2026 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
DIGITAL ENGINEERING/SYSTEMS ENGINEERING (DE/SE) TECHNICAL SESSION
AUG. 11-13, 2026 - NOVI, MICHIGAN

CI/CD FOR MISSION-CRITICAL SYSTEMS: A FRAMEWORK FOR ANY LEVEL OF RIGOR

Michael Lingg, PhD[1], Howard Paul[1], Brian Schulte[1], and Robert Wilkinson[2]

[1]Array of Engineers, Grand Rapids, MI
[2]DEVCOM GVSC, Warren, MI

ABSTRACT

Modern mission-critical ground vehicle systems must adapt to rapidly evolving threats, deploying changes in months or days while maintaining reliable and safe operation. Historic manual development and testing methods cannot keep pace without compromising safety assurances. Continuous Integration and Continuous Deployment (CI/CD) pipelines offer proven approaches to accelerating development, but implementing them for mission-critical systems requires careful attention to verification rigor.

This paper presents a practical framework for implementing CI/CD pipelines across any level of rigor, from rapid prototyping to DO-178C and ISO 26262 certified systems. Drawing on experience from aviation, medical device, and ground vehicle development, the framework provides guidance for each pipeline stage based on the system's desired level of rigor. This framework includes an examination of the value of Software-in-the-Loop vs Hardware-in-the-Loop testing to optimize development timelines while maintaining software quality.

Citation: M. Lingg, H. Paul, B. Schulte, R. Wilkinson, "CI/CD for Mission-Critical Systems: A Framework for Any Level of Rigor," In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 11-13, 2026.

1. INTRODUCTION

Development of mission-critical systems, systems designed to ensure the highest likelihood of successfully completing a mission, continues to change at an increasing pace. In the past new cutting edge systems took years to decades to develop. The M1 Abrams tank development stretched to nearly a decade, while the Patriot Missile System required more than a decade of development. System development took a significant shift during the War on Terror where operations shifted to ground operations in an urban environment requiring unexpected changes to mission-critical systems such as adapting and deploying Mine-Resistant Ambush Protected (MRAP) vehicles to counter the threat of Improvised Explosive Devices (IEDs). More recently in the Ukraine war the use of commercial and improvised drones has created a new and difficult to counter threat.

Modern mission-critical systems still require years to develop from the ground up, but also must quickly adapt to new threats and mission parameters, often in months or even days.

This need to accelerate system development comes into direct conflict with the requirement that a mission-critical system must work reliably, repeatably and safely to both protect the soldiers using the system and allow them to successfully complete their mission. Safety-critical systems are designed to ensure the safety of people interacting with them. These systems are governed by safety process standards such as DO-178C [1] and ISO 26262 [2], which in military projects can be used to satisfy the framework defined in MIL-STD-882E [3] and elaborated in the Joint Software Systems Safety Engineering Handbook (JSSSEH) [4]. These processes are historically predicated on stable and well-defined systems where safety-critical concerns can be identified and mitigated. Having shorter time frames to integrate new components into an existing system compresses otherwise careful and methodical processes used to ensure new components will integrate safely into a new system without unintended consequences. Many historic safety-critical development methods such as the waterfall approach and manual system testing cannot be accelerated without compromising the safety assurances they are designed to provide.

Automated approaches to testing and deployment, particularly Continuous Integration and Continuous Deployment (CI/CD), have proven effective at accelerating development timelines in commercial software. Many methods exist for automated test generation and detection of common programming errors. These methods are widely adopted because they are relatively straightforward to implement and can run with minimal human intervention. However, verifying that software performs its intended functionality correctly requires formal specifications of expected behavior. Defining and testing these formal specifications is a time-consuming and costly process that is often deprioritized in favor of more easily automated crash detection. For mission-critical systems governed by standards such as MIL-STD-882E where safe and reliable operation is a priority, ensuring the system performs its intended functionality is paramount. A key challenge in implementing CI/CD for mission-critical systems is to ensure the automated testing pipelines verify the system not only operates without crashing, but also functions as intended.

