JÖNKÖPING UNIVERSITY
School of Engineering

Applying and Evaluating Automated Testing for Triple-Ended Systems

Edvin Mujanovic, Edin Nasic

May 30, 2025

Main subject area: Computer Engineering
School: School of Engineering

This final thesis has been carried out at the School of Engineering at Jönköping University within Computer Engineering. The authors are responsible for the presented opinions, conclusions and results.

Examiner: Johannes Schmidt

Supervisor: Jasmin Jakupovic

Scope: 15 hp (first-cycle education)

Date: May 30, 2025

Abstract

The increasing complexity of Internet of Things ecosystems, especially in triple-ended systems (systems involving a backend, frontend, and a third end such as Bluetooth), calls for efficient and scalable automated testing approaches. This thesis investigates the application of the Patrol testing framework for Flutter cross platform mobile applications to automate functional and integration tests in such systems, with a focus on Bluetooth-driven Firmware Over-the-Air updates. The research evaluates Patrol's effectiveness in automating regression tests, reducing manual effort, and enhancing test reliability across interconnected components. Key findings include the successful automation of 70% of Thule's regression test suite, significant time savings, and improved validation consistency. The study also provides practical insights into applying test automation in real-world industrial environments and discusses limitations related to hardware interactions and external dependencies. The results contribute to more efficient testing practices for cross-platform and IoT applications, offering a foundation for further research in automated testing of interconnected systems.

Acknowledgements

We would like to express our sincere gratitude to Thule Sweden AB for their collaboration and support throughout this thesis work. Their commitment to innovation and quality has provided us with an invaluable opportunity to contribute to the field of automated testing for interconnected systems.

A special thanks to Magnus Myrstedt, whose efforts made this project possible, and to Joel Johansson, whose insights and ideas have helped shape our approach. We are also deeply grateful to Srinivasan Raman for his technical guidance and assistance during the development process.

Finally, we would like to extend our appreciation to our supervisor at Jönköping University, Jasmin Jakupovic, for his continuous support, constructive feedback, and encouragement, ensuring that we maintain a high standard in our work.

Contents

1 Background and Problem Description **1**
1.1 Overview of Software Testing 1
1.2 Evolution of Software Testing 1
1.3 Automated Testing 1
1.4 Challenges in Software Testing 2
1.5 Emerging Trends and Future Directions 3
1.6 Industry Context: Collaboration with Thule 3

2 Purpose, Research Questions **4**
2.1 Purpose 4
2.2 Research Questions 4
2.3 Delimitations 4
2.4 Expected Findings 5
2.5 Relevance to the Field 5

3 Research Design and Methodology **6**
3.1 Research Design 6
3.2 Document Analysis 6
3.3 Experimental Setup 6
3.4 Analysis Methods 7

4 Theoretical Framework **8**
4.1 Testing in Software Development 8
4.2 Automated Testing in Triple-Ended Systems 8
4.3 Automated Testing Frameworks 9
4.3.1 The Patrol Testing Framework 9
4.4 Bluetooth-Driven Firmware Updates 10
4.5 CI/CD Pipelines in Software Development 11

5 Results and Discussion **12**
5.1 Implementation of Automated Testing with Patrol 12
5.2 Test Automation Coverage Analysis 13
5.2.1 Automated Test Breakdown 13
5.3 Time Efficiency and Execution Comparison 15
5.4 Test Complexity and Maintainability 16
5.5 Platform Performance and Stability 17
5.6 Patrol Framework Evaluation 18
5.7 Improved Quality Assurance 19
5.8 Answering the Research Questions 19

6 Conclusion **20**

7 References **22**

1 Background and Problem Description

1.1 Overview of Software Testing

In software development, testing has evolved from a final checkpoint to a continuous, integral process. As systems grow in complexity, with layers of interactions and dependencies, the scope of testing must expand to address quality, security, and resilience at every level. Testing is a critical activity to ensure quality assurance and involves designing test cases, running them against the software, and analyzing the results to discover defects and ensure alignment with the requirements (Harrold, 2000).

By estimating potential error rates early on, as Ottenstein's model suggests, testing teams can focus their efforts where bugs are most likely to occur, ensuring that resources are used efficiently throughout the validation phase (Ottenstein, 1979). This proactive approach not only enhances software reliability but also supports a more strategic allocation of testing resources, enabling teams to address areas of high complexity and risk with targeted testing efforts (Ottenstein, 1979).

Boehm and Papaccio (1988) further highlight that the costs associated with testing escalate when errors are detected later in the development process, making early testing not only a quality measure but also a cost-saving strategy. Detecting defects sooner allows teams to avoid costly rework, reduce project delays, and better manage overall development budgets (Boehm, 1988).

1.2 Evolution of Software Testing

Testing has undergone significant changes over the decades. Initially, it was a manual process focused on final quality checks, but modern software development practices now integrate testing throughout the development lifecycle. Continuous Integration and Continuous Delivery (CI/CD) pipelines have made testing a continuous process, ensuring every change is validated (Harrold, 2000). Practices like Test-Driven Development (TDD) emphasize the writing of tests before code, fostering more reliable and maintainable software. This evolution reflects the industry's growing reliance on automation and agile methodologies to meet the demands of modern development cycles.

1.3 Automated Testing

Automation has become essential to handle the scale and speed of modern software development. Automated testing helps achieve more consistent test coverage, supports CI/CD workflows, and enables rapid regression testing - the repeated execution of test cases to ensure that recent code changes have not adversely affected existing functionality (Rani, 2024). In agile environments, where speed and flexibility are crucial, automated testing supports CI/CD workflows by enabling rapid and reliable software delivery (Garg, 2024). However, setting up and maintaining an automated test suite requires expertise and ongoing resources, particularly for complex systems like triple-ended environments.

1.4 Challenges in Software Testing

One of the unique challenges in connected product ecosystems is testing **triple-ended systems**, which consist of three interconnected components: the app frontend, the backend, and the Bluetooth connection to physical devices. These components must integrate seamlessly, as failures in any one of them can disrupt overall functionality.

