https://doi.org/10.48047/AFJBS.8.06.2026.19-29

African Journal of Biological Sciences

Journal homepage: http://www.afjbs.com

Research Paper Open Access

# Redefining Verification Velocity in Embedded Medical Devices Through Scalable Test Automation Using AI-Enhanced Testing Techniques and Hardware-in-the-Loop Capabilities

**Bhavya Thota[1], Prashanth GV[2], Supreetha Keminje[3], Rama Chandrudu Boya[4]**

[1] *Baxter Innovations & Business Solutions Ltd, Bengaluru, India*

[2] *Baxter Innovations & Business Solutions Ltd, Bengaluru, India*

[3] *Baxter Innovations & Business Solutions Ltd, Bengaluru, India*

[4] *Baxter Innovations & Business Solutions Ltd, Bengaluru, India*

1st bhavya_thota@baxter.com; 2nd prashanth_gv@baxter.com;

3rd supreetha_keminje@baxter.com; 4th rama_chandrudu_boya@baxter.com

Volume 8, Issue 6, June 2026

Received: 02 March 2026

Accepted: 14 April 2026

Published: 06 June 2026

doi:10.48047/AFJBS.8.6.2026.19-29

***Abstract*— The rapid evolution of embedded medical devices has significantly increased system complexity, intensifying the need for faster yet rigorous verification processes that ensure safety, reliability, and regulatory compliance. Traditional verification approaches struggle to scale under growing functional requirements, tight development schedules, and stringent medical standards. This paper presents a unified scalable verification framework that redefines verification velocity through the integration of automated test execution, artificial intelligence (AI)–enhanced test generation, and advanced Hardware-in-the-Loop (HIL) capabilities for embedded medical systems. The proposed architecture is built on four cohesive layers: (i) a domain-specific language that formalizes clinical, safety, and regulatory requirements into executable verification logic; (ii) an AI-assisted test generation engine that expands test coverage by identifying edge cases and latent failure scenarios; (iii) an HIL orchestration layer that enables synchronized validation of embedded software behavior against real-time physical device responses; and (iv) an adaptive reporting and documentation engine that generates traceable, audit-ready verification artifacts aligned with medical regulatory standards. Experimental results from a representative embedded medical device platform demonstrate substantial improvements in verification throughput, test coverage, and system robustness compared to conventional manual and simulation-only approaches. The framework further introduces automated recovery mechanisms, continuous real-time state monitoring, and exception-handling workflows tailored for safety-critical medical interfaces. By unifying intelligent automation with hardware-coupled validation, this work establishes a practical and scalable paradigm for next-generation verification of embedded medical devices, accelerating development cycles while strengthening product quality and patient safety.***

***Keywords-*** *Artificial intelligence (AI)–enhanced test generation, embedded medical devices, hardware-in-the-loop (HIL), verification automation.*

## I. INTRODUCTION

Embedded medical devices are increasingly software-driven and safety-critical, demanding high levels of reliability, efficiency, and verification rigor. With rapidly increasing system complexity and compressed development timelines, there is a critical need for *agile verification processes that enable early defect detection within the development lifecycle*, thereby significantly reducing rework cost while improving product confidence and overall quality. *Shift-left* engineering practices emphasize early validation and continuous feedback, enabling faster identification and resolution of defects and minimizing costly late-stage corrections [1].

To achieve this, requires a fundamental transformation in verification practices. This paper proposes a technology-agnostic, unified, and scalable verification framework designed to redefine verification velocity for embedded medical devices.

Following are the highlights of the proposed framework

- *A Ubiquitous Language*, which translates the requirements into executable test cases in a structured and consistent way to describe the system behavior which helps readability across cross functional teams/stakeholders.
- An efficient means to generate tests by *covering all the edge cases* and supported variability without leaving any gap during the test design.
- Provide a capability to execute tests on *targeted hardware* emulating various conditions which are hard to simulate in real world conditions.
- Supporting *self-healing* capabilities in reducing maintenance efforts to adapt to the system changes and increase resiliency.
- Comprehensive *reporting* with end-to-end traceability ensuring compliance towards regulatory standards.

In addition, the approach reduces the development costs and in turn provides the overall safety and quality of medical devices.

## II. LITERATURE REVIEW AND RESEARCH GAPS

The verification of embedded medical devices has evolved significantly, with the adoption of modern engineering practices which improve early defect detection, scalability, and system reliability. Recent research *emphasizes Ubiquitous Language - Domain Specific Language (DSL), AI Test Generation with Variability, Testing on Targeted Hardware, Self-Healing, Comprehensive Reporting validation* to address the growing complexity of medical embedded systems. Despite these advancements, existing approaches limit their ability to deliver end-to-end verification efficiency and regulatory compliance.

### *A. Ubiquitous Language (DSL)*

The existing paper provides the importance of using Ubiquitous language (DSL) during test design but lack semantic expressiveness for verifying safety-critical medical devices. Additionally, they overlook persona-based behavioral contexts, limiting their ability to model real-world usage in hospital environments. This creates a gap in achieving context-aware, semantically rich validation of embedded medical device behavior [5].