This paper provides a practical framework for using CI/CD pipelines to accelerate the development and integration life cycle. This framework draws on decades of experience working on DO-178C certified safety-critical systems, medical device development and testing as well as practical experience in developing CI/CD pipelines for ground vehicle systems. This broad experience is used not to push the most strictly regulated but slow and costly civilian aviation development methods, but to provide a range of approaches that can be adapted to the need of the current project. The approaches represented can range from the integration of rapidly developed prototype components that may enter the field quickly at the risk of less rigorous testing, to systems that must perform reliably in the field to avoid casualties. The goal of this paper is to help identify why different methods within CI/CD are used, so practitioners can make informed decisions while developing their own processes.

2. BACKGROUND

Historically the options for automated testing and deployment were limited. Software updates would need to be loaded into a test system and executed by trained operators to determine if the change is accepted. Once the software update was considered good, the fielded systems would need to be returned to have their software updated, or possibly have software changes physically brought to the fielded system. This manual test execution approach is too slow and unreliable to keep up with rapidly evolving environments. Full system testing can take hours to months of time to be executed by trained operators, depending on the complexity of the system. Manual test execution is also error prone and difficult to repeat reliably depending on how well the operators are trained, and even the most trained operators can suffer from different interpretations of how a test should be performed. Designing a system to be easy to update the software can be a security issue as the update interfaces can be hacked, while systems where the software is well secured can be difficult if not impossible to update in the field.

Automated testing was one of the first methods to address the shortfalls of the manual testing and deployment approach. These automated tests can range from testing a single module or function in isolation, to testing a fully integrated software package, on up to using a full Hardware-in-the-Loop (HiTL) test system to test an entire hardware/software integrated system. Full system testing allows the system to be tested with the hardware and environmental restrictions the system would operate within the field. In an ideal world all testing would use the full system as this most closely matches how a system will operate in the field. To accelerate system testing and make it more accessible, HiTL test systems can be used to automate these tests, and enable the testing of conditions that are very difficult for human testers to replicate. These HiTL systems have been proven [5] to reduce manual system tests taking weeks down to a few dozen hours and improve repeatability of testing. Unfortunately the speed of testing is still limited by running on real time hardware as well as the hardware required to create such a test lab continues to be expensive and difficult to acquire in sufficient quantities for many projects.

Software-in-the-Loop (SiTL) tests can further accelerate the testing timeline and mitigate expensive system hardware costs by running tests in a virtual environment. These virtual tests can range from simulating the full system and its hardware environment, to executing tests on a single software module. While SiTL tests do not perfectly replicate the actual system, these simulated environments continue to become a more accurate representation of actual systems. Virtual environment for SiTL tests can be quickly spun up on one or more server computers, providing the ability to inexpensively scale up the number of virtual test benches unlike HiTL labs. The different test levels available in SiTL can be particularly useful for the Modular Open Systems Approach (MOSA) [6]. Module tests can verify software updates for individual modules provide the expected behavior, then integration tests can ensure that the modules interact correctly, and finally system level testing verifies the entire system operates as expected in this simulated environment before testing on the limited real hardware environments.

One additional benefit which automated testing can provide is abstracting the test procedure functions so that they mimic system interfaces. The test developer can simply call a test method to press a virtual button, and at runtime the appropriate

libraries handle the actual hardware interactions. The same test functions can execute module-level test harnesses, integration test environments, or full SiTL or HiTL systems simply by changing an execution context parameter. This approach allows test authors to focus on what should be tested rather than how to execute tests at each level, while maintaining a single source of truth for expected system behavior.

During the same time automated testing was becoming a reality, software deployment methods continued to evolve to support faster update cycles. Early systems required physical access to load software via dedicated programming interfaces or removable media such as floppy disks or USB drives. Modern systems increasingly support over-the-air (OTA) updates that can deploy software patches wirelessly to fielded systems without manual intervention, dramatically reducing the logistics burden of software updates. However, each deployment method presents security challenges be it physical update methods requiring physical ports to load software or wireless deployment introducing network security concerns.

CI/CD pipelines integrate these automated testing and deployment capabilities into a unified workflow, running on a scalable cloud system. Software changes progress through a series of automated stages. Storing the change in a source code repository, static analysis of the code, building the software, build artifact storage, automated testing of the build artifacts and finally deployment. The results of each stage can trigger automated checks that stop progress or roll back failed changes, as well as provide quality gates for manual inspection before proceeding. Each software update can result in the execution of simple delta tests or full regression analysis. A particular benefit of the automation is comprehensive regression testing can be run overnight without interfering with active development work.

