Continuous On-Commit Certification Testing of Railway Control Software: An Experience Report

Jacopo Maltagliati
University of Milano-Bicocca
Dept. of Informatics, Systems and Communication
Milan, Italy
jacopo.maltagliati@unimib.it

Salvatore De Simone
Rete Ferroviaria Italiana
R&D Systems Development
Naples, Italy
sa.desimone@rfi.it

Giovanni Denaro
University of Milano-Bicocca
Dept. of Informatics, Systems and Communication
Milan, Italy
giovanni.denaro@unimib.it

Abstract

The regulatory standards of railway control software impose stringent certification requirements. These constraints, combined with the heterogeneity of the components, the interactive user interfaces, and the proprietary environments that characterize the development processes of this class of software, complicate the adoption of standard practices for continuous integration and continuous testing.

This paper discusses our experience in supporting continuous on-commit certification testing of railway control software in an industrial setting. We describe the challenges that we addressed and the design decisions that we adopted accordingly.

The result is a solution for continuous integration and continuous certification testing that decouples build, orchestration, and artifact management using isolated execution environments. It features a multi-runner model that enables efficient on-commit execution of hundreds of certification test sequences, supporting full automation of test sequences that require interactions with both the simulated environment and the driver. It aggregates the results from multiple, heterogeneous logging sources, facilitating regression testing, failure analysis, and certification reporting.

Overall, our design enables continuous execution of large certification test suites while preserving reproducibility, traceability, and result analysis.

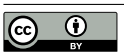
CCS Concepts

• **Software and its engineering** → **Software verification and validation**; **Software testing and debugging**; **Agile software development**.

Keywords

continuous integration, regression testing, railway control software, parallel execution

ACM Reference Format:
Jacopo Maltagliati, Salvatore De Simone, and Giovanni Denaro. 2026. Continuous On-Commit Certification Testing of Railway Control Software: An Experience Report. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3832783.3834532



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834532

1 Introduction

Developing and certifying railway control software requires strict adherence to regulatory standards that mandate coding rules, architectural constraints, reproducible testing, and conformance to detailed test sequences defined by certification authorities. The resulting development practices oftentimes contrast with the assumptions and the support in the mainstream automation tools for continuous integration and continuous testing, making it difficult to directly adopt the same facilities that are well established in other software domains, e.g., the continuous integration and continuous deployment (CI/CD) tooling adopted in agile processes.

In the industrial setting we consider in this paper, this challenge is particularly evident in the development of the European Train Control System (ETCS) [5] onboard software, where certification-driven restrictions shape the build process, system architecture, and testing workflow.

First, a significant portion of the system is developed in model-based fashion with ANSYS SCADE [1], which facilitates the generation of certifiable C code, but whose proprietary compiler is not available in standard containers [3, 12] and depends on licensed executables that cannot be freely replicated.

Next, the system architecture decomposes into several embedded components, executed in distinct processes, possibly with different types of interfaces, spanning from network communication to human interface devices. For instance, two core components are the ETCS subsystem and the Driver Machine Interface (DMI) [7], which are Win32 applications operated through network sockets and a graphical interface, respectively. This leads to certification test sequences that require both executing scripted mouse clicks on the DMI (possibly also multiple DMI instances for some test sequences that consider interactions from multiple drivers) and injecting network messages toward the ETCS subsystem, while also evaluating both visual oracles on the DMI(s) output and behavioural oracles on the responses of the ETCS subsystem.

As the components do not expose any internal operation, the test sequences must be executed by using the actual interfaces. For example, the DMI inputs cannot be injected by calling graphical event handlers directly, but only by raising proper mouse click events via the Win32 graphical API (which is not available in Windows containers [12]).

The testing environment itself is highly interactive and relies on an external simulator with limited automation interfaces. Moreover, while the tests mandated by the certification authority comprise the execution of a large number of test sequences [6], the simulator supports only batch execution of test cases.

Overall, these constraints rule out conventional, container-based CI/CD solutions and require a fully interactive Windows environment. The existing industrial infrastructure executes several hundred certification test sequences sequentially, resulting in long turnaround times and generally preventing on-commit testing. Moreover, the system produces several heterogeneous logs from the DMI, the ETCS subsystem, the simulator, and further auxiliary agents, which must be correlated and presented in a unified way to support debugging and certification reporting.