### B. *AI Test Generation with Variability*

Existing literature emphasizes the use of AI and Large Language Model (LLM) based approaches to enhance verification capabilities; they lack a domain-specific context layer to constrain model outputs and ensure alignment with safety-critical requirements [2]. Additionally, existing methods do not effectively capture variability in medical products, limiting their applicability in context-aware and robust verification [4].

C. *Testing on Targeted Hardware*

Existing studies highlight the advantages of testing the product on the targeted device, but they lack defined methodologies and frameworks to achieve this in practice. Additionally, there is a minimal focus on embedding testing capabilities within the product itself, limiting the ability to perform continuous, real-time verification in embedded systems [6].

D. *Self-Healing*

The existing literature highlights the usage of self-healing capabilities in test execution; it primarily focuses on tool-driven solutions and neglects the development of self-healing as an integral part of custom frameworks. This limits flexibility, domain adaptation and compliance particularly in safety-critical environments [3].

*E. Comprehensive Reporting*

While existing literature highlights the importance of reporting in regulated industry, it lacks comprehensive approaches for ensuring end-to-end traceability, compliance alignment and completeness of evidence artifacts required for audits and certification in healthcare systems [7].

## III. RESEARCH OBJECTIVES

The increasing complexity of embedded medical devices, verification practices identified in the literature, necessitates a *unified, scalable, and intelligent approach to verification*. Existing methods mentioned in the literature review does not cover seamless integration across *shift-left verification automation*, *AI-driven context-aware testing, target-based verification, and regulatory-compliant reporting*, thereby limiting verification efficiency, scalability. Addressing these limitations requires a comprehensive framework that enables *early defect detection, accelerated feedback, and end-to-end traceability*, while supporting *real-time system verification*.

To achieve this, the objectives of this research paper are defined as follows:

*A. Ubiquitous Language (DSL*)

The proposed research advances the state of the art by introducing a semantically enriched, persona-aware DSL framework, enabling context aware modelling for real hospital workflows and ensuring that verification aligns with safety-critical behavioural expectations.

*B. AI Test Generation*

The proposed framework introduces a domain-specific knowledge layer integrated with AI, enabling context-bounded, bias-controlled test generation while systematically capturing product variability and usage scenarios, directly bridging the identified gap in context-aware verification.

*C. HIL Integrating in Product*

The research introduces a methodology for integrating Hardware-in-loop (HIL) capabilities within the embedded product itself, enabling real-time verification and transforming testing from an external activity into a built-in system capability, thereby addressing the absence of practical implementation frameworks.

*D. Self-Healing and Adaptive Automation*

This research shifts the paradigm by embedding self-healing as a core capability within the custom automation framework, enabling context-aware, explainable, and extensible healing mechanisms. Tailored

for embedded medical systems, thus overcoming the lack of framework level design highlighted in the gap. Additionally, the framework ensures efficient and scalable execution through *infrastructure optimization* using containerization and parallel processing. It enhances performance through efficient resource usage, selective test execution, and continuous monitoring to prevent bottlenecks and reduce costs.

*E. Regulatory-Compliant Reporting Mechanism*

The research proposes a traceability-driven, compliance-aware reporting framework that ensures complete linkage between requirements, tests, results and evidence artifacts enabling audit-ready, real-time reporting and directly addressing the identified deficiency in comprehensive regulatory support.

## IV. METHODOLOGY

The Test Automation Framework adopts a layered architectural model, where interconnected components operate sequentially to provide structured, scalable, and intelligent verification of embedded medical devices.