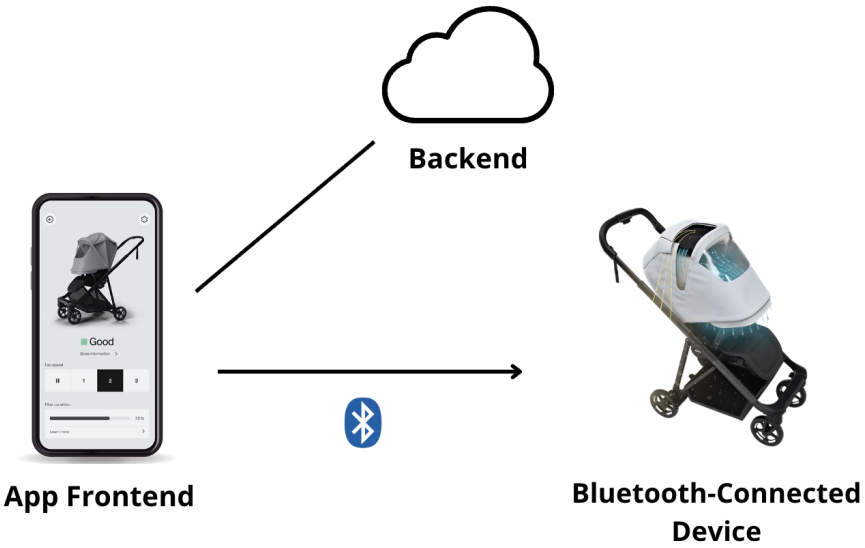

Figure 1: *Triple-ended system showing the app frontend (mobile phone), backend (cloud), and Bluetooth-connected device. The app communicates with the backend via the internet and connects to the IoT device over Bluetooth for firmware updates.*

Testing these systems involves resolving key challenges, including real-time data synchronization, hardware-software integration, and platform-specific behaviours. Flutter, a cross-platform framework used in this project, complicates testing due to its platform-specific functionality and compatibility issues. Cross-platform development tools often face challenges in ensuring consistent behavior between platforms, which adds to the complexity of testing and debugging in frameworks like Flutter (Jamali, 2022).

Traditional end-to-end testing often neglects the Bluetooth component, focusing more on app-to-backend interactions (Baker, 2020). This oversight can create significant inefficiencies during Firmware-Over-The-Air (FOTA) updates. Effective FOTA testing ensures seamless updates, prevents system failures, and maintains reliability across devices (Thomas, 2022). However, current research and tools inadequately address the unique challenges of testing firmware updates within triple-ended systems, particularly when integrating app, backend, and Bluetooth functionalities.

This thesis aims to address these challenges by evaluating the applicability of automated testing in triple-ended systems, focusing on improving the validation

of Bluetooth-driven firmware updates. By leveraging automation, the goal is to enhance failure detection and diagnosis, with future potential for integrating these processes into CI/CD pipelines. This research contributes to a deeper understanding of how automation can improve test efficiency, reliability, and scalability in environments where seamless updates and consistent performance are critical.

Currently, testing frameworks that support triple-ended systems remain limited, with most solutions primarily addressing app or backend validation while neglecting Bluetooth-driven interactions. This thesis explores how existing automation tools can be adapted to bridge this gap, reducing reliance on manual testing, improving test coverage, and ultimately enhancing the reliability of connected systems.

1.5 Emerging Trends and Future Directions

As technology advances, so do the methods and tools for software testing. Emerging trends include AI-driven testing, advanced test automation frameworks, and the integration of security testing into the CI/CD pipeline (DevSecOps). These advancements promise to make testing more effective, efficient, and proactive in identifying vulnerabilities. Additionally, as IoT and connected devices become more common, testing frameworks must evolve to accommodate the unique challenges posed by real-time synchronization, hardware variability, and cross-platform compatibility.

1.6 Industry Context: Collaboration with Thule

This thesis is conducted in collaboration with Thule Sweden AB, a global leader in solutions that simplify active lifestyles (Thule Group, nd). Thule provides access to their technologies, hardware, and software, including their cross-platform app, which serves as a key tool for implementing and evaluating automated testing for Bluetooth-driven firmware updates in triple-ended systems.

Thule has also supplied essential hardware, such as iPhones, Android devices, and connected products, enabling comprehensive testing across their ecosystem. Additionally, access to their backend architecture and testing environments ensures efficient execution and refinement of the project. This collaboration supports Thule's goals of improving product reliability and testing efficiency while aligning with the academic objectives of this thesis.

While Thule's existing CI/CD pipeline supports app and backend testing, its integration with Bluetooth testing remains incomplete. Addressing this gap will not only improve testing coverage but also reduce deployment risks and enhance the reliability of product updates.

2 Purpose, Research Questions

2.1 Purpose

The increasing reliance on IoT devices and the complexity of their ecosystems require testing solutions that can efficiently validate interactions across multiple system components, ensuring product reliability and user satisfaction. This thesis aims to evaluate methodologies for applying automated testing frameworks to triple-ended systems. The focus will be on integrating Bluetooth-driven firmware updates, improving failure detection and diagnosis, and outlining how such tests could be integrated into CI/CD pipelines in future workflows. By achieving this, the project aims to contribute to more reliable and efficient testing processes in cross-platform test environments, reducing the need for manual intervention.

The guidelines developed in this thesis have the potential to be adapted for other IoT applications, including different types of connected systems. Ultimately, the purpose of the project is to support higher product quality and reduced development cycles. This lays the foundation for investigating specific research questions related to the topic.

2.2 Research Questions

The thesis will address the following research questions:

RQ1: How can testing frameworks, such as Patrol, be utilized to support Bluetooth-driven firmware updates in triple-ended systems?

RQ2: How can testing frameworks, such as Patrol, be improved for triple-ended testing processes?

RQ3: To what extent can validation in triple-ended systems be accomplished?