This paper reports on our experience with designing and deploying a pipeline that enables continuous, on-commit testing of safety-critical railway software under the above certification constraints. Our contribution is not a new testing technique, but an industrial-grade infrastructure that integrates: (i) a build subsystem that invokes the SCADE toolchain, assembling and deploying mixed-origin binaries, (ii) a custom integration with the external simulator, (iii) dedicated agents for GUI automation and visual-oracle evaluation on the DMI, along with network communication with other components, (iv) a virtualized execution infrastructure supporting parallel test execution across isolated Windows environments, and (v) a centralized logging-management service that aggregates heterogeneous outputs into a unified stream that can be suitably visualized through dashboards.

In the paper, we describe the engineering decisions, trade-offs, and lessons learned while deploying this solution in a railway industrial context. We believe that this experience can be relevant to practitioners who face similar constraints in safety-critical domains, by showing how continuous testing can be achieved even when proprietary toolchains, interactive components, and certification requirements impose constraints that diverge substantially from conventional CI/CD practices. It may also benefit researchers by exposing practical challenges and research gaps grounded in safety-critical industrial reality.

2 Challenges

Developing a continuous, on-commit testing pipeline for ETCS onboard software requires addressing constraints that diverge from the assumptions underlying mainstream CI/CD automation tooling. In this section, we elaborate on the main challenges that drove our design decisions.

Challenge: On-Commit Testing with Certification Tests. The certification authority mandates the execution of several hundred test sequences, each comprising long, state-rich, and interactive scenarios [4, 6]. The existing industrial infrastructure relies on a custom simulator that orchestrates the execution of these test sequences by dispatching messages both to the components under test and the human testers responsible for operating the DMI.

Originally designed for manual testing workflows, the simulator supports only sequential, batch-style execution, with no facilities for dynamic configuration or integration into automated pipelines. Thus, executing the test campaign requires prohibitively long turnaround times, preventing on-commit testing.

Enabling continuous testing, therefore, calls for a distributed execution model capable of running large subsets of the test sequences in parallel, while preserving reproducibility and isolation.

Challenge: Interactive DMI Operation. Most test sequences include actions that, in operational settings, are performed by train drivers on the DMI graphical interface. In typical deployments, a train may provide up to two driver cabs, each equipped with its own DMI device. To enable automated testing, the driver actions scripted in the test sequences must be executed programmatically. However, because the DMI implementation is part of the certified system, code-level instrumentation is not viable, and interactions must be injected through the actual Win32 input APIs to preserve behavioural fidelity.

Similarly, the test sequences include visual oracles that must be evaluated against the DMI's graphical output, which requires capturing screenshots at precise execution points and interpreting them automatically.

Overall, the interactive nature of these sequences, combined with the need to support multiple concurrent DMI instances in multi-cab scenarios, introduces substantial complexity that must be properly addressed to achieve on-commit automated testing.

Challenge: Result Analysis with Heterogeneous Logging. The testing environment comprises many distinct log sources, including those generated by the components under test, such as the DMI and ETCS executables, as well as the logs from the custom simulator. As part of enabling continuous testing, additional logs are introduced by auxiliary agents (e.g., the agent responsible for programmatically operating the DMI) and by the runners that orchestrate the parallel execution of the test campaign. These logs differ widely in format, semantics, and granularity.

To support regression analysis, debugging, and certification reporting, developers must be able to inspect these results in a unified and coherent way. This requires consistently correlating, aggregating, and structuring the log data across components, runners, and test sequences, with accurate attribution of each event to the originating component and corresponding test sequence. Such correlation is essential for reconstructing the execution lifecycle, enabling event tracing, monitoring execution progress, and supporting failure analysis and debugging.

Challenge: Infeasibility of Container-Based Approaches. Modern CI/CD infrastructures rely heavily on containerized runners to ensure reproducibility, isolation, and scalability. In our context, however, this approach is infeasible for two independent reasons.

First, the build process depends on the a proprietary toolchain, ANSYS SCADE, which is not available in standard container images and requires licensed executables that cannot be freely replicated or executed in parallel within ephemeral environments such as containers or dynamically provisioned virtual machines.