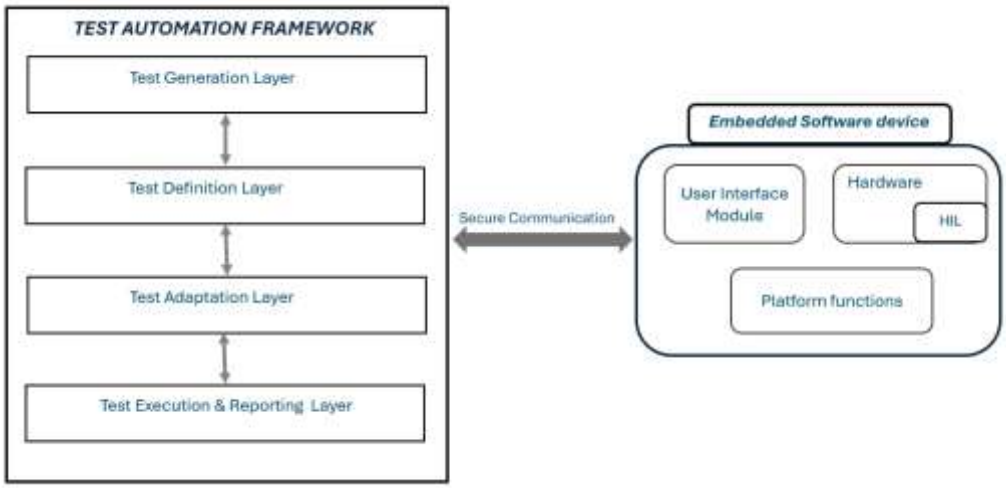



FIG. 1 TEST AUTOMATION FRAMEWORK ARCHITECTURE

The flow begins with the *Test Generation Layer*, which creates test scenarios using *domain knowledge and AI-driven logic*. These generated tests are then passed to the *Test Definition Layer,* where they are represented in *step definitions formats* with defined inputs, expected outcomes, and requirement mappings, ready to be used by downstream execution layers. The defined tests are further processed by the *Test Adaptation Layer*, which dynamically adapts them based on the target embedded device configuration, platform variations, or hardware constraints, ensuring that the tests are compatible with different system conditions. Finally, the tests move into the *Test Execution & Reporting Layer*, where they are executed against the embedded medical device and the results, logs, reports are generated.

The framework communicates with the *Embedded Software Device via a secure interface, ensuring controlled and traceable interaction*. In the proposed architecture, the HIL capability is embedded within the device under test, enabling internal signal simulation through its hardware layer alongside platform functions

and UI modules. This integration allows low-latency, real-time verification where inputs are generated internally and responses are captured seamlessly. Additionally, the Infrastructure Optimization layer ensures efficient scalability, resource utilization, and performance across the testing pipeline.

A. *Test Generation Layer*

This Layer represents a structured, AI-driven pipeline where all the components in the diagram are interconnected to transform raw inputs into validated test scenarios.

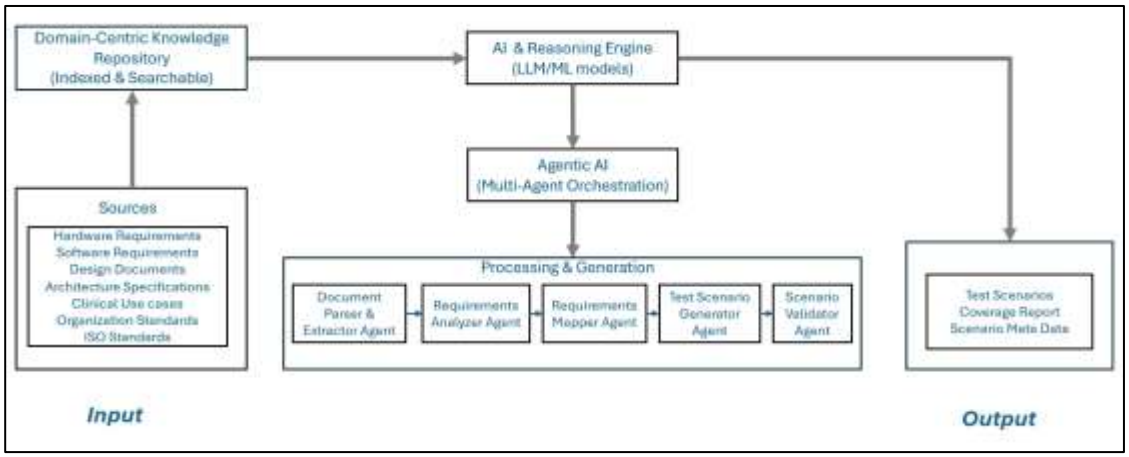



FIG. 2 TEST GENERATION LAYER

The *AI & Reasoning Engine LLM/Machine Language (ML) models* takes multiple input sources which includes hardware and software requirements, design documents, architecture specifications, clinical use cases, organizational standards, and ISO regulations which are consolidated into a Domain-Centric Knowledge Repository. This repository acts as a central hub, where all inputs are organized, indexed, and made searchable, so that relevant context can be retrieved efficiently. The AI & Reasoning Engine interprets the requirements, while the Agentic AI (multi-agent orchestration) orchestrates the overall workflow. From there, the data flows into the Processing & Generation pipeline, which follows a well-defined sequence as shown in the diagram: *Document Parser & Extraction Agent → Requirements Analyzer Agent → Requirements Mapper Agent → Test Scenario Generator Agent → Scenario Validator Agent*. Each stage enriches the data first extracting structured information from documents, then understanding system behaviour, linking requirements to design, risks, and standards, and finally generating comprehensive test scenarios.

For example, if a requirement states that a device should shut down when temperature exceeds a threshold, the pipeline processes it through these stages to automatically generate multiple scenarios, including normal operation, boundary conditions, overheating cases, sensor failures, and noise conditions. These scenarios are then passed to the Scenario Validator Agent, which verifies correctness, completeness, and compliance with defined rules and standards.