RQ4: What are the measurable benefits of automation in testing of triple-ended systems?

2.3 Delimitations

This thesis is limited to Flutter applications, using Patrol as the sole automation tool. Patrol is a Dart-specific framework chosen for its compatibility with the target development environment and its ability to support end-to-end tests in cross-platform scenarios. Applications built with other frameworks were excluded from the scope of this research.

The backend, while essential in Thule's architecture, was not tested directly. Instead, backend functionality was indirectly validated through frontend UI-based flows using Patrol, as the framework operates primarily at the app layer.

Bluetooth testing included both Firmware Over-the-Air (FOTA) updates and other BLE-based interactions such as product registration and settings changes. However, advanced scenarios like partial updates, multi-device synchronization, and performance-related testing (e.g., signal stability or latency) were excluded.

Testing was primarily conducted on iOS using Apple hardware, including a Mac mini (2023, M2, 8GB RAM) and an iPhone 13 running iOS 18.3.1, to

ensure consistent and reproducible timing results. While some tests were also executed on Android, only iOS results were evaluated to maintain a controlled environment.

Other exclusions include performance testing (e.g., stress or capacity testing), security testing, and penetration testing. Additionally, certain product flows—such as QR code scanning or initial Bluetooth pairing—still required manual interaction, limiting automation coverage in those areas.

2.4 Expected Findings

This thesis is expected to provide insights into enhancing automated testing for triple-ended systems, particularly focusing on Bluetooth-driven firmware updates and the effectiveness of Patrol in automating regression tests. The findings will explore methodologies for automating Firmware Over-the-Air (FOTA) processes, assessing Patrol's ability to improve coverage and speed while reducing manual testing efforts. Additionally, the research will analyze the maintainability of these testing workflows and their long-term impact on product quality.

2.5 Relevance to the Field

For Thule, this research directly supports the advancement of testing workflows for connected products. By automating testing across the app, backend, and Bluetooth systems, seamless integration, faster deployment cycles, and higher product quality will be ensured. The findings of this thesis contribute to the broader IoT and software engineering domains, establishing workflows for automating tests in triple-ended systems, applicable across various industries.

This study addresses a significant research gap concerning automated testing frameworks for triple-ended cross-platform systems. Expected academic contributions include advancing the theoretical understanding of automated testing methodologies for interconnected systems and providing a foundation for future research in advanced Bluetooth testing.

3 Research Design and Methodology

3.1 Research Design

This research followed an exploratory and empirical design to address the complexities of testing in triple-ended systems. The study began with a thorough review of existing documentation and tools at Thule, with a focus on the current implementation of Patrol, a testing framework specifically designed for Flutter applications. Building upon this foundation, the framework was utilized to encompass triple-ended systems by incorporating Bluetooth testing and optimizing it for Firmware Over-the-Air (FOTA) scenarios.

To ensure a systematic approach, an iterative process was employed, consisting of planning, implementation, testing, and refinement cycles. This iterative design aimed to validate the efficiency and scalability of the proposed outcome while addressing any challenges encountered during development.

3.2 Document Analysis

The study incorporated document analysis as a tool to review existing resources. According to Bowen (2009), document analysis offers a systematic way to evaluate documents, helping researchers uncover patterns, identify gaps, and gain meaningful insights. This approach was instrumental in analyzing the current implementation of Patrol and guiding the creation of a more optimized testing framework.

3.3 Experimental Setup

A controlled environment was created to configure Patrol for app, backend, and Bluetooth interactions. The experiments involved iterative testing cycles to refine the framework and validate its capabilities.

A key aspect of the experimental setup was simulating real user interactions by automating Thule's existing regression tests. These tests replicate end-user workflows across the triple-ended system. The automation was planned to ensure consistency and reliability by covering critical use cases such as user and product management, Bluetooth pairing, and firmware update processes.

To achieve this, the following steps were undertaken:

- **Environment Configuration**: A dedicated testing environment was established, including physical devices, Bluetooth modules, and backend systems connected to the Patrol framework.
- **Regression Test Selection**: Existing manual regression test cases at Thule were analyzed and converted into automated test scripts.
- **Test Execution and Refinement**: Automated tests were executed iteratively to identify inefficiencies and refine workflows, ensuring robustness and accuracy.
- **Data Collection**: Test execution results, including logs, error reports, and execution times, were recorded for analysis.

Furthermore, by automating the current regression testing suite, the research provided insights into how well the framework can mimic real-world usage patterns, detect edge cases, and support long-term validation strategies. This approach helped measure the efficiency of automated workflows in reducing manual intervention while maintaining high test coverage across all system components.

All activities outlined in this setup, including environment configuration, test automation, and iterative execution, were performed directly by the authors.

3.4 Analysis Methods

The performance and effectiveness of the testing framework were evaluated using the following metrics:

- **Test Case Coverage:** The framework was evaluated based on how many of the defined regression test cases can be automated using Patrol. Each category of functionality was analyzed to understand automation feasibility and reasons for manual fallback.
- **Time Efficiency:** Manual versus automated test execution times were measured to quantify time savings. Results were gathered across key workflows and averaged based on repeated test runs to reflect realistic usage.
- **Implementation Complexity:** Metrics such as average lines of code per test, time required to write individual test cases, and maintenance effort were collected to evaluate the effort needed to adopt Patrol in a production setting.
- **Platform Performance:** Patrol's behavior on emulators and physical devices were compared in terms of setup complexity, execution speed, and test stability. This is particularly relevant for hardware-dependent flows like Bluetooth interactions.
- **Framework Capabilities:** Patrol's feature support were reviewed across core test requirements, and limitations in handling external systems.

By combining insights from previous work with hands-on implementation and systematic evaluation, this methodology ensures a comprehensive analysis of Patrol's suitability for automated testing in Thule's triple-ended system, aligned with both industrial needs and the research objectives of this thesis.