Second, the test environment includes several Windows-based applications, most notably the DMI and the custom simulator, that

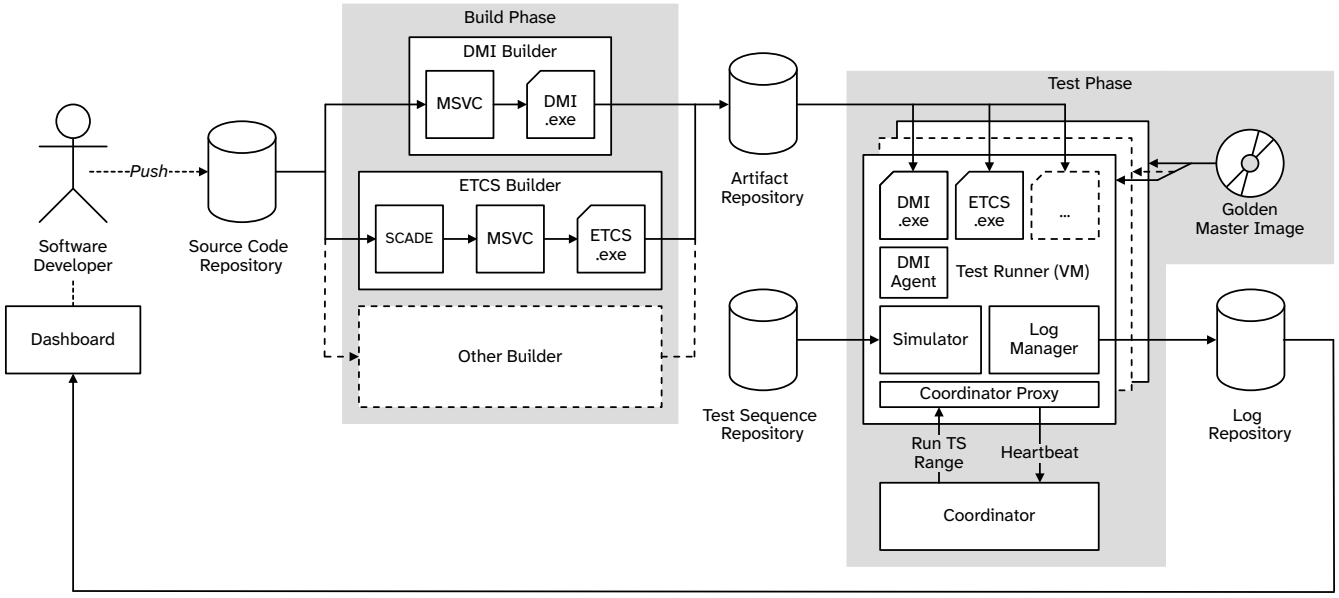


Figure 1: The continuous integration and continuous testing pipeline

require access to the Win32 graphical subsystem. These applications cannot run headless and cannot be executed inside Windows containers, which lack support for GUI-level APIs.

As a result, containerized runners are unsuitable for both building and testing, compelling us to adopt a virtual-machine-based execution model.

3 Continuous On-Commit Certification Testing

This section presents the solution that we designed to enable continuous, on-commit execution of certification test sequences for ETCS software. The design addresses the challenges discussed in Section 2 and is structured around three pillars: a virtual-machine-based execution model, a distributed, fully automated test execution phase, and a log management mechanism facilitating result analysis.

3.1 VM-Based Execution Model

Figure 1 provides an architectural overview of our solution. To address the constraints discussed in the previous section, on the infeasibility of relying on containers, the architecture exploits an execution model based on virtual machines (VMs).

As shown in the figure, the build phase executes across a set of dedicated VMs, one for each component that must be built. For instance, the DMI component is built in a VM equipped with the Microsoft Visual C++ toolchain along with the Qt graphical libraries, whereas the ETCS subsystem component is built in a VM hosting the ANSYS SCADE Suite and its licensed compiler. This architecture can be straightforwardly extended with further VMs for the build of any additional component as needed. This design isolates the specific build requirements per component, avoiding interference between toolchains. The resulting executables are published to the artifact repository.