Finally, the validated outputs flow into the Output Layer, producing test scenarios, coverage reports, and scenario metadata, ensuring full traceability back to the original requirements. In simple terms, the diagram represents a connected pipeline where data flows step by step from sources to AI processing to test generation getting refined at each stage until it becomes reliable, validated test cases. This structured and interconnected design ensures that raw documentation is converted into intelligent, traceable, and high-quality test artifacts, improving coverage, consistency, and overall verification efficiency.

B. *Test Definition Layer*

This layer stores and organizes structured test specifications. It contains test scenarios defined in Gherkin format (Given–When–Then), making them clear, standardized, and easy to understand for both technical and non-technical users.

For example: Given the ventilator is ready and running, When Airway pressure exceeds threshold, Then the ventilator should trigger a high-pressure alarm.

Along with Gherkin scenarios, this layer also *includes Step Definition* files, which map each readable step to the underlying automation logic used during execution. In addition, it maintains *test variability* in the form of structured test data, allowing different input conditions such as *normal, edge, and failure cases*. It also includes *traceability* information, where each test is tagged with details like *requirement ID, risk level, and priority, ensuring full linkage between requirements, tests, and results*. These Gherkin feature files, test data, and traceability metadata are then consumed by downstream layers to drive actual test runs using BDD frameworks such as Cucumber, Spec Flow, or Behave.

*C. Test Adaption Layer*

This layer takes the test cases created in the previous step and makes them ready to run on different embedded devices. It adjusts each test based on the device type, system setup, and hardware conditions, so the same test can work across multiple devices without needing to be changed. In simple terms, it prepares the test to "fit" the target system.

Instead of changing the test every time, this layer handles all device-specific differences internally, allowing the original test logic to remain the same and reusable. The earlier layer focuses on deciding what to test, while this layer focuses on how to run it on a particular device or application enabling the proposed framework to be scalable.

To achieve this, the framework uses different helper modules (libraries). For example, GUI libraries handle interactions with screens, so tests are not affected by UI changes. HIL libraries simulate hardware behaviour such as sensor values, noise, or faults. Business logic libraries adjust behaviour based on device versions or configurations, allowing the same test to work across different product variants.

The framework also includes test management libraries that control which tests should run, in what order, and under what conditions, including handling retries and environment-specific setups. In addition, secure communication modules ensure reliable and safe communication with the embedded medical device during test execution.

*D. Test Execution and Reporting Layer*

This Layer is responsible for running tests in a controlled way and capturing reliable evidence of what happened. This layer turns test definitions into actual test runs and produces outputs that engineers, reviewers, and auditors can trust. This layer answers what ran, what passed, what failed, and why, providing necessary evidence.

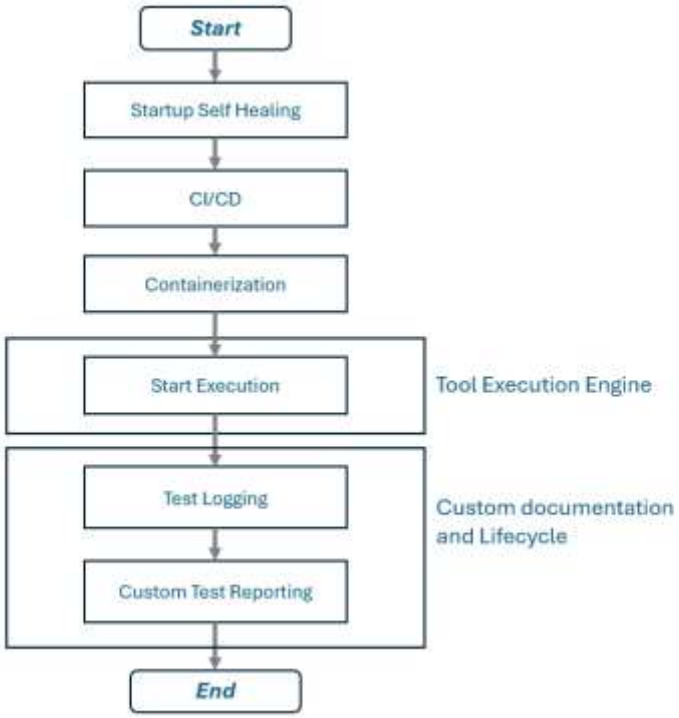



FIG. 3 TEST EXECUTION AND REPORTING LAYER

1) *Self-Healing AI Engine:* The proposed framework introduces a *tool-agnostic, execution-free self-healing AI engine* designed to *proactively verifies* and optimize element locators prior to test execution. Unlike conventional approaches that rely on runtime failures to trigger healing, this framework adopts a *pre-validation strategy*, wherein all locators defined in the object repository are systematically evaluated at the start of the testing cycle. This proactive approach ensures that potential locator inconsistencies are identified and mitigated in advance, thereby improving reliability while reducing runtime overhead.