4 Theoretical Framework

4.1 Testing in Software Development

Testing plays a foundational role in the software development lifecycle, ensuring the quality and reliability of a software product (Honest, 2019). It helps identify defects, confirm that the system meets requirements, and assess whether it functions correctly under different conditions. Through validation and verification, testing ensures that the software performs as expected and meets user needs. Effective testing reduces the risk of failure, lowers maintenance costs, and contributes to the long-term success of the software (Honest, 2019).

4.2 Automated Testing in Triple-Ended Systems

Automated testing is a critical component of modern software development, particularly in agile methodologies and CI/CD pipelines, where rapid iterations demand efficient and reliable validation processes (Rani, 2024). Based on formal methods, it can derive test cases directly from system specifications, enabling the automation of test execution, which enhances the testing process's efficiency and coverage (Tretmans, 1999). Test automation tools are often designed for specific application types, requiring tailored configurations to accommodate different user interfaces and interaction methods. However, in complex systems integrating multiple application types across various platforms, automated testing must account for cross-platform interactions and dependencies (Akat and Sözer, 2023).

Testing in triple-ended systems, encompassing app frontends, backends, and Bluetooth connections, introduces additional complexity. Effective end-to-end testing in Internet of Things (IoT) ecosystems, as highlighted by Maheta (2019), requires validation at multiple stages, ensuring seamless integration of diverse components like mobile applications, cloud infrastructure, and embedded devices. Traditional testing frameworks often focus on either frontend or backend testing, neglecting the interactions with physical hardware. This limitation underscores the importance of frameworks that can handle real-time synchronization, hardware-software integration, and platform-specific behaviors. Bluetooth functionalities, particularly Firmware Over-the-Air (FOTA) updates, remain an underexplored area in automated testing and are a core focus of this research.

Flutter's cross-platform capabilities introduce challenges like platform-specific bugs, varying device APIs, and inconsistencies in hardware behavior. Debugging such issues is often complicated, especially when testing solely on emulators. For instance, Flutter's "hot reload" feature, while useful, occasionally requires full app restarts to resolve certain bugs. Combining real device testing with tools like Patrol helps identify these platform-specific issues effectively (Peterson, 2024).

Triple-ended systems require fault-tolerant testing to handle dropped connections or incomplete transmissions in Bluetooth-driven processes. For example, a temporary connection loss can disrupt data flow between a Bluetooth device and its backend. Test scenarios must simulate these failures to ensure the system can recover seamlessly. Similarly, automated tests should evaluate synchronization challenges, such as latency between the app, backend, and Bluetooth device, ensuring a smooth user experience even under real-world conditions.

4.3 Automated Testing Frameworks

Automated testing frameworks address the inefficiencies of manual testing by minimizing repetitive human effort, increasing test coverage, and improving cost-effectiveness (Jian-Ping et al., 2012) By leveraging keyword-driven approaches and automation tools, these frameworks enable scalable and maintainable testing processes, especially in complex software systems. Jian-Ping et al. further highlight how automated testing frameworks not only reduce manual labor but also enhance efficiency through structured and tool-based test execution strategies.

Tools like Patrol and Appium have been evaluated for their ability to streamline testing workflows, particularly in cross-platform app development (Ghalioun, 2024). Patrol, a Dart-native testing framework, was found to be better suited for Flutter applications compared to Appium, primarily because of its integration within the Flutter ecosystem and its ability to leverage Dart for end-to-end testing (Ghalioun, 2024).

4.3.1 The Patrol Testing Framework

Patrol is an open-source UI testing framework developed by LeanCode, designed to enhance Flutter's built-in testing functionality. It enables native platform interactions directly from Dart code, allowing more realistic test scenarios across both Android and iOS devices (LeanCode, 2024b). Supported interactions include system permission dialogs, toggling Wi-Fi or Bluetooth, handling WebViews, tapping on notifications, and verifying native popups. These capabilities are essential for replicating real-world app behavior. However, some platform-specific limitations remain—for instance, iOS does not support simulating the back button, which can affect certain navigation flows (LeanCode, 2024a). Such constraints should be considered when designing robust cross-platform test strategies.

One of Patrol's most valuable features is its use of custom finders, which enable precise targeting of both Flutter widgets and native UI components during test execution. This makes it possible to automate user flows that involve cross-layer behavior, such as accepting a permission prompt or navigating through platform-specific settings. These flows are traditionally difficult or impossible to cover using default Flutter test tools (LeanCode, 2024c).

In addition, Patrol integrates seamlessly with the `flutter_test` and `integration_test` packages, supporting test reusability and better alignment with existing CI/CD pipelines. The framework also supports hot restarts during test development, enabling rapid test iteration without full rebuilds—significantly improving developer productivity and feedback cycles.

Patrol enhances Flutter's testing by bridging Flutter's widget tests with native automation. By leveraging Dart's integration, Patrol simplifies writing comprehensive tests, improving test coverage and reliability in triple-ended systems (Sorathiya, 2024).

Patrol's alignment with Flutter's cross-platform testing goals makes it particularly suitable for applications that require end-to-end validation across multiple layers. These include systems that rely on seamless interaction between mobile interfaces,

backend services, and Bluetooth-connected hardware. By supporting both Flutter and native UI layers, Patrol fills a critical gap in the testing stack—enabling comprehensive automation that extends beyond what traditional Flutter tests can cover. Its integration with CI/CD workflows further enhances its value, allowing teams to maintain test reliability, scalability, and long-term maintainability within modern development pipelines.

4.4 Bluetooth-Driven Firmware Updates

Firmware updates over Bluetooth require robust testing frameworks to ensure that updates are applied consistently and securely across devices. Challenges include handling interruptions during updates, ensuring compatibility across platforms (e.g., iOS and Android), and maintaining data integrity throughout the process. Previous research has identified the lack of comprehensive Bluetooth testing capabilities in many existing frameworks as a critical gap, making this an essential focus for this thesis.