In the next section we will discuss each stage of a CI/CD pipeline in detail. Covering how it can be implemented and some tools that might be used, as well as levels of processes that might be desired depending on the level of rigor identified for the testing and deployment of the software.

3. FRAMEWORK AND IMPLEMENTATION GUIDANCE

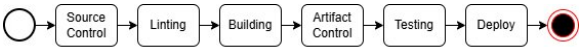

Figure 1: Example CI/CD pipeline workflow.

A key feature of CI/CD pipelines is they are made of multiple discrete steps that provide the output of one step as the input to the next, as visualized in Figure 1, once the previous step passes certain gateway conditions. These steps require the storage and labeling of data or produce results from executing some process. Each step can run or store data using any appropriate computing hardware, so long as it is capable of communicating the results to the hardware running the next step. Any step can even be a manual process, such as field testing, where the tester updates a flag in the pipeline or uploads a result document to satisfy the gateway allowing the pipeline to proceed.

This pipeline enables a digital engineering or cloud computing approach where a single machine can spin up multiple Virtual Machines (VMs) or container instances to handle the current step(s) being executed, execute each step on a separate machine, or even spread computationally or storage intensive steps across multiple machines as needed. A CI/CD pipeline may prefer to use one server for software and artifact repositories with high storage requirements, and one or more other servers for computationally intensive steps like SiTL testing. The actual hardware running steps in the CI/CD pipeline are less important

than their ability to pass results along the pipeline. This means a CI/CD pipeline can be shared between multiple programs, adapting as one program scales up and another scales down. Different projects can even use different tools, appropriate to the project's level of rigor, for their use of the shared cloud resources. This also means considerations need to be made for security of the CI/CD pipeline where hardware is shared across projects with different security requirements.

Additionally while configuration management and change tracking provides vital stages of the CI/CD pipeline, when the pipeline is configured through Infrastructure as Code, the pipeline configuration itself can be tracked through configuration management. Things like which applications are required to run a SiTL test, or which nodes have access to required hardware, can be submitted as configuration files so that configuration management can ensure reproducibility and track changes to the CI/CD pipeline.

3.1. Code Repository

Version control and configuration management is very important at multiple levels of CI/CD for tracking what artifacts are associated with what test results. Tracking code changes in particular can be helpful to determine when any stage of the CI/CD pipeline fails, and what changes are related to the failure. Source code repositories are typically designed to archive text documents as source code and often include functionality for difference reporting and change analysis of textual documents. If the source code is primarily textual source code, the repository can perform compression by only storing what has changed from one version to the next.

Low rigor projects may only require a basic Git repo with version history information. Higher levels of rigor may require problem tracking and code reviews. Problem tracking needs the ability to link an issue or bug to the version of the code that introduced the issue as to versions of the code that resolved the issue. Code reviews likewise need the ability to link to the version of code that triggered the code review, as well as versions of the code that were updated as a result of the code review. For the highest levels of rigor, code reviews might include Change Control Board (CCB) approval of the software and bidirectional traceability between the source versions and the requirements.

A number of high quality tools exist for software version control, PTC Integrity Lifecycle Manager, Subversion, Git. Most modern projects use some version of Git.

While Git by itself provides excellent version control capabilities, linking software versions to requirements, test or problem reports often requires additional tools or custom development work.

3.2. Static Analysis

Static analysis tools automatically examine source code without executing it, identifying potential bugs, security vulnerabilities, and standards violations. This is a critical early step in the CI/CD pipeline, identifying code patterns that often lead to bugs and security issues. Static analysis is typically the first step performed after new code is submitted to the code repository, and may be a blocking step before proceeding to building the code.

Basic linting tools check code style and simple syntax issues. Tools like pylint (Python), ESLint (JavaScript), or basic compiler warnings provide quick feedback with minimal false positives. These can be run on every commit with negligible performance impact.

A more rigorous analysis can be performed through coding standards compliance and deeper analysis. This can include checking adherence to coding standards

like MISRA-C (Motor Industry Software Reliability Association C standard in automotive/embedded), CERT-C (Computer Emergency Response Team C standard in security), or analyze code against project-specific guidelines. Example tools include PC-lint, Coverity, or SonarQube. These tools often generate hundreds of warnings that require review and potential suppression/justification but these warnings indicate issues with the code more often than not and are important to consider in a more rigorous environment.