The test phase (right of Figure 1) relies on multiple test-runner VMs instantiated from a single golden master image and managed by a coordinator VM. This enables the coordinator to launch several runners concurrently, distributing subsets of the certification suite among them. The golden master image satisfies all prerequisites for executing the certification test sequences, as we discuss in detail in the next section.

The overall architecture relies on a set of repositories to decouple and coordinate the build and test-runner VMs. The source code repository represents the git repository of the project. It is an instance of GitLab, which starts the pipeline upon commit events. The artifact repository stores the compiled executables produced during the build phase, allowing ephemeral test runners to download those executables in the testing environment. In the current implementation, it is realized with the Sonatype Nexus Repository Manager [15]. Each test runner can retrieve its own subset of certification test sequences from the test sequence repository. The log repository collects all logging data produced during test execution, enabling result analysis and long-term retention. We implemented it with Grafana Loki [9]. Together, these repositories support data transfer across VM boundaries while avoiding direct dependencies between execution environments, thus allowing the pipeline to scale easily.

3.2 Test Phase

The test phase, the core of the pipeline, enables parallel execution of certification test sequences. The coordinator orchestrates the test campaign by instantiating a set of ephemeral test-runner VMs from the golden master image, which provides a fully configured

Windows environment preinstalled with the simulator, the DMI agent, log manager, and coordinator proxy components.

Upon startup, each runner downloads the executables that comprise the software under test (i.e., the DMI, the ETCS subsystem, and so forth) from the artifact repository. The coordinator proxy then prepares the test environment. It receives the test-sequence range assigned by the coordinator, retrieves from the test sequence repository the corresponding subset of certification test sequences, generates the configuration files required by the simulator for executing those test sequences, and launches the simulator. During execution, it periodically reports the runner's status to the coordinator through heartbeat messages, enabling liveness monitoring and early detection of stalled or failed executions.

The simulator governs the lifecycle of each test sequence. It emulates onboard and trackside equipment and executes the steps scripted in the test sequences by dispatching messages to the ETCS subsysem and issuing driver-level actions to be operated on the DMI. The runners rely on the DMI agent to emulate the graphical interactions with the DMI. The agent instantiates one or two DMI processes, depending on the scenario, injects mouse-event sequences through the Win32 API, and captures screenshots of the DMI screens when requested from the simulator. The screenshots are fed both to the simulator for visual-oracle evaluation and recorded as logging data for facilitating result analysis. This mechanism addresses the challenge of providing fully automated execution of interactive test sequences that would otherwise require manual operation.

All logging data produced throughout the execution by all components (the simulator, the DMI agent, the executables under test, and the coordinator proxy) is collected locally, processed in the log manager to include metadata identifying the runner, the test sequence, and the component that issued that logging data, and then forwarded to the log repository. The metadata enables later analysis of the results produced in a distributed fashion across runners and components.

Overall, the test phase realizes dynamic workload distribution, isolated execution environments, and automated interaction with the DMI and the simulator, facilitating efficient on-commit execution of the certification suite while preserving reproducibility and analyzability of the test results.

3.3 Result Gathering and Analysis

The execution of certification test sequences produces a heterogeneous set of outputs, including textual logs from the DMI and ETCS executables, test execution data from the simulator, screenshots captured by the DMI agent, and system-level metrics from the coordinator proxy of each test-runner VM. To support debugging, regression analysis, and certification reporting, the pipeline gathers and presents these results in a unified and coherent way.

First, each test-runner VM includes a log manager (based on Grafana Alloy [8]) that aggregates the logs from all local sources and streams them to the centralized log repository. It parses the log entries into structured fields and enriches them with metadata identifying the runner, the test sequence, the component that produced the data, and the commit that triggered the pipeline. This enables proper interpretation and analysis of the events produced across distributed executions of multiple sequences.

Second, the log repository integrates with an interactive dashboard that makes the collected logging data accessible to engineers. The dashboard offers views for effectively exploring textual logs, DMI screenshots, simulator outputs, and test execution data. It can be deployed independently of the test infrastructure, supporting customized analysis workflows. Engineers can inspect the progress of a test campaign in near real time and examine the combined evidence produced by all components.

