BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2026
http://excel.fit.vutbr.cz

Automated Testing of IoT Edge Devices and its Integration into CI/CD

Jan Šulák*

Abstract

High reliability is critical for IoT edge devices, yet testing multiple protocols without physical hardware remains challenging. This work presents an automated integration testing framework that streamlines testing, quickly identifies issues, and reduces the need for manual testing resources. The framework, called the Integration Test Engine, was developed in cooperation with the Logimic company. It uses hardware mocking and schema-based validation to simulate and verify complex communication flows, such as MQTT and Modbus, without requiring physical devices. The solution features interactive HTML reporting and full integration with Logimic's CI/CD infrastructure. By enabling early regression detection and rapid gateway configuration validation, this framework enhances software reliability, speeds up development cycles, and ensures secure, consistent deployments across edge gateways.

*xsulak07@vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The Internet of Things demands exceptional reliability from edge devices in distributed environments. Ensuring software stability across diverse protocols necessitates automated testing to minimize system failures.

This demand for reliability further exposes a critical issue: insufficient integration testing capabilities for IoT gateways. For instance, current setups fail to simulate hardware boundaries, creating a gap in realistic testing. To address this gap, a suitable solution must isolate environments, manage dependencies dynamically, and integrate into CI/CD pipelines.

However, in contrast to the outlined requirements, existing methods in Logimic's testing infrastructure primarily run isolated unit tests or manual tests. Furthermore, the tests merely stub dependencies instead of exercising them realistically. Integration tests that traverse genuine system boundaries are currently absent.

To overcome these constraints, the proposed Integration Test Engine introduces a layered framework for hardware interface simulation and systematic test orchestration. Leveraging specialized test doubles, the system replaces physical environments to deterministically evaluate the orchestration of logic and protocol handling.

The primary contribution is an extensible testing system that interfaces with any standard edge gateway and is integrated into CI/CD pipelines. The framework validates complex communication without physical hardware and generates interactive reports, enabling earlier detection of integration issues and faster troubleshooting.

2. Architecture

The Integration Test Engine establishes a robust architecture. It places the framework directly between edge gateways and their external environments. Specialized test doubles replace external systems, enabling comprehensive integration testing without the constraints of physical hardware. As presented on Figure 1, the architecture centers on three abstraction layers: Tested Module, Mocked Module, and Validation Module. The Tested Module orchestrates test execution by instantiating and wiring together mock and validation components. The Mocked Module establishes connections and issues protocol-specific commands to the System Under Test (SUT).

As the SUT processes requests, it emits asynchronous responses. The Mocked Module captures these responses and buffers them in an internal message queue. This queuing mechanism ensures that the synchronous validation loop receives all protocol outputs. It does so without blocking gateway operations.

3. Message Validation Algorithm

The framework uses a Message Validation Algorithm, shown in Figure 2, to verify SUT outputs. The process initiates by routing expected messages from the test case into the Validation Module. A central loop continuously polls the Mocked Module's queue to retrieve intercepted messages. If the queue is empty, the algorithm waits 50 milliseconds before retrying, continuing until all expectations are successfully validated or a timeout elapses. Upon retrieving a message, the algorithm applies a two-layer validation sequence.

The first layer executes schema validation, which enforces structural conformance by checking data types, string formats, required fields, and numeric ranges. Using schemas ensures runtime type safety. IoT environments produce volatile data, so the schema acts as a partial oracle. It asserts only semantically relevant fields and ignores non-deterministic content like timestamps, similar to the partial oracles described by Barr in his paper [1].

If the message passes this check and includes an expected payload, the second layer runs a rigorous data comparison. Data errors are prioritized over general schema mismatches. This yields highly informative debugging insights.

4. Tested Modules

As shown in Figure 3, the test engine supports validation across industrial edge gateways, with a focus on the Logimic Edge and Teltonika TRB140[1]. The framework evaluates specific background daemons operating within these devices. For the LgmcEdge gateway, the GwStatus module reports hardware telemetry, and the Scheduler module manages timed tasks. Both modules interface over the cloud MQTT[2] protocol. The Serial module translates raw hardware data, interfacing simultaneously over cloud MQTT and physical serial connections. The LiftManager orchestrates vehicle lift operations. It concurrently manages cloud MQTT for upstream telemetry, local MQTT for display synchronization, and Modbus[3] TCP for interacting with physical lift controllers. For the TRB140, the framework targets the InputOutput module. It validates digital states using MQTT and Modbus protocols.

5. Report Dashboard

The reporting subsystem generates an interactive Report Dashboard as a fully self-contained, single-page HTML document. As shown in Figure 4, the dashboard features a persistent sidebar that displays the test counts for passed and failed tests. The initial page displays a statistical overview. This overview presents high-level metric cards. It also shows per-suite donut charts and bar charts, providing an immediate understanding of suite stability.

Selecting a suite from the dashboard shows a final report, as illustrated in Figure 5. The report is organized into three hierarchical collapsible levels: configurations, test cases, and individual protocol actions. These configurations rely on Configuration Pairs. A configuration pair strictly couples a validation strategy with an external-system mock. Test suites can include multiple pairs at once. This allows for complex cross-channel communication. The report comprehensively monitors and displays this communication. It logs every message. The interface allows users to inspect messages directly. When a test fails, the report clearly highlights the error, which helps developers analyze the issue.

6. Integration of the Integration Test Engine to the CI/CD

The test framework functions as an automated quality gate embedded within CI/CD pipelines. The framework automatically discovers and exercises gateway configurations, eliminating the need to alter test logic when adding modules. To embed this into the Logimic infrastructure, a GitHub Actions[4] workflow structures the pipeline. The automated tests execute exclusively on a self-hosted runner. This self-hosted environment is critical because integration tests must communicate with genuine daemon processes across functional network topologies. The physical machine provides direct access to edge gateway hardware, Modbus TCP servers, and physical serial ports. This ensures tests validate accurate, real-world interactions. Following execution, the pipeline uploads the interactive HTML report, structured logs, and additional Mochawesome[5] output to support persistent analysis capabilities.

[1] TRB140 website: https://www.teltonika-networks.com
[2] MQTT website: https://mqtt.org
[3] Modbus website: https://www.modbus.org
[4] GitHub Actions website: https://github.com/features/actions
[5] Mochawesome website: https://www.npmjs.com

Acknowledgements

I would like to thank my supervisor Ing. Jiří Hynek Ph.D and consultant Ing. Ondřej Šulc for their support.

References

[1] Earl Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41:1–1, January 2014.