Very rigorous development environments such as safety-critical systems with high levels of risk are supported by full standards compliance with zero-tolerance policies and formal deviation tracking. Tools like LDRA, CodeSonar, or Polyspace provide comprehensive analysis qualified to meet DO-178C or ISO 26262 requirements. Every identified violation must be either fixed or formally justified with documented rationale. These tools are often qualified with certain guarantees for detecting code issues, which results in a notable increase in cost but is required by some certification authorities for use to certify the software.

Static analysis tools are typically easier to integrate into a CI/CD pipeline as they simply need to be executed on the latest code submitted to the software repository and their results analyzed to determine if the software is ready to progress to the build stage. The main concern with linting tools is as they become more strict there can be a significant number of warnings to analyze and address. It can become easy to dismiss or suppress warnings without proper analysis. However, the more rigorous the environment, the more critical it becomes to properly evaluate each warning. While integration is typically straightforward, some tools may require custom development work to fit specific CI/CD pipeline architectures.

3.3. Code Build

The code build step compiles source code into executable binaries using only approved toolchains, ensuring compatibility with the target system. Automated builds in CI/CD pipelines provide immediate feedback when code changes break compilation and guarantee that all developers use consistent build configurations. This step typically occurs after the static analysis has been performed to ensure the code is written to the proper quality, and a successful build must be created before the software can be tested. Eliminating all code warnings may be a guard condition before moving onto testing depending on the level of rigor.

In basic research and development projects, standard open-source compilers (GNU Compiler Collection, Clang, Microsoft Visual C++) can be used with basic optimization settings. Build configuration may be documented but not strictly enforced. Any build warnings should be addressed, though this may not be strictly enforced at this level of rigor.

At higher levels of rigor processes should include controlled compiler versions and build flags documented in version control. Compiler warnings should be treated as errors to prevent any code generating warnings from progressing. Build processes must be documented so that builds are strictly reproducible. When using specific embedded systems, specific build processes and tools may be required to work with the specific systems. Build logs and compiler output should be captured for debugging failed builds or as certification artifacts.

In fully certified safety-critical environments, qualified or certified compiler toolchains are often required to meet certification requirements. For example, DO-178C qualified compilers can cost tens to hundreds of thousands of dollars in order to satisfy certification

requirements. Due to this cost, it is important to have appropriate budget expectations and understand which qualification level is truly required. Certification standards such as DO-178C or ISO 26262 often require entire toolchain qualification documentation to be presented to the certification authority. To support this the build process should be under configuration management with formal change control. Compiler settings should only be changed through a formal review process. Build artifacts must include all metadata needed for traceability (source version, toolchain version, build flags, timestamps).

Simple open-source compilers can be easily integrated into most CI/CD pipelines. Embedded systems or qualified environments can result in complex dependencies. The need for specific tool versions, correct environment variables, and build system quirks like cross-compilation can make build processes fragile. However, once a build process is working in a CI/CD pipeline, an automated process can ensure the build process continues to work as long as the target hardware remains the same. Having a specific automated build process used by all developers can avoid the issue of local configurations that successfully build the software but cannot be replicated by anyone else.

3.4. Build Artifact Repository

Build artifact repositories store versioned binaries, executables and configuration files produced by the build process, providing traceability from tested artifacts back to source code versions and forward to deployment. Unlike source code repositories that efficiently store text-based changes, artifact repositories must manage large binary files that change frequently. The artifact repository is often directly provided the files resulting from the build stage, and need the build to provide all of the necessary artifact information. Newly stored artifacts typically trigger automated tests to execute. Artifacts can be marked with a state progression such as in development, in test and released which can be updated manually as well as automated state transitions depending on the results of the automated tests.

Artifact repositories associate each successfully built artifact with version information, linking it to the specific source code version, build configuration, and toolchain used. This enables precise reproduction of any deployed binary and supports rollback to previous versions if issues are discovered. Common tools include Artifactory, Nexus, and cloud-based solutions like Amazon S3 or Azure Blob Storage. Commercial or privately managed cloud services are often used for artifact repositories due to the ease of scalability.

Basic artifact repositories may store artifacts on shared file systems with directory structures or text files providing version information. Retention may be informal, keeping recent builds without defined policies.