One key aspect of FOTA testing is ensuring that rollback functionality works as expected. If an update fails, whether due to incomplete data transfer or an incompatible firmware version, the device should automatically revert to its previous stable version without losing functionality. To test this, automated scenarios can simulate issues like partial or corrupted update files, checking how well the system detects and recovers from such failures. Additionally, compatibility testing for different Bluetooth protocols, such as BLE 4.2 and BLE 5.0, can help identify any problems that might occur during firmware updates across various devices.

Regression testing plays a pivotal role in identifying compatibility issues, hidden defects, and unintended side effects during software upgrades. By automating repetitive test cases, modern testing frameworks help improve test coverage and reliability, both of which are essential for ensuring a seamless user experience during firmware deployment (Gopalakrishnan, 2024).

While many published frameworks have focused on safety-critical sectors such as the automotive industry, the underlying principles of robust firmware validation are equally applicable to consumer electronics and IoT ecosystems. For instance, the National Highway Traffic Safety Administration (NHTSA) emphasizes the importance of thorough firmware testing to mitigate risks associated with firmware updates, ensuring updates preserve system integrity without introducing new vulnerabilities (National Highway Traffic Safety Administration (NHTSA), 2021).

Similarly, in the embedded systems domain, Falas et al. (2020) present a secure firmware update framework that incorporates formal verification and structured validation techniques. Their work further highlights the importance of thorough testing throughout the update process.

By incorporating insights from both industrial safety standards and academic research, the need for robust firmware testing becomes evident. These perspectives reinforce the relevance of automated testing in Bluetooth-connected consumer products like those developed by Thule, supporting system reliability, data security, and long-term maintainability

4.5 CI/CD Pipelines in Software Development

CI/CD pipelines are foundational in modern software engineering, enabling teams to automate the integration and deployment of code changes efficiently (Garg, 2024). Continuous Integration and Delivery (CI/CD) practices offer numerous advantages in software development, including accelerated release cycles and the earlier detection of defects (Zampetti et al., 2023). Integrating automated testing into these pipelines is critical for validating all aspects of a triple-ended system.

In the context of embedded devices, a study by Bernhardt (2021) explores the implementation of a continuous integration pipeline that automatically tests the update channel between Android phones and embedded devices over Bluetooth. This approach not only validates updates in every commit but also automates testing, ensuring a smooth upgrade process even with the limited connection channels typically available in embedded systems. The study demonstrates that automating the testing of Bluetooth firmware updates reduces human error and enhances scalability during the testing process.

Incorporating physical Bluetooth devices into CI/CD workflows involves automating tasks such as resetting, pairing, and connecting devices during testing. Custom scripts can manage Bluetooth pairing and data transfers, validating their outcomes to ensure proper functionality. Emulation environments can also simulate backend responses, allowing for efficient testing alongside physical devices.

5 Results and Discussion

This section presents the implementation and results of automated testing using Patrol within Thule's mobile application. The goal was to automate as much of the existing regression test suite as deemed feasible, minimizing manual testing efforts, validating firmware updates, and improving overall test coverage. The results analyze the extent of test automation achieved, highlight challenges and limitations, and suggest potential improvements.

5.1 Implementation of Automated Testing with Patrol

Automated testing was successfully implemented to validate key functionalities in the Thule app, with a focus on comprehensive coverage of tests that include validation of user authentication, product registration, settings management, and device interactions.

The following major areas were fully automated using Patrol:

- **User Authentication & Account Management:**
  - Account creation with valid and invalid inputs (e.g., missing country, unmatched passwords, short passwords)
  - Handling of existing accounts and proper error message display
  - Login with correct and incorrect credentials, including invalid email/password/country combinations
  - Validation of missing fields and incorrect formatting in the login and password reset flows
  - Change password workflow, including validation of current password, enforcement of complexity rules, error handling, and successful password update
  - Logout functionality
- **Product Registration & Management:**
  - Manual registration of a new product using a serial number (valid and invalid)
  - Handling of duplicate product registrations
  - Selecting product type to add
  - Product deletion via settings
- **App Navigation & General UI:**
  - Visibility of Terms of Use, Privacy Policy, and login/create account buttons
  - Navigation through the main app menu: My Products, About, Support, and Account Settings
  - App version visibility and user session validation
- **Settings and Configurations:**

  - Accessing and viewing firmware versions, MAC address, serial number, and hardware version
  - Managing location services, data sharing preferences, and notification toggles (within app scope)
  - Updating product name
  - Viewing product manual
- **Home Screen & Device Interaction:**
  - Air quality info and AQI value displays
  - Filter status viewing and resetting
  - Fan control from the app interface (0–3 levels)
  - Verification of visual components like the Learn More overlays
- **Firmware Updates:** Patrol successfully automated firmware updates, verifying that devices could detect, download, and apply updates without manual intervention. This implementation demonstrates that Patrol can automate Bluetooth-driven firmware update flows in a real-world triple-ended system.

Testing on Products in Active Development

In parallel with automating tests for Thule's current connected product available on the market, Patrol was also used to support testing of new products that are still under active development. These unreleased systems differ significantly in both functionality and technical architecture and are undergoing continuous changes across firmware and app layers.

Despite the evolving nature of these products, Patrol was successfully applied to automate several key flows, including product addition, product settings, and Bluetooth-based interactions. This demonstrates that Patrol is not only suitable for regression testing of finalized systems but also effective in supporting development workflows, where early detection of UI issues and regressions is critical.

While high-level results were consistent with those seen in the released product, all detailed measurements and comparisons presented in the results section refer exclusively to the completed and commercially available product, in order to maintain consistency and data integrity.

5.2 Test Automation Coverage Analysis

To quantify the success of automation, we categorized test cases based on their automation feasibility.

5.2.1 Automated Test Breakdown

The table below summarizes the test categories, number of test cases automated with Patrol, and reasons for non-automated cases.

