arrieta, P. Valle, J. A. Agirre, and G. Sagardui, "Some Seeds Are Strong: Seeding Strategies for Search-based Test Case Selection", *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, 17:1–17:47, Feb. 13, 2023, ISSN: 1049-331X. DOI: 10.1145/3532182. [Online]. Available: https://dl.acm.org/doi/10.1145/3532182.

[32] D. Ståhl, T. Mårtensson, and J. Bosch, "The continuity of continuous integration: Correlations and consequences", *Journal of Systems and Software*, vol. 127, pp. 150–167, May 2017, ISSN: 01641212. DOI: 10.1016/j.jss.2017.02.003. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121217300328.

[33] B. Vogel-Heuser, J. Fischer, S. Feldmann, S. Ulewicz, and S. Rösch, "Modularity and architecture of PLC-based software for automated production Systems: An analysis in industrial companies", *Journal of Systems and Software*, vol. 131, pp. 35–62, Sep. 1, 2017, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.05.051. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217300985.

[34] F. Urdih, T. Theodoropoulos, and U. Zdun, "Architectural Design Decisions and Best Practices for Fast and Efficient CI/CD Pipelines", en, in *Software Architecture*, V. Andrikopoulos, C. Pautasso, N. Ali, J. Soldani, and X. Xu, Eds., Cham: Springer Nature Switzerland, 2026, pp. 297–305, ISBN: 978-3-032-02138-0. DOI: 10.1007/978-3-032-02138-0_19.

[35] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, "Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab", in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA: ACM, Nov. 8, 2020, pp. 327–337, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409709. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3409709.

[36] "ISO/IEC/IEEE International Standard–Information technology–DevOps–Building reliable and secure systems including application build, package and deployment", *ISO/IEC/IEEE Std 32675:2022*, pp. 1–94, Sep. 2022. DOI: 10.1109/IEEESTD.2022.9882056. [Online]. Available: https://ieeexplore.ieee.org/document/9882056/.

[37] *CI/CD YAML syntax reference | GitLab Docs*. Accessed: Apr. 9, 2026. [Online]. Available: https://docs.gitlab.com/ci/yaml/.

[38] "QEMU - A generic and open source machine emulator and virtualizer". [Online]. Available: https://www.qemu.org/.

[39] M. E. Conway, "How do committees invent?", *Datamation*, Apr. 1968, Reprinted by permission of Datamation magazine. Copyright 1968, F. D. Thompson Publications, Inc.

[40] "Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements and amending Regulations (EU) No 168/2013 and (EU) No 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act)",

[41] "Manifesto for Agile Software Development". [Online]. Available: https://agilemanifesto.org/.

[42] European Parliament and Council of the European Union. “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”, EUR-Lex. [Online]. Available: http://data.europa.eu/eli/reg/2016/679/oj.

[43] R. B. Johnson and A. J. Onwuegbuzie, “Mixed Methods Research: A Research Paradigm Whose Time Has Come”, *Educational Researcher*, vol. 33, no. 7, pp. 14–26, Oct. 1, 2004, ISSN: 0013-189X. DOI: 10.3102/0013189X033007014. [Online]. Available: https://doi.org/10.3102/0013189X033007014.

Appendix A EMFIS survey questionnaire

This appendix presents the survey instrument used to assess the current state of the CI/CD pipeline. The survey is based on the EMFIS model (Enable More Frequent Integration of Software) [11] and was adapted for embedded firmware development.

The survey evaluates developer expectations and current experiences regarding CI pipeline performance, including speed, feedback, and reliability.

A.1 Survey overview

Format:

- 12 key factors rated on a 1–5 Likert scale.
- Additional open-ended questions for qualitative feedback.

Scale:

- 1 = Major impediment to frequent integration
- 2 = Needs significant improvement
- 3 = Neutral
- 4 = Works reasonably well

- 5 = Works very well for frequent integration

A.2 Respondent information

- Role
- Years of experience:
  - < 2 years
  - 2–5 years
  - 5–10 years
  - > 10 years

A.3 Activity planning and execution

How we plan and coordinate work affects pipeline flow.

- **Work breakdown**: A way of working that supports work breakdown into small pieces that can be delivered to the software mainline. It involves clear directives on whether a commit must include complete functions, documentation, or tests.

How well does our current work breakdown allow you to merge small, independent deltas daily?	1 2 3 4 5

- **Teams and responsibilities** : An organization that supports both working with functional changes and at the same time takes responsibility for the architecture.