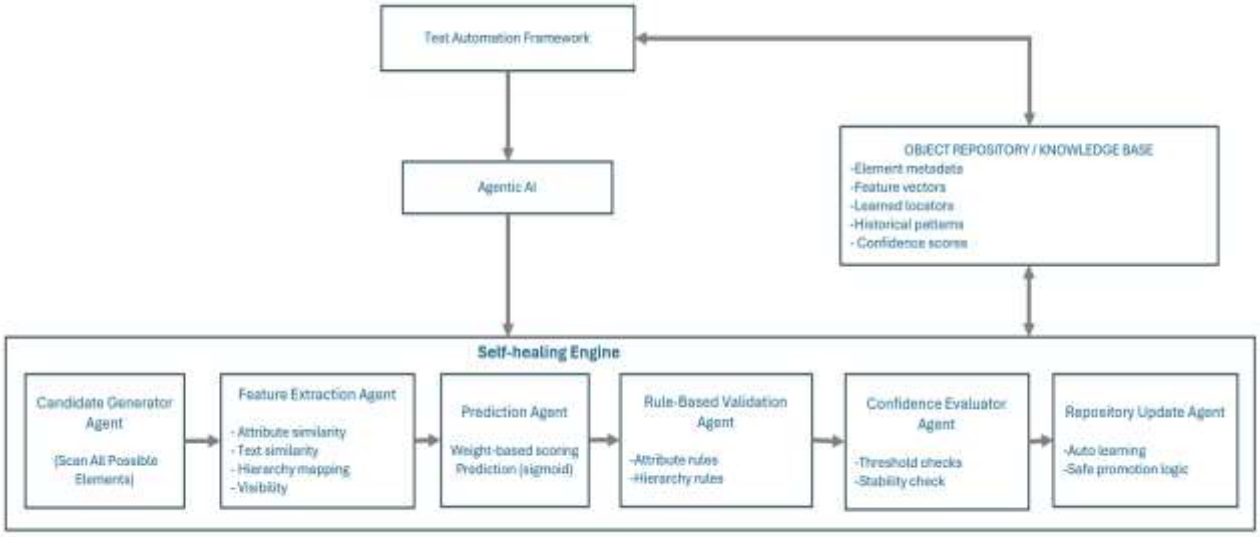



FIG. 4 SELF-HEALING AI ENGINE ARCHITECTURE (TOOL-AGNOSTIC, NO EXECUTION)

At the beginning of the testing process, the Test Automation Framework invokes the *Agentic AI*, which orchestrates the execution of the *Self-Healing Engine* in a non-execution verification mode. Instead of interacting with the application or system in real time, all locators are analysed using *pre-existing metadata* which is retrieved from the Object Repository / Knowledge Base, which stores element definitions, feature vectors, learned locators, and historical patterns.

The process starts within *the Self-Healing Engine flow* as shown in the diagram, beginning with the *Candidate Generator Agent,* which scans all possible elements available in the system under test to create a comprehensive pool of potential matches. This ensures that even if locators change due to UI or structural updates, all possible candidates are considered. These candidates are then passed to the Feature Extraction Agent, where key attributes such as element type, text similarity, visibility, and *hierarchical relationships such as parent–child structures are analysed a*nd converted into *structured numerical feature vectors*, enabling efficient computational evaluation.

The *feature vectors* are then processed by the *Prediction Agent*, which employs a lightweight machine learning model based on *weighted linear combinations* followed by a sigmoid activation function to *assign a confidence score to each candidate, indicating how closely it matches the expected element*. The initial weights are derived from domain knowledge and are progressively refined through an adaptive learning mechanism embedded within the system, allowing continuous improvement in prediction accuracy.

To further enhance robustness, the model predictions are passed through the *Rule-Based Agent*, which enforces strict deterministic constraints. These constraints include mandatory tag consistency, validation of hierarchical relationships, and adherence to minimum similarity thresholds. This layer acts as a safeguard, filtering out candidates that may achieve high probabilistic scores but fail to satisfy essential structural and contextual requirements.

Following validation, the shortlisted candidates are evaluated by the *Confidence Evaluator Agent*, which performs threshold-based filtering and stability analysis. In the absence of execution-based *verification*, this component ensures correctness by verifying that the candidate meet high-confidence thresholds and maintain consistency across multiple evaluation cycles. This repeated validation process effectively establishes reliability without requiring direct interaction with the system under test.

In the final stage, the *Repository Update Agent* implements a controlled and autonomous learning mechanism. Rather than immediately replacing existing locators, the system monitors the stability and consistency of validated candidates over successive observations. Once a candidate demonstrates sufficient reliability, it is promoted as the updated locator within the *Object Repository / Knowledge Base*. This repository maintains comprehensive information, including element metadata, feature vectors, learned locator mappings, historical patterns, and confidence scores, thereby enabling continuous system evolution.

Overall, the integrated design presents a scalable, extensible, and robust architecture that dynamically adapts to changes in user interfaces and application structures without requiring runtime execution or manual intervention. By combining feature-based analysis, machine learning–driven prediction, rule-based validation, and controlled learning, the framework delivers a balanced approach that ensures both high accuracy and long-term maintainability in dynamic testing environments.