For higher levels of rigor more formal versioning information is stored tracking data such as build date, source version, and links to associated tests. Automated retention policies can delete old artifacts after defined periods and access permissions can be used to limit who can upload or download artifacts.

In fully certified safety-critical system, a complete traceability chain must often be maintained. To achieve certification, artifacts often include cryptographic checksums for integrity verification, links to qualification documentation, and full build metadata such as source version, toolchain version, compiler flags, build environment. Digital signatures of artifacts prevent tampering, ensuring the integrity of the files. Strict retention policies

for certified releases often last for the operational lifetime of the system or longer.

The size of files being stored is a significant consideration in artifact repositories. A single embedded system binary might be 50-500MB. Artifacts produced by daily builds across multiple branches storage needs can quickly reach into terabytes. Retention policies must balance traceability needs (keeping everything) against storage costs (deleting old artifacts). Safety-critical projects may be legally required to retain certified artifacts for the product's entire service life.

3.5. Requirement Management

Requirements management depends on the level of rigor the system is being developed to. In most CI/CD systems, requirements sit outside of the pipeline, but traceability might be used to identify if a requirement is fully implemented in code and if all of its tests are passing. For low levels of rigor, the tests themselves can be used to define what the system behavior should be. For a higher level of rigor, basic use cases and definitions of the key behaviors the system should implement can be described. For the highest level of rigor, requirements or models may describe the entire behavior of the system. At this highest level, bidirectional traceability is used to identify what code is implementing which requirements and which tests are verifying which requirements. Software coverage analysis of the tests would be used alongside of the traceability to ensure all requirements are implemented and no functionality beyond the requirements exist.

Defining behavior of the entire system through requirements can be expensive and time consuming. However once the definition of the system is in place, traceability can quickly identify what code and tests need to be updated when requirements change, or where requirements and code mismatch if tests fail. At this level of software development, tests are used less to find bugs in the code, and more to identify where requirements and code do not match, allowing the system and software developers to determine if requirements, or code has the correct behavior, or if both are incorrect. This can avoid confusion between what the system should do and how the system was implemented.

Requirements management defines what the system should do. This is not simply a description of how the software will be implemented but is also the foundation for verifying the implementation matches the requirements. The level of rigor in requirements management directly impacts development costs, traceability overhead, and certification viability.

For rapid prototyping or low-risk systems, tests themselves can define system behavior. Test-Driven Development (TDD) writes tests before implementation, and can even skip the creation of requirements to define software behavior with the tests themselves serving as the specification. This approach minimizes documentation overhead and keeps requirements implicit in executable tests. However, the lack of a formal requirement definition can make it difficult to verify complete system behavior or identify gaps in functionality.

In projects with higher levels of rigor the requirements document key system behaviors through use cases, user stories, or functional specifications. These specifications define critical workflows and expected system responses without requiring exhaustive detail. Requirements may be managed in tools like Jira, Confluence, or lightweight requirements management systems. Some traceability may be maintained manually through spreadsheets or simple linking mechanisms to connect requirements to code and tests, helping to reduce the time diagnosing a test failure.

This level provides an improved overview of system intent without the high costs of formal requirements engineering.

Safety-critical and mission-critical systems require complete, traceable requirements. Every system behavior must be specified in descriptive textual requirements or models (such as Universal Modeling Language (UML), Systems Modeling Language (SysML), or Simulink models). Bidirectional traceability links each requirement to implementing code and verifying tests, reducing the time and cost to diagnose test failures. Requirements management tools like Dynamic Object-Oriented Requirements System (DOORS), Jama Connect, or Polarion maintain these trace relationships and support impact analysis when requirements change.

At this level, software coverage analysis works alongside traceability to ensure all requirements have implementing code, all code traces to requirements, all requirements have verifying tests, all tests trace to requirements.

Modern requirements management tools can integrate with CI/CD pipelines to automatically flag suspect traces. This process helps to identify when changes have been made in requirements, code or test and have not been propagated to the rest of the process. When tests fail, automated tools can identify which requirements are affected through trace links. This automation helps ensure consistency in identifying connections between requirements, code and test, though initial trace creation and ongoing verification remains labor-intensive.

3.6. Issue Tracking

Another process that exists at higher levels of rigor is problem reporting and change tracking. This is another stage in the CI/CD pipeline that often guards against proceeding with software changes until they are confirmed to have successfully addressed a previously identified issue. Much of the process involves manual input: identifying the software issue and associated versions, marking updates as proposed resolutions, and obtaining final approval of the solution.