Overall, the result-management subsystem provides unified observability across distributed test executions and long-term retention for all collected data. By externalizing log storage, the architecture avoids coupling the runners to specific analysis tools and allows logs to be queried independently of the execution environment for different purposes, including diagnosis, regression analysis, and certification reporting.

4 Working Example

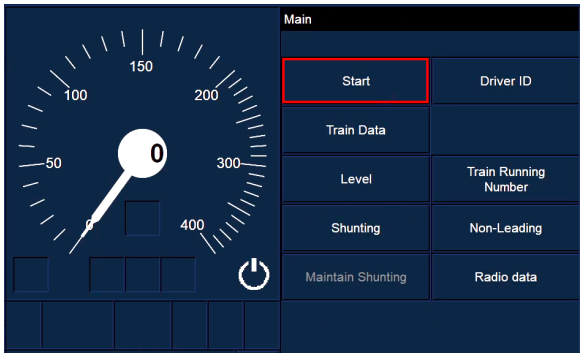
We discuss an example of how our pipeline handles the execution of the test cases, by referring to test sequence `3040400_03` from the certification suite «ETCS Subset 76» [6] which includes the following test case:

Test Case 16	
STEP	The driver selects 'Start'
STEP	Driver's selection of 'Start' is recorded on the Juridical Recording Unit
STEP	An acknowledgment request for running in Staff Responsible mode is displayed on the DMI

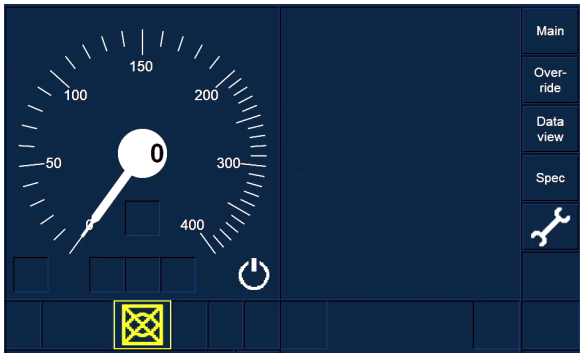
This test case requires the driver to press the *Start* button on the DMI (first step). Accordingly, it should be verified that a proper acknowledgment gets visualized on the DMI as a result (third step). In total, the entire test sequence `3040400_04` requires 37 mouse clicks to be executed by the driver.

Figure 2 illustrates how the DMI agent handles the interactions of test case 16, in order to achieve fully automated execution. The figure shows: (subfigure a) the status of the DMI screen at the beginning of the test case, (subfigure c) the data that the simulator transmits to the DMI agent, to indicate the DMI cab to be operated, the coordinates where to click (highlighted with a red box in the first subfigure) and the duration of the click event, (subfigure d) the pseudocode of the operations executed by the DMI agent to issue a mouse-click event according to the data transmitted by the simulator, and (subfigure b) the screenshot returned to the simulator. In this example, as the simulator successfully detects the expected acknowledgment within the screenshot image (that is, the yellow, crossed box at the bottom of the screenshot), the test case passes.

During execution, the log manager streams all logging data to the log repository, allowing for engineers to effectively monitor and analyze the results via the dashboard. For instance, Figure 3 exemplifies the use of the dashboard during the execution of the test sequence `3040400_04`. Initially, the engineers may be filtering the logging data to monitor the test results produced by the simulator (corresponding to source RTF in the figure), resulting in the visualization of only the rows with source RTF and RTF_Eval. Assuming a test run in which the oracle of test 16 reveals a failure



(a) Driver shall click inside the red rectangle



(b) Screenshot taken by the DMI Agent and sent to the simulator

Packet: Exec DMI Actions			
ID_DMI	0	uint8	0 for Cab A, 1 for Cab B
X_TARGET	660	uint32	X coord. of click action
Y_TARGET	140	uint32	Y coord. of click action
TIME_PRESS	5	int32	Press. time (deciseconds)

(c) Action Packet sent by the simulator to the DMI agent

```
FUNCTION Click(target, time_press):
    SetCursorPos(target.x, target.y)
    SendEvent(MOUSE_LEFT_DOWN):
    SleepDs(time_press)
    SendEvent(MOUSE_LEFT_UP)
```