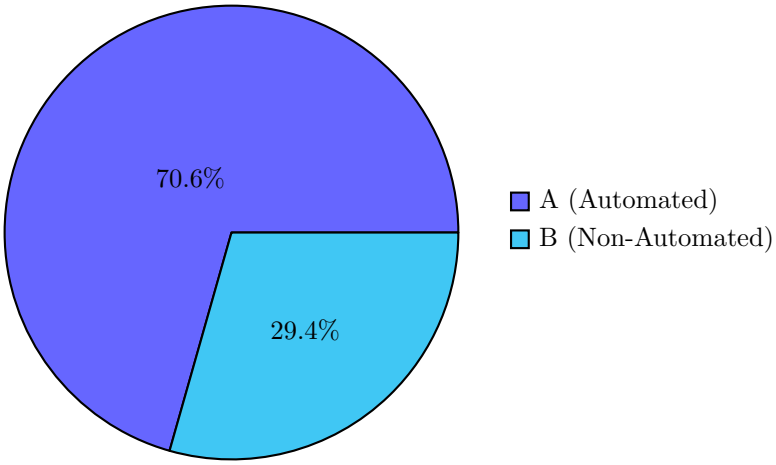

Figure 2: *Distribution of automated vs. non-automated test cases.*

Test Category	**Total**	**Patrol**	**Manual**	**Reason for Manual Testing**	**Rate**
UI Verification	1	1	0	-	100%
Create Account	5	5	0	-	100%
Login	6	6	0	-	100%
Forgot Password	4	1	3	Email verification required	25%
Change Password	7	7	0	-	100%
App Menu	6	6	0	-	100%
Product Selection	2	2	0	-	100%
Product Registration	7	4	3	Requires manual QR code scanning	57%
Connect Product	10	0	10	Requires manual interaction with product	0%
Product Interaction	6	5	1	Requires manual interaction with product	83%
Notifications	2	0	2	Requires activity outside app (phone settings)	0%
Product Settings	7	6	1	Factory reset is not efficient to automate	86%
Version Settings	4	4	0	-	100%
Delete Product	1	1	0	-	100%
Total	**68**	**48**	**20**	–	**70.6%**

Table 1: *Automated Patrol vs. Manual Test Breakdown*

Overall, 70.6% of the regression test cases were automated, with the remaining cases requiring manual testing due to hardware limitations, system-level interactions, or external service dependencies. This coverage significantly reduced testing overhead and ensured that all critical workflows were validated consistently.

5.3 Time Efficiency and Execution Comparison

To evaluate time savings, each automated test flow was measured and compared against manually executed counterparts. Manual execution times were recorded across multiple test runs by four members of the Thule app development team, authors of this paper included. The results were averaged to provide a realistic baseline based on actual workflows. Since regression testing is typically performed by individuals already familiar with the app's functionality, no time measurements were taken from users outside the development team, as they would not be representative of real-world test cycles.

All Patrol tests were executed on a Mac mini (2023) with an Apple M2 chip and 8 GB of memory. Physical test runs were performed on an iPhone 13 running iOS 18.3.1. All time measurements are based on this environment to ensure consistency and reproducibility.

Setup time was excluded from both approaches, as it varies by device and context (typically 1–2 minutes). In practice, multiple tests can be batched together to reduce overhead.

Since many key tests (such as Bluetooth interactions and firmware validation) require real hardware, the execution time comparison in this section focuses solely on physical device performance. A separate section (6.6) compares emulators and physical devices in terms of stability and usability.

One of the major benefits of automation is that while Patrol tests are running, the tester is free to focus on other tasks. In contrast, manual testing requires full attention, meaning no other productive work can be performed simultaneously. This indirect time savings is a crucial advantage of automation, as it enables better resource utilization.

Additionally, the manual execution times shown below represent the time taken per test case when executed in isolation. Timing values shown have been rounded to the nearest full or half minute to maintain consistency and readability in presentation. Raw execution data was used in the underlying analysis. Patrol test durations include any added await and delays required to ensure stable execution across different devices.

However, in real-world scenarios, executing multiple manual tests back-to-back can be mentally and physically demanding. Testers typically require breaks between sessions, which further increases the actual time required to complete a full regression cycle. These breaks have not been accounted for in the overall measurements and the total time presented in the table represents a summary of individually executed manual test. Automation mitigates this issue by eliminating fatigue-related slowdowns, ensuring consistent execution speed regardless of test volume.

Flow	**Manual Time (min)**	**Patrol Time (min)**	**Time Saved (min)**	**% Faster**
Create Account + UI	5	0.5	4.5	90%
Login etc.	6	1.0	5.0	83%
Change Password	7	1.5	5.5	79%
App Menu	3	0.5	2.5	83%
Product Registration	4	1.0	3.0	75%
Product Interaction + FOTA	17	3.0	14.0	82%
Total	**42**	**7.5**	**34.5**	**82%**

Table 2: *Manual vs. Automated Test Execution Time*

This comparison highlights not only the direct time savings but also the broader efficiency and scalability benefits of automation. Patrol eliminates the need for continuous tester involvement, allows concurrent work, and ensures consistent execution times, all of which contribute to a more streamlined development and quality assurance process.

5.4 Test Complexity and Maintainability

To evaluate the practical effort required for automated testing with Patrol, the average code size, time investment, and maintenance burden were measured across individual test cases. The table below presents representative values based on the implementation conducted during this thesis.

These values represent individual low-level test cases (e.g., "login with invalid credentials"), not grouped scenarios. All data is based on hands-on implementation by the authors during the thesis work.

Metric	Value	Notes
Avg. lines of code (LOC) per test case	∼8 LOC	Most small test cases ranged from 4 to 15 lines. Code remained concise and readable.
Avg. time to write test	∼15–25 min	Includes identifying UI elements, adding keys in the source code, and writing the test logic.
Maintenance required	Low	Tests remained stable across app updates, with minimal need for refactoring.

Table 3: *Test Complexity and Maintainability*

5.5 Platform Performance and Stability

Automated tests were executed on both emulators and physical iOS devices to evaluate platform differences. While both environments were suitable for testing, they served different purposes during development and execution.

Physical devices were essential for testing flows that required real Bluetooth interaction, such as product connection and firmware updates. In contrast, emulators provided a faster and more flexible setup for developing and iterating on test cases, particularly for UI-based scenarios.