Problem reporting and change tracking systems document software defects, enhancement requests, and change history throughout the development lifecycle. While these systems exist outside the core CI/CD pipeline, they integrate with it to ensure software changes address identified issues and maintain traceability from problem identification through resolution verification.

Simple research projects may not use any issue tracking, or may use basic tools like GitHub Issues, GitLab Issues, or spreadsheets. Developers can informally document bugs and link commits to issue numbers through commit messages. This provides basic history but limited formal tracking or approval workflows.

Larger projects or projects with increased rigor will often use structured issue tracking systems like Jira, Azure DevOps, or Bugzilla to formally document each problem with priority, assigned owner, and status tracking. Issues are linked to specific code versions that introduced the problem and versions that resolved it. Code changes reference issue IDs, creating bidirectional traceability between problems and fixes. Automated workflows can require code reviews before issues are marked resolved.

Projects with the highest rigor or safety-critical systems will typically include formal problem reporting with CCB approval. Issue reports commonly include information like:

- Affected versions and configurations
- Root cause analysis
- Proposed resolution approach

- Impact assessment on requirements and tests
- CCB approval before implementation
- Verification evidence that the fix resolved the issue

In higher rigor systems, issue tracking tools often integrate into the CI/CD pipeline with traces between issue reports, requirements, code versions and tests to make tracking issues and identifying resolutions less complex. The issue tracking tools can also act as guards to the CI/CD pipeline, requiring not just passing tests, but independent approval of issue resolutions before software is released.

3.7. Software Tests

Automated test execution operates at multiple levels, balancing test fidelity, execution speed, and resource availability. Both Software-in-the-Loop (SiTL) and Hardware-in-the-Loop (HiTL) testing play important roles, with the choice depending on what is being validated and resource constraints.

Module Test Module or unit level tests typically isolate individual functions, set the input parameters to the function and observe the function's output value. The test is executed by a test harness that is able to execute the function with the desired inputs and compare the outputs against expected results and can be run on a normal Personal Computer (PC), not requiring system hardware. These tests are very low level and it is often difficult to tell how a combination of inputs to a function relate to the overall system behavior. The benefit of these low level tests is it is much easier to test combinations of inputs that are difficult to test in the full system. Typically these tests are simple PC executables that run the software under test per a test script defining the inputs and expected outputs.

Integration Test Integration or high level tests typically set the signal inputs to a software component comprised of many functions and observe the signal outputs from the component. These tests exercise the interface between multiple functions, helping to prove that the data being passed between functions is consistently handled by every function, for example no mistaken metric to english conversions. Integration tests can range from stubbing out individual libraries to exercising a full application. One additional benefit with integration tests is typical data is limited to the practical range of values that are allowed by the system, while module tests may test values that are explicitly prevented by other code used to filter and limit data to valid ranges. Integration tests typically execute in SiTL environments on standard compute hardware, enabling parallel execution across multiple test instances. For timing-critical integration scenarios, representative hardware may be used to execute tests in real time.

System Test System level tests typically execute tests with a fully integrated software and hardware environment using either simulated hardware SiTL tests or real hardware HiTL tests. These tests verify software behavior as it interacts with real hardware interfaces and is executed in simulated or real time hardware execution cycles. While system tests were historically run through manual execution with many system inputs connected to real world sensors, newer HiTL systems interface with the system to simulate the environment the system interacts with. This simulation, known as a digital twin, simulates conditions that might otherwise be impossible or dangerous if the simulation conditions were performed in the real world.

SiTL simulated environments can be deployed elastically in cloud infrastructure,

allowing extensive parallel testing without the need for expensive system hardware. While SiTL environments require significant computational resources to fully simulate hardware behavior, high-performance servers are typically more readily available and less expensive to procure than actual system hardware. Where hardware costs might limit a test facility to a few HiTL systems, the same budget can often support more than double the number of SiTL test instances. While costs are a significant issue, hardware availability can be more significant, with Commercial off the Shelf (COTS) servers being far easier to procure at scale. Additionally, the same computational resources used for system-level SiTL testing can be dynamically reconfigured to run module or integration tests as needed, providing flexibility that real system hardware cannot provide. A full system test suite requiring dozens of individual test scenarios can complete in hours when run in parallel across the larger number of SiTL instances, compared to potentially days of real time execution on more limited HiTL hardware.