(d) Pseudocode that generates the simulated event

Figure 2: Handling of the DMI by the DMI agent in test case 16

Source	Timestamp	Level	Message
RTF_Eval	2026-04-28 14:20:09	PASS	[28] State "3040400_04_28.png" passed.
⊞ RTF	2026-04-28 14:20:10		Test Case 16 – [29] The driver selects 'Start'.
DMI_Agent	2026-04-28 14:20:11	debug	ActionMsg - DMI: DMI_A, x: 660, y: 147, t_p: 10 ds
DMI	2026-04-28 14:20:11	Debug	Exiting Main Window Procedure ...
DMI	2026-04-28 14:20:11	Debug	Main Window Stopped
DMI	2026-04-28 14:20:11	Debug	Sending Btn Pressed : 5
DMI	2026-04-28 14:20:11	Debug	Btn Pressed sent
DMI	2026-04-28 14:20:12	Info	RX MSG Dim : 8
DMI	2026-04-28 14:20:12	Info	RX MSG Bytes : "12178df3691a0010"
RTF	2026-04-28 14:20:12		[30] Driver's selection of 'Start' recorded on JRU
DMI	2026-04-28 14:20:12	Info	DMI_SendRequest
DMI	2026-04-28 14:20:12	Info	TX MSG Dim : 12
DMI	2026-04-28 14:20:12	Info	TX MSG Bytes : "6b1b0000000e000c00000005"
RTF	2026-04-28 14:20:13		[31] An acknowledgement request for running in Staff Responsible mode is displayed on the DMI
DMI_Agent	2026-04-28 14:20:13	debug	CheckMsg - DMI: DMI_A, Task: TAKE_SHOT
DMI_Agent	2026-04-28 14:20:13	debug	Saved screenshot "3040400_04_31.png"
RTF_Eval	2026-04-28 14:20:14	FAIL	[31] State "3040400_04_31.png" not valid: E03

Figure 3: Dashboard visualization of logging data

(as indicated by the red log at the bottom of the figure), then the engineers may further exploit the dashboard, expanding also the logging data from other sources between the timestamps of the first and last log message in the figure. This also reveals the logs from the DMI and DMI agent (shown with gray fonts in the figure) assisting them in investigating possible reasons for the failure.

5 Lessons Learned

Our experience with deploying the pipeline revealed several practical insights, capturing both technical and operational factors that complement the architectural design discussed above.

Operational Management of the VM-Based Pipeline. Adopting a VM-based execution model introduces recurring operational tasks that must be managed throughout the lifecycle of the pipeline. In our setting, two aspects proved particularly significant. First, both the build-management VMs and the test-runner VMs require periodic updates of tools and external dependencies, making the maintenance of these environments an ongoing responsibility rather than a one-time setup activity. To support this process, we found it essential to provide a suite of provisioning scripts that allow the maintenance team to rebuild the VMs and regenerate the golden master image automatically when needed.

Second, the provisioning scripts play a central role in configuring the degree of parallelism for test campaigns. They allow tuning both the number of test-runner VMs to deploy and the level of concurrency used during their instantiation. In our current configuration, the pipeline uses ten test-runner VMs and handle the instantiation of these VMs with five parallel deployment jobs. Under these settings, the full deployment completes in approximately 3.5 minutes.[1] After deployment, the VMs of the ten test runners collectively consume about 20 GB of RAM.

Once the environment is properly configured and dimensioned, executing test campaigns requires the coordinator to simply distribute the selected test sequences across the available runners, and the pipeline proceeds as we described above.

Reliability of GUI Automation and Visual Oracles. Automating DMI interactions raises challenges for coordinate-based input injection and screenshot-based visual oracles. In our setting, these techniques proved robust mainly because the DMI implements standardized, fixed-size layouts with large touch targets and fully deterministic behaviour. However, such interactions can become fragile in the presence of dynamic or adaptive interfaces. GUIs with variable layouts or richer visual dynamics may require more resilient automation strategies to achieve comparable reliability.