The comparison in Table 4 outlines three key aspects—*setup complexity*, *test execution speed*, and *stability*—rated as **Low**, **Medium**, or **High** to indicate relative performance:

- **Setup**: Refers to the time and effort needed to configure the environment.
  - **Low**: Minimal configuration required.
  - **Medium**: Moderate setup steps, such as configuring release builds or pairing devices.
- **Speed**: Describes how fast tests can be written, modified, and executed.
  - **Fast**: Ideal for rapid test cycles and development feedback.
- **Stability**: Indicates the consistency and reliability of test results over repeated runs.
  - **High**: Test outcomes are consistent and robust against variability or flakiness.

Environment	Setup	Speed	Stability	Notes
Physical Device	Medium	Fast	High	Required app to be built in release mode; no hot reload; supported real Bluetooth flows
Emulator	Low	Fast	Medium	Supports **patrol develop** with hot reload; no Bluetooth support; good for UI logic

Table 4: *Platform Comparison: Emulator vs Physical Device*

Test reliability here refers to how stable and consistent the test outcomes are across repeated runs. Physical devices offered the highest reliability for real-world flows due to actual hardware behavior, while emulators occasionally experienced flakiness in flows involving timing-sensitive elements like animations or asynchronous UI rendering.

5.6 Patrol Framework Evaluation

To better understand Patrol's suitability for automated testing in a triple-ended architecture, this section evaluates the framework across key functional areas. The table below summarizes Patrol's current capabilities and limitations, based on both practical usage during the project and official documentation. These insights help highlight where Patrol excels and where future improvements could further expand its value.

Feature	Supported	Notes
UI Navigation	Yes ✓	Clean syntax and fast execution
Bluetooth Communication	Yes ✓ (indirect)	Works with real devices; relies on app's built-in BLE handling
In-App Permission Dialogs	Yes ✓	Can interact with native permission pop-ups shown inside the app
System Settings Access	No ✗	Cannot open external system settings (e.g., turning off Bluetooth or camera)
External Authentication (Email links)	No ✗	Cannot handle incoming email flows or simulate email interaction
Home Screen Widgets	No ✗	Currently unsupported by Patrol
Cross-Platform (iOS/Android)	Yes ✓	Tested and compatible on both platforms

Table 5: *Framework Feature Evaluation*

5.7 Improved Quality Assurance

Beyond the reduction in manual testing time, the introduction of Patrol-based automation brought notable improvements in test reliability and consistency.

- **Eliminated Human Error:** Manual testing is inherently prone to oversight, especially during repetitive UI or settings validation (Global App Testing, 2024). Patrol ensures that each test is executed precisely and consistently, identifying even subtle UI regressions or missing elements that might go unnoticed during manual verification.
- **Consistent Firmware Validation:** Firmware updates, previously validated manually, were successfully automated, ensuring every update followed the same validation procedure. This consistency greatly reduced the likelihood of faulty firmware reaching production due to incomplete testing.
- **Scalability of Test Coverage:** As Thule's app evolves with new features, automated tests can be easily adapted or extended to cover new functionality. This ensures sustainable quality assurance over time.

Automation allows larger systems to continue evolving and scaling without needing to increase manual testing at the same rate, saving both time and resources while reducing the risk of human error.

Additionally, Patrol helped enforce input integrity by automatically validating required fields, password logic, email formatting, and region selection. While formal security testing was beyond the scope of this work, these built-in checks contributed to overall app stability and reduced the chance of user-flow issues slipping through unnoticed. By integrating this validation into automated tests, Patrol further enhanced the reliability and depth of Thule's quality assurance efforts.

5.8 Answering the Research Questions

RQ1: Patrol enabled fully automated Bluetooth-based firmware updates, covering detection, download, and installation without manual steps. This validated its ability to handle real-world FOTA workflows in a triple-ended system, including unreleased and commercial products.

RQ2: While Patrol performed reliably for in-app testing, it lacks access to system settings, iOS widgets, external email handling, and external flows like QR scanning. Expanding support in these areas could improve its effectiveness for broader triple-ended validation.

RQ3: Automation covered approximately 70% of the test suite, spanning UI, backend-triggered flows, and Bluetooth. However, system setting toggles, and some physical product handling still required manual checks, defining clear limits to what can be automated.

RQ4: Automation reduced test time by over 80%, improved consistency, enabled multitasking, and reduces human error. These measurable benefits make Patrol well-suited for scalable QA in triple-ended systems.

6 Conclusion

This thesis has demonstrated the efficacy of integrating the Patrol testing framework into Thule's mobile application development process. By automating a substantial portion of the regression test suite, we achieved significant reductions in manual testing efforts while improving the consistency and reliability of test outcomes.

Extent of Automation: As previously noted, approximately 70% of the regression test cases were successfully automated, encompassing critical workflows such as user authentication, product registration, and Bluetooth-driven firmware updates. This significantly reduced the need for manual intervention.

Reduction in Regression Testing Time: Manual regression testing time decreased from 42 minutes to 7.5 minutes, resulting in a time efficiency improvement of over 80%. These savings allow developers and testers to focus on more strategic tasks rather than repetitive verification steps.

Improved Test Reliability: Patrol detected unexpected UI outcomes immediately and enforced consistent test execution across devices. This ensured more accurate validation of app behavior and firmware update flows, reducing the likelihood of faulty software reaching production.

Firmware Update Testing: Firmware updates over Bluetooth were fully automated using Patrol. This enabled consistent, repeatable validation of update flows that previously required manual verification, improving confidence in device update reliability.

Enhanced Quality Assurance: Beyond efficiency, test automation significantly reduced the risk of human error. Manual testers may overlook edge cases or subtle UI inconsistencies, while Patrol consistently identified unmet conditions and verified all expected outcomes.