Do current team structures provide clear ownership to functional changes and architecture?	1 2 3 4 5

- **Activity sequencing** : Synchronization between the teams in order to optimize the flow of activities when functions and systems are implemented.

How well synchronized are team activities to avoid integration delays?	1 2 3 4 5

Open questions:

- What needs to be improved in activity planning and execution?
- Who is the best driver of this improvement?

A.4 System thinking

Architecture and system understanding determine test scope and feedback clarity. Tight coupling → slow pipelines.

- **Modular and loosely coupled architecture**: A modular architecture with small components, which makes it possible for many teams to work in parallel. Loosely coupled architecture with as few dependencies as possible between the components.

Does the firmware architecture allow you to commit changes without being hindered by cross-module dependencies?	1 2 3 4 5

- **Developers must think about the complete system**: Developers understand the functions and design of the whole system, and not just their own sub-system.

How confident are you that you understand the full system impact of your changes before you commit?	1 2 3 4 5

Open questions:

- What needs to be improved in System Thinking to get effective CI pipeline?
- Who is the best driver of this improvement?

A.5 Speed

- **Tools and processes that are fast and simple**: All tools and processes related to integration of the software are fast and simple to use.

How fast and simple are our build, test, and integration tools to use daily?	1 2 3 4 5

- **Availability of test environments**: Developers can get access to sufficient test resources for their test activities before committing to the mainline.

How available are hardware, simulators, and HIL when developers need them?	1 2 3 4 5

- **Test selection**: Strategies are established to define which tests run on an event basis, fixed schedules(e.g. Nightly), and release testing.

How effectively does our automated test selection run only the relevant tests for your specific change?	1 2 3 4 5

- **Fast feedback from the integration pipeline**: The developer gets fast feedback when software is committed to the mainline, signaling any deviations or problems.

How quickly do you receive actionable feedback after commits?	1 2 3 4 5

Open questions:

- What needs to be improved in Speed to get faster pipeline feedback?
- Who is the best driver of this improvement?

A.6 Confidence through test activities

- **Test before merge**: The test activities that are performed by the developers before merging code to the mainline are appropriate and conducive.

How effective are pre-merge tests available to developers?	1 2 3 4 5

- **Regression tests on the mainline**: The regression tests on the mainline include a mix of test activities of varying scope and test coverage that protect mainline quality and stability.

How effectively do our "Regression tests" support system stability and frequent integration?	1 2 3 4 5

- **Reliability of test environments**: The test environments have idempotent behavior and do fully represent the production environment.

How stable and reproducible are our test environments?	1 2 3 4 5

Open questions:

- What needs to be improved in Confidence through test activities to get faster pipeline feedback?
- Who is the best driver of this improvement?

A.7 Final remarks

- Based on your experience. how well does the CI pipeline perform particularly in terms of detecting issues, maintaining firmware quality, and supporting your productivity?
- If you could improve one aspect of the CI pipelines to better meet your expectations, what would it be?
- Additional comments.

Appendix B AI usage declaration

In accordance with the University of Turku guidelines on the responsible and transparent use of artificial intelligence in studying and research, the use of AI-based tools in this thesis is documented in Table B.1.

AI Tools	Stage of work	Use in thesis work
ChatGPT (GPT-5.3-mini)	Drafting and revision	Used for iterative drafting, rewriting, summarization, and refinement of technical and academic text. Assisted in developing the thesis structure, restructuring explanations, condensing verbose sections, maintaining consistency, and academic tone throughout the thesis writing process.
Gemini (Gemini 3)	Drafting and revision	Used similarly to ChatGPT throughout the writing process.
GitHub Copilot (Claude, GPT-Codex) in VS Code	Code study and implementation	Used for explaining code, debugging, refactoring suggestions, generating test ideas, quick validation of implementation ideas, and assisting with CI/CD scripting during experimentation and development.
Grammarly	Proofreading and final editing	Used for grammar correction, punctuation checking, readability improvement, and sentence-level proofreading of the thesis manuscript.

Table B.1: Summary of AI tool usage in this thesis

The author confirms full responsibility for the content, accuracy, originality, and integrity of this thesis. All research design, analysis, and conclusions are the result of independent academic work and critical judgment.

AI-based tools were used solely as supporting tools to assist with language re-

finement, structural improvement, coding support, and technical clarifications. In particular, AI tools were not used to autonomously generate experimental results, measurements, or final research findings. All AI-assisted outputs were reviewed, verified, and manually validated by the author before inclusion in the thesis.