2) *CI/CD Integration:* The proposed framework incorporates CI/CD integration, a foundational approach to enable *continuous automation*, *early-stage verification* of embedded systems, further strengthened by a *Shift-Left testing* strategy. This supports a continuous and feature-level *verification* model, where test execution is *triggered automatically on every code commit*, ensuring that developers validate changes at the *earliest stage of development rather than waiting for downstream integration*. CI/CD pipelines are built using tools such as *Jenkins, Azure DevOps, or GitHub Actions* which are configured to orchestrate the complete testing lifecycle. For example, when a developer commits a new feature or modification, the pipeline automatically triggers *verification*, executes targeted tests within isolated containers, and provides immediate feedback on the impact of the change. This approach ensures that defects are identified and resolved at the feature level, significantly *reducing defect propagation into later stages such as integration* or release. By combining automated test triggering, *proactive verification*, and feature-level feedback, the CI/CD integration not only accelerates development cycles but also improves software *quality, traceability* and verification velocity.

3) *Containerization*: The proposed framework utilizes containerization as a *core approach to enable scalable, consistent, and parallel test execution environments*, further strengthened by integrated orchestration capabilities. Containerization supports an environment-independent verification model, where each test instance is executed within an isolated runtime, eliminating dependencies and configuration inconsistencies. The framework leverages predefined container images, which encapsulate all required test tools, configurations, and dependencies, ensuring that every execution environment is standardized and reusable. Rather than manually managing environments, the framework handles custom orchestration of these container images, dynamically provisioning and coordinating test runs based on execution requirements.

   Docker are used to build and manage these container images, while orchestration is enabled using platforms such as Kubernetes deployed on cloud infrastructure like AWS (e.g., Amazon EKS). Kubernetes manages container lifecycle operations including scheduling, scaling, load balancing, and fault tolerance, allowing the system to automatically spawn, monitor, and terminate test execution pods. For example, during a large regression cycle, Kubernetes can scale horizontally to create hundreds of container instances across AWS nodes, each executing independent test scenarios in parallel against the embedded medical device. This orchestration ensures optimal resource utilization and high availability while maintaining execution isolation. Additionally, containerization guarantees environment reproducibility, as each test is executed in a controlled and consistent setup derived from the same base image. This eliminates environment-related failures and reduces configuration drift across executions. By combining predefined container images, automated orchestration and cloud-native deployment through AWS and Kubernetes, the framework achieves highly scalable, reliable, and parallel test execution, significantly improving verification velocity by reducing execution time, eliminating setup delays, and enabling seamless scaling of test workloads.

4) *Tool Execution Engine:* The proposed framework acts as the central orchestration layer responsible for managing the complete test execution lifecycle. It triggers test runs based on predefined configurations and interacts with the embedded system using structured test inputs. The engine coordinates environment setup, execution sequencing, synchronization with the device, and teardown activities to ensure controlled and consistent execution. It also includes configurable retry mechanisms to handle transient failures and improve execution stability. Additionally, it integrates logging and reporting to provide traceable and reliable verification results.

5) *Custom Documentation and Lifecycle:* The framework supports custom documentation and lifecycle integration to ensure complete *traceability and audit readiness* throughout the testing process. It links test cases to requirements, execution results, and supporting evidence, ensuring that every validation step is fully documented. For example, a test case can be directly mapped to a requirement, with its execution results and logs preserved across releases. This ensures that *verification artifacts are consistent*, *version-controlled*, and *easy to retrieve for audits* or *reviews*. This can be achieved by test management tools like ALM, Polarion. By integrating testing with the overall development lifecycle, the framework improves transparency, compliance, and long-term maintainability.

6) *Test Logging Module:* The framework includes a test logging module that captures detailed execution information in a consistent and structured manner. It records all interactions such as *commands executed, responses received, errors*, warnings, and timestamps. These logs are normalized across different interfaces, ensuring consistency regardless of the system being tested. For example, during execution, *every signal sent to the embedded device* and its corresponding response is logged for traceability. This enables *easy debugging, root cause analysis*, and validation of system behaviour. By maintaining detailed and consistent logs, the framework improves troubleshooting efficiency and ensures reliable test *traceability*.