HiTL system testing finally executes the software on the same hardware that will be used in the field, achieving the highest fidelity testing. As demonstrated in [5], HiTL automation can reduce test execution from weeks of manual testing to a few dozen hours of automated execution time by executing tests 24/7 at computer speed on real hardware. While this significantly closes the test execution time gap to SiTL testing, the tests are still executed in real time and the cost of system hardware will limit the number of available HiTL test systems. HiTL tests are as close as we can get to operating the actual system, so are typically the final tests to certify the system is operating as desired, testing both software and assembled hardware, before releasing the system to the field.

The benefits of HiTL and SiTL can be combined to provide an intermediate solution. Often the system under test is composed of hardware components, one or more primary processing units and many peripheral sensors and actuating devices producing system outputs or performing actions. An HiTL system can focus on testing the processing unit(s) in isolation, providing a digital twin of the other hardware the processing unit(s) would interact with. While this approach can still suffer with a simulation possibly being a slightly less perfect representation of real hardware, it can significantly reduce the dependence on system hardware, with the system software still operating on the same processing hardware it would be installed on in the field.

Software coverage analysis is typically included in automated testing as a part of SiTL testing. In projects with a low level of rigor, the coverage analysis may simply show how much of the code is being exercised by a test. In higher levels of rigor, tests instead verify the implementation against the requirements and, through traceability, help to identify requirements without code or code without requirements, in addition to identifying untested code. Based on the level of rigor, coverage may expect each line of code to be executed (statement coverage), each branch to be executed (branch coverage) or each condition of each branch to be exercised (Multiple Condition/Decision Coverage or MC/DC). At the highest levels of rigor, missing test coverage will prevent the CI/CD pipeline from progressing, even with all tests passing.

The CI/CD pipeline orchestrates test execution to optimize the use of both scalable SiTL resources and limited HiTL hardware. A typical test execution schedule varies depending on the time required to execute each stage. An example test execution

schedule may look like:

Every commit Module level tests verify that modified modules still operate as expected

Nightly/continuous Integration level tests execute tests on the full software suite without simulating real hardware

Weekly/release candidates System-level SiTL regression suites run in parallel across cloud instances

On-demand HiTL tests can be requested when investigating specific issues, validating timing-critical changes or to certify a full system release

The pipeline scheduler manages resource allocation, prioritizing critical tests and scaling cloud-based SiTL resources dynamically, including managing the sharing of resources between multiple projects, while efficiently utilizing limited HiTL hardware. This can even include running full HiTL or SiTL system tests earlier than otherwise scheduled if resources become available. From the developer's perspective, the infrastructure is transparent, the same test submission interface works whether tests execute in cloud SiTL environments or physical HiTL labs.

3.8. Deployment

Software deployment provides a wide range of methods to transfer new software to the system and each method presents distinct security challenges. Physical deployment requires securing the programming interface and validating the authenticity of the media used for updates. Wireless deployment introduces network security concerns, requiring encryption, authentication, and integrity verification to ensure only authorized updates are installed. Mission-critical systems must carefully balance the operational benefits of remote updates against the increased attack surface they create.

For CI/CD integration, the deployment method must align with the pipeline's automation capabilities. Systems designed with OTA update capabilities can integrate deployment as an automated pipeline stage, with cryptographic signing of artifacts and automated verification of successful installation. Systems requiring physical media updates may use the pipeline to generate signed update packages and deployment documentation, with the physical deployment step performed manually. The level of deployment automation achievable depends fundamentally on the system's built-in update mechanisms, a CI/CD pipeline cannot automate deployment to a system that does not support the same update method.

An additional benefit is the automation of the pipeline can be used to generate documentation of the processes followed for developing and testing the system. This documentation can satisfy most to all of the documented evidence required by a certification authority.

4. CONCLUSION

As we have seen, historic manual development and testing methods are too slow to keep up with the rapidly changing requirements for modern missions. The use of automation tools can be applied to every stage of software development and deployment to create a CI/CD pipeline able to keep up with the pace of modern deployments. While the up front cost of configuring a reliable CI/CD pipeline can be significant, the long term benefits of a repeatable, automated process quickly begin to outweigh the initial investment. Once established, a CI/CD pipeline provides the confidence that every software change has been subjected to the same rigorous and repeatable process, reducing costs through earlier detection of issues and reducing the

risk of any issues reaching the field.