Execution Traceability. To support effective post-execution analysis of test results, it is essential that developers can reliably attribute the collected evidence to the specific test sequences executed on each runner and to the executables that produced each piece of evidence. To this end, we purposely designed our pipeline

[1] On a host machine equipped with an Intel® Core™ i9-12900HK CPU, 32 GB of DDR5 RAM, and a 1 TB NVMe SSD, running Windows 11 25H2 and Hyper-V.

to enrich log streams with execution-specific metadata, such as unique run identifiers, timestamps, and test-sequence labels. Without such contextual information, correlating events across parallel executions would become impractical, severely limiting failure analysis and traceability.

In our setting, this process is also facilitated by the fact that the test sequences are completely independent from each other, preventing cross-sequence interference during parallel test executions. In more general environments, where this assumption might not hold (for example, due to shared resources or ordering constraints), ensuring correct attribution of test results may require logging metadata more sophisticated than ours, capable of capturing runtime context propagation and dependency tracking.

Decoupling Test Sequences from the VM Environment. In our experience, managing the dependencies between tested artifacts and the runtime environment benefits from keeping the test-sequence repository external to the VM instances. This design simplifies maintenance, as updates to the test sequences do not require modifying the VMs or the golden master image. It also allows us to straightforwardly reconfigure test campaigns to focus on specific subsets of the certification suite without altering or redeploying the underlying VM infrastructure.

6 Related Work

Other authors have explored the adaptation of DevOps and continuous engineering to regulated domains and safety-critical systems. Zampetti et al. report on CI/CD adoption in cyber-physical systems, identifying challenges related to simulation, heterogeneity, and hardware-in-the-loop integration [16, 17]. Myklebust et al. explored the use of agile development in railway signaling systems [13]. Zeller proposes a DevOps pipeline that grounds model-based approaches for continuous assessment of safety-critical software [18]. The DevOps workflow of Pfaff et al. comprises a container-based CI/CD infrastructure that integrates simulation and on-vehicle Rtesting [14]. Zeller et al. introduce a Git-centric pipeline for railway components based on machine learning [19].

Our work properly complements the above research efforts exploring parallel execution of certification test suites for ETCS software, ultimately reinforcing the evidence that continuous testing can be adapted to safety-critical contexts.

Other contributions addressed some of the challenges that also belong to our context. Kajiser et al. integrated simulation-based verification into continuous integration pipelines [11]. Bautista et al. exploited Grafana Loki to support scalable log aggregation and monitoring in high-throughput distributed systems [2]. Guida et al. proposed a robotic-arm setup to automate realistic interactions for physical DMI testing [10]. Our approach tackles the combination of such challenges through a structured, continuous testing solution that facilitates automated execution of certification test sequences for ETCS software.

7 Conclusions

In this experience report, we discussed the design of a pipeline for continuous integration and continuous certification testing of ETCS software, highlighting the practical challenges we faced in our industrial setting and the solutions we deployed correspondingly. Our experience provides empirical evidence that on-commit certification testing can be effectively supported, and highlights key architectural choices to automate and parallelize execution of certification test suites. Furthermore, our solution conveys useful insights on how to handle test suites that require interactions with drivers, depend on proprietary simulators, and produce heterogeneous logging data that must be effectively visualized in various analysis scenarios.

We are now working to extend the pipeline with test-selection and test-prioritization capabilities, based on tracing the relations between the source artifacts and the impacted requirements, in order to further improve the efficiency of on-commit testing.

Data Availability Statement

Our executable pipeline cannot be made publicly available due to confidentiality constraints of the industrial project in the work is being developed.

References

[1] Ansys Inc. 2024. *Ansys SCADE Suite.* https://www.ansys.com/products/embedded-software/ansys-scade-suite

[2] Elizabeth Bautista, Nitin Sukhija, and Siqi Deng. 2022. Shasta Log Aggregation, Monitoring and Alerting in HPC Environments with Grafana Loki and ServiceNow. In *2022 IEEE International Conference on Cluster Computing (CLUSTER).* 602–610. doi:10.1109/CLUSTER51413.2022.00079

[3] Docker Inc. 2026. Docker Overview. https://docs.docker.com/get-started/docker-overview/ Docker Documentation.