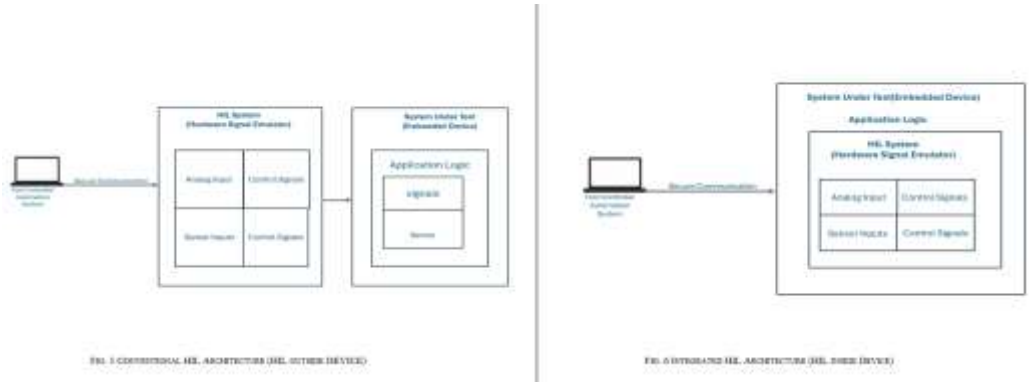
7) *Custom Reporting:* The framework provides custom reporting to represent the test results in a clear, *structured, and easy-to-review format*. Instead of raw outputs, reports are generated in a standardized way that includes *pass/fail summaries*, *step-level executio*n details, and *supporting evidence* such as logs, *screenshots or captured data*. For example, a test report can show which steps passed, where a failure occurred, and link it directly to the corresponding requirement or test case. This makes it easier for engineers and auditors to quickly understand results, track issues, and ensure compliance. Overall, custom reporting improves *visibility, simplifies analysis*, and supports better decision-making.

8) *Infrastructure optimization:* It focuses on how tests are executed, not on what is being tested. This layer is necessary for large scale, repeatable test execution and prevents the test framework itself from becoming a bottleneck and minimize the commercial tools cost. Below are the key features.

- *Efficient Use of Compute resources* –Test Automation framework should apply Containerization to use the resources when needed and release when finished. This prevents overloaded test machines and long test queues.
- *Failure Isolation and Automatic Cleanup* – Test Automation framework must prevent cascading failures and improve stability by ensuring the same environment for every test run, no left-over state from previous runs, reinitialize interfaces automatically and no dependency on manual setup
- *Scalable Test Execution* – Test Automation framework should support running more test without redesign, parallel execution and easy addition of new embedded devices.
- *Smart Test Execution Control* – Test Automation Framework must provide the support for running only the tests that matter for a given Change. This can be achieved by tags to execute subsets based on feature, risk level, requirements, smoke vs regression, etc.
- *Execution Monitoring and metrics* – Measure the framework behavior over time to track execution duration per test.

*E. HIL Signal Emulators*

Hardware-in-the-Loop (HIL) testing enables verification of embedded systems by simulating real-world input signals such as sensor readings, without relying on physical hardware. Traditionally, this is achieved using an external simulation setup; however, *the proposed framework introduces an integrated approach where HIL capability is embedded directly within the embedded device under test*.



1) *Conventional HIL Architecture*: In this approach, the HIL system is *external and detached from the embedded device*, operating as a separate simulation setup. A test controller sends predefined scenarios to the HIL system, which generates electrical signals and feeds them into the device. The device processes these inputs, and outputs are captured and verified in a controlled loop (input - execution - output - validation). While this enables realistic testing without physical sensors, it introduces latency, limited internal visibility, higher cost, and scalability challenges.

2) *Integrated HIL Architecture:* In the proposed framework, the Hardware-in-the-Loop (HIL) capability is *directly embedded within the system under test*, eliminating the need for an external simulation setup. Instead of relying on a separate HIL system to generate inputs, the device itself is equipped with an internal HIL engine that can simulate real-world signals such as sensor readings, noise conditions, and fault scenarios. The testing flow in this architecture is simple and efficient, generating input →process within device →verify →output, all happening within the same system. The embedded HIL module injects simulated signals (e.g., analog values, digital triggers, abnormal conditions) directly into the internal data paths of the embedded software. This removes dependencies on external wiring, communication interfaces (such as UART, CAN, or SPI), and dedicated signal generation hardware. As a result, latency is minimized, synchronization issues are eliminated, and signal accuracy is improved because there is no physical transmission delay. From an implementation perspective, the HIL engine is integrated as part of the device's firmware, where it mimics hardware inputs by interfacing directly with modules such as ADC processing units, control logic, or sensor

interface layers. This allows the framework to simulate a wide range of operational conditions, including boundary values, environmental variations, and fault injection scenarios, in a controlled and repeatable manner.

For example, instead of connecting a physical temperature sensor, the system can internally simulate values like 25°C (normal), 80°C (high), or a sensor failure condition, and immediately validate whether the system responds correctly such as activating cooling mechanisms or triggering alerts. The integrated approach also provides deep internal visibility, allowing testers to observe intermediate processing states, decision logic, and internal data flows, which are not easily accessible in conventional external HIL setups. This enables faster debugging and more accurate validation. Additionally, because the HIL system is software-driven, it supports easy scalability and extensibility, allowing new test scenarios to be added without modifying hardware. Overall, the integrated HIL architecture significantly improves efficiency by reducing hardware complexity, lowering cost, enabling real-time validation, and supporting faster execution cycles. It provides a more flexible and scalable testing solution, particularly suitable for complex and safety-critical embedded systems.