The framework presented in this paper is intentionally flexible. Not every project requires the same level of rigor. The tools and processes described can be scaled to match the level of rigor or safety criticality, budget, and timeline of a given project. Early prototype development may benefit from basic automated testing and version control, while safety-critical certification may require the full suite of qualified tools, formal requirements with independently developed tests, bidirectional traceability, and CCB-approved change tracking. The key insight is that the same CI/CD automation can benefit any project, only the rigor of each stage differs.

SiTL testing plays a particularly important role in making CI/CD practical for mission-critical systems. By enabling extensive parallel testing on scalable server infrastructure, SiTL allows development teams to run comprehensive regression suites that could be significantly limited by the hardware availability HiTL systems depend on. Despite the benefits of SiTL testing, HiTL testing remains essential for final validation, ensuring the software performs correctly on the actual hardware it will operate on in the field, and in some certification contexts is required to demonstrate that both software and hardware are functioning as intended. HiTL environments can be made more accessible through improved automation and hardware simulation. Together, SiTL and HiTL testing provide a complementary approach that balances development speed with the high confidence required for mission-critical deployment.

Ultimately, the goal of CI/CD for mission-critical systems is not simply to move faster, but to move faster reliably and repeatably. A well-configured pipeline does not just provide tested artifacts, it creates a well documented, repeatable process that provides certifiable evidence of the quality of the system at every stage of development. When systems must adapt faster to changing mission requirements, while still operating correctly, CI/CD pipelines provide a vital tool for ensuring the software that is deployed is what the mission requires.

5. REFERENCES

[1] "DO-178C: Software considerations in airborne systems and equipment certification," RTCA, Inc., Washington, DC, Tech. Rep., December 2011, available for purchase from RTCA. [Online]. Available: https://my.rtca.org/productdetails?id=a1B36000001IcmqEAC

[2] "ISO 26262-1:2018: Road vehicles – functional safety – part 1: Vocabulary," International Organization for Standardization, Geneva, Switzerland, Tech. Rep., 2018, available for purchase from ISO. [Online]. Available: https://www.iso.org/standard/68383.html

[3] "MIL-STD-882E: Department of defense standard practice: System safety," U.S. Department of Defense, Tech. Rep., February 2026, available online. [Online]. Available: https://safety.army.mil/Portals/0/Documents/ON-DUTY/SYSTEMSAFETY/Standard/MIL-STD-882E-change-1.pdf

[4] "Joint software systems safety engineering handbook," U.S. Department of Defense, Tech. Rep., May 2026, available online. [Online]. Available: https://mail.system-safety.org/Documents/SOFTWARE_SYSTEM_SAFETY_HDBK_2010.pdf

[5] M. Lingg, T. J. Kushnier, R. Proenza, H. Paul, B. Grimes, and E. Thompson, "Multi-level hardware-in-the-loop test api for hardware-software integration testing," GVSETS 2022, August 2022. [Online]. Available: http://gvsets.ndia-mich.org/publication.php?documentID=926

[6] "Implementing a modular open systemsapproach in department of defense programs," Office of Systems Engineering and Architecture, Tech. Rep., February 2025. [Online]. Available: https://www.cto.mil/wp-content/uploads/2025/03/MOSA-Implementation-Guidebook-27Feb2025-Cleared.pdf

6. CONTACT INFORMATION

Michael Lingg, PhD
Principal Research Engineer
Array of Engineers
2350 Oak Industrial Dr NE #1
Grand Rapids, MI 49505
michael.lingg@arrayofengineers.com
LinkedIn:
www.linkedin.com/in/michael-lingg-8b881b11

7. ACRONYMS

CCB	Change Control Board
CI/CD	Continuous Integration/Continuous Deployment
COTS	Commercial Off the Shelf
HiTL	Hardware-in-the-Loop
IED	Improvised Explosive Device
MC/DC	Modified Condition/Decision Coverage
MOSA	Modular Open Systems Approach
MRAP	Mine-Resistant Ambush Protected
OTA	Over-the-Air
SiTL	Software-in-the-Loop
TDD	Test-Driven Development
UML	Universal Modeling Language
VM	Virtual Machine