[4] European Committee for Electrotechnical Standardization. 2023. *Railway applications – Requirements for software development.* Technical Report EN 50716:2023. CENELEC, Brussels, Belgium. https://standards.cencenelec.eu/ords/f?p=305:110:::::FSP_ORG_ID,FSP_PROJECT,FSP_LANG_ID:1257173,72410,25&cs=10DBEB0FC42801BB40F44D744D792A6E2

[5] European Union Agency for Railways. 2026. European Rail Traffic Management System (ERTMS). https://www.era.europa.eu/domains/infrastructure/european-rail-traffic-management-system-ertms_en

[6] European Union Agency for Railways (ERA). 2017. ERTMS Scope of the Test Specifications. https://www.era.europa.eu/system/files/2023-01/sos3_index037d_-_subset-076-7_v330_era-opi-2017-5.zip Version 3.3.0.

[7] European Union Agency for Railways (ERA). 2023. ETCS Driver Machine Interface. https://www.era.europa.eu/system/files/2023-12/index006_-_ERA_ERTMS_015560_v400.zip Version 4.0.0.

[8] Grafana Labs. 2024. *Grafana Alloy.* https://grafana.com/docs/alloy/

[9] Grafana Labs. 2024. *Grafana Loki.* https://grafana.com/oss/loki/

[10] Giuseppe Guida, Mario D'Avino, Massimo Nisci, Roberto Villano, Barbara Di Giampaolo, Michele Schettino, and Erasmo Alessio Bortone. 2025. Automated Testing in Railways: Leveraging Robotic Arm Precision and Efficiency for Superior Customizability and Usability Test Case Execution. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 139–148. doi:10.1109/ICSTW64639.2025.10962477

[11] Henrik Kaijser, Henrik Lönn, Peter Thorngren, Johan Ekberg, Maria Henningsson, and Mats Larsson. 2018. Towards Simulation-Based Verification for Continuous Integration and Delivery. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018).* Toulouse, France. https://hal.science/hal-02156371

[12] Microsoft. 2026. Lift and Shift to Windows Containers - What can't be moved to Windows containers. https://learn.microsoft.com/en-us/virtualization/windowscontainers/quick-start/lift-shift-to-containers#what-cant-be-moved-to-windows-containers Microsoft Learn Documentation.

[13] Thor Myklebust, Tor Stålhane, and Narve Lyngby. 2015. Application of an Agile Development Process for EN50128/Railway Conformant Software. doi:10.1201/b19094-529

[14] Raphael Pfaff, Maria-Lenka Wolter, and Luigi Ferraioli. 2025. DevOps for Automatic Train Operation with the DEFACTO Pipeline. In *2025 IEEE International Automated Vehicle Validation Conference (IAVVC).* 1–6. doi:10.1109/IAVVC61942.2025.11219450

[15] Sonatype, Inc. 2024. *Nexus Repository.* https://help.sonatype.com/repomanager3

[16] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. 2022. Problems and Solutions in Applying Continuous Integration and Delivery to 20 Open-Source Cyber-Physical Systems. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. Association for Computing Machinery, New York, NY, USA, 646–657. doi:10.1145/3524842.3527948

[17] Fiorella Zampetti, Damian Tamburri, Sebastiano Panichella, Annibale Panichella, Gerardo Canfora, and Massimiliano Di Penta. 2023. Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study. *ACM Transactions on Software Engineering and Methodology* 32, 3, Article 73 (April 2023). doi:10.1145/3571854

[18] Marc Zeller. 2021. Towards Continuous Safety Assessment in Context of DevOps. (2021), 145–157. doi:10.1007/978-3-030-83906-2_11

[19] Marc Zeller, Thomas Waschulzik, Carmen Carlan, Marat Serahlazau, Claus Bahlmann, Zhiliang Wu, Sigurd Spieckermann, Denis Krompass, Simon Geerkens, Christian Sieberichs, Konstantin Kirchheim, Batu Kaan Özen, and Lucia Diez Robles. 2024. Continuous Development and Safety Assurance Pipeline for ML-Based Systems in the Railway Domain. In *Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops*. Springer Nature Switzerland, Cham, 446–459. doi:10.1007/978-3-031-68738-9_36

Received 2026-05-01; accepted 2026-07-18