## V. RESULTS

A. *HIL Signal Emulators*

The integration of HIL emulators within the target device as proposed in the framework has significantly improved requirement validation coverage from 40% in manual simulation-based approach to 80% on targeted hardware. This improvement enhanced overall test completeness and confidence in system performance under real-world operating conditions enhancing the patient safety.

B. *Self-healing Capability and Resiliency*

The self-healing capability integrated within the proposed automation framework has significantly enhanced execution stability and reduced maintenance efforts. Test execution resilience improved from 0 to 90%, thereby ensuring more reliable and robust test execution even under evolving system conditions.

*C. Ubiquitous Language*

The integration of DSL-driven test design in the proposed framework has significantly improved consistency, reduced review comments and rework by 95%.

D. *AI*

AI-based test generation has enhanced functional coverage, variability coverage and overall productivity from 40 to 80%.

E. *Reporting*

The proposed framework enhanced audit compliance by reducing non-conformance incidents from 10% to 0%, ensuring complete adherence during audit cycles. This demonstrates the effectiveness of the framework in delivering fully compliant, audit-ready reporting.

## VI. SCOPE OF FUTURE STUDY

Future work may focus on strengthening the knowledge layer with richer contextual intelligence, while integrating enhanced AI-based security features and more advanced HIL-driven validation capabilities.

## VII. CONCLUSION

The proposed framework has brought in significant velocity improvements with desired verification process which enables the cross functional teams /stakeholders to review and give feedback early in the cycle. With HIL capabilities embedded on the targeted device has given flexibility to drive variability of data to cover more edge cases. Test execution success has significantly improved with tool agnostic self-healing capability in the framework. The proposed reporting framework significantly reduces internal and external audit findings by ensuring complete traceability from requirements to corresponding evidence, thereby improving overall validation completeness and regulatory adherence. Overall, the process is fruitful in delivering high quality of verification in safety critical medical devices enhancing the quality and safety of the patient.

## ACKNOWLEDGEMENT

The authors acknowledge the use of Microsoft Copilot for assistance with grammar correction and sentence rephrasing during the preparation of this manuscript. The use of this tool was limited to improving language clarity and readability. All ideas, methodologies, analyses, and conclusions presented in this work are solely those of the authors.

## REFERENCES

[1] G. Sharma, “Revolutionizing shift-left testing through an agentic AI framework: Enhancing software quality and digital trust”in Proc. Int. Conf. Intelligent Systems, Springer, 2026.
[2] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, “Combinatorial test generation for software product lines using minimum invalid tuples,” in Proc. IEEE Int. Symp. High-Assurance Systems Engineering, 2014.
[3] K. Qayyum, S. Ahmadi-Pour, and M. Hassan, “AI-driven self-healing in test automation: A review of autonomous testing systems,” in Proc. IEEE Conf. Automation Systems, 2025.
[4] S. R. Kumar and P. Sharma, “An AI-based smart test case generator for embedded devices,” in Proc. IEEE Int. Conf. Artificial Intelligence Systems, 2022.
[5] M. Becker and T. Kühne, “Formal verification of domain-specific languages for embedded systems,” in Proc. IEEE Int. Conf. Software Engineering, 2018.
[6] O. Bause, J. Werner, and O. Bringmann, “Hardware integration testing for smart video-based medical device prototypes,” in Proc. IEEE Conf. Embedded Systems, 2025.
[7] International Organization for Standardization, ISO 13485: Medical Devices—Quality Management Systems, ISO, Geneva, Switzerland, 2016.
[8] M. Abbas, M. Saadatmand, E. P. Enoiu, and B. H. Schlingloff, “State of test optimization for variability in industry,” in Proc. Int. Conf. Information Technology, 2025.
[9] IEC 62304, Medical Device Software—Software Life Cycle Processes, IEC, 2015.
[10] A. J. Watkins and J. R. Eddy, “Verification challenges in safety-critical embedded systems,” IEEE Embedded Systems Letters, 2019.
[11] S. Anand et al., “Automated test generation for embedded systems: An industrial perspective,” IEEE Design & Test, 2017.
[12] M. Törngren and U. Sellgren, “Hardware-in-the-loop testing of embedded systems,” IEEE Software, 2015.
[13] V. R. Hajari, A. Chhapola, O. Goel, and P. K. G. Pandian, “Automation strategies for medical device software testing,” Universal Research Reports, Aug. 2024.
[14] P. Nama, “Integrating AI in testing automation: Enhancing test coverage and predictive analysis for improved software quality,” World Journal of Advanced Engineering Technology and Sciences, 2024.
[15] R. Jetley, P. Jones, and J. Abraham, “Formal methods-based verification of medical device software,” Embedded Systems Design, 2010.
[16] V. R. Hajari et al., “Innovative techniques for software verification in medical devices,” International Journal for Research Publication and Seminar, 2024.