Extent of Validation in Triple-Ended Systems: While some system-level and hardware-specific scenarios still require manual testing, this study demonstrates that a substantial portion of the triple-ended architecture—comprising the mobile app, backend services, and Bluetooth- connected physical products—can be reliably validated through automated tests using Patrol. Approximately 70% of Thule's full regression suite was automated, covering all critical app flows and Bluetooth-driven firmware update functionalities. Although low-level system settings, native OS dialogs, and certain physical device interactions must still be tested manually, the framework enables comprehensive validation of end-to-end functionality within a triple-ended environment, demonstrating the extent and measurable benefits of automation in practice.

Final Reflections: This study reinforces that while automation is essential for scalability and test reliability, it is not a complete replacement for manual testing. A well-balanced approach that combines automation with manual validation of edge cases, hardware interactions, and OS-level permissions will ensure sustainable and accurate testing coverage in complex mobile systems.

Further Research

Further research could focus on extending the current testing scope and overcoming existing limitations. One direction is full CI/CD integration using tools like GitHub Actions, enabling continuous regression testing and improved long-term stability. Expanding support for complex system interactions—such as unstable Bluetooth behavior, permission edge cases, and system-level toggles—would also enhance coverage. Additionally, simulating hardware responses and automating manual flows like QR code scanning could reduce reliance on physical devices. As the Patrol framework matures, evaluating its capabilities for features like widget testing or external authentication flows would further strengthen automation in triple-ended systems.

7 References

Akat, Ö. and Sözer, H. (2023). Automated testing of systems of systems. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 73–79. Springer Nature Switzerland, Cham.

Baker, K. (2020). Writing end-to-end tests for bluetooth applications is difficult. *Beam Benefits*. https://medium.com/beambenefits/writing-end-to-end-tests-for-bluetooth-applications-is-difficult-441dde7c93 Accessed: 2025-04-14.

Bernhardt, A. J. (2021). CI/CD pipeline from Android to embedded devices with end-to-end testing based on containers. (Master's thesis, KTH Royal Institute of Technology). DiVA portal https://www.diva-portal.org/smash/get/diva2:1618170/FULLTEXT01.pdf.

Boehm, B.W. & Papaccio, P. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477. https://doi.org/10.1109/32.6191.

Bowen, G. A. (2009). Document analysis as a qualitative research method. *Qual. Res. J.*, 9(2):27–40. https://doi.org/10.3316/qrj0902027.

Falas, G., Yang, F., and Cazorla, F. J. (2020). Towards secure firmware updates for embedded systems. *arXiv*. https://arxiv.org/abs/2007.09071.

Garg, R. (2024). How to integrate automation testing into your CI/CD pipeline? *Frugal Testing*. https://www.frugaltesting.com/blog/how-to-integrate-automation-testing-into-your-ci-cd-pipeline?

Ghalioun, T & Obidhonyi, M. A. (2024). Software test automation in a cross-development environment : Evaluating the effectiveness of patrol and appium. Bachelor's thesis, Jönköping University, School of Engineering. https://www.diva-portal.org/smash/get/diva2:1898838/FULLTEXT01.pdf.

Global App Testing (2024). Advantages and disadvantages of manual testing. https://www.globalapptesting.com/blog/advantages-of-manual-testing.

Gopalakrishnan, M. (2024). Software update testing challenges and test cases. Embien Technologies. Retrieved April 14, 2025 from https://www.embien.com/blog/software-update-testing-challenges-and-test-cases.

Harrold, M. J. (2000). Testing. In *Testing: A Roadmap*, pages 61–72. https://doi.org/10.1145/336512.336532.

Honest, N. (2019). Role of testing in software development life cycle. *International Journal of Computer Sciences and Engineering*, 7(5):886–889. https://doi.org/10.26438/ijcse/v7i5.886889.

Jamali, I. (2022). Identifying trade-offs associated with cross-platform mobile development tools. Master's thesis, University of Saskatchewan. https://harvest.usask.ca/bitstream/10388/13995/1/JAMALI-THESIS-2022.pdf.

Jian-Ping, L., Juan-Juan, L., and Dong-Long, W. (2012). Application analysis of automated testing framework based on robot. pages 194–197. https://doi.org/10.1109/icndc.2012.53.

LeanCode (2024a). Native feature's parity — patrol. https://patrol.leancode.co/native/feature-parity. Accessed: 2025-04-01.

LeanCode (2024b). Patrol – flutter ui testing framework. https://github.com/leancodepl/patrol. Accessed: 2024-03-31.

LeanCode (2024c). Write your first test. https://docs.page/leancodepl/patrol/write-your-first-test. Accessed: 2024-03-31.

Maheta, V. (2019). End-to-end testing for IoT ecosystem and importance of multistage validation. Embedded Computing Design. https://embeddedcomputing.com/technology/iot/gateways/end-to-end-testing-for-iot-eco-system-and-importance-of-multistage-validation.

National Highway Traffic Safety Administration (NHTSA) (2021). Cybersecurity best practices for the safety of modern vehicles. Technical report, U.S. Department of Transportation. https://www.nhtsa.gov/technology-innovation/vehicle-cybersecurity.

Ottenstein, L. M. (1979). Quantitative estimates of debugging requirements. *IEEE Transactions on Software Engineering*, SE-5(5):504–514. https://doi.org/10.1109/tse.1979.234211.

Peterson, M. (2024). What are the challenges in flutter app development? https://medium.com/@akash.chhabra/what-are-the-challenges-in-flutter-app-development-46a83706962a. Accessed: 2024-12-23.

Rani, S. (2024). Automated software testing: Tools, techniques, and best practices. *Journal of Software Engineering & Software Testing*, 9(2). https://admin.mantechpublications.com/index.php/JoSEST/issue/viewFile/7251/7850.

Sorathiya, N. (2024). Explore the power of Patrol Flutter. https://www.dhiwise.com/post/patrol-for-flutter-writing-better-tests-for-your-flutter-apps. Accessed: 2024-12-23.

Thomas, R. (2022). FOTA testing for IoT. https://blog.iotify.io/fota-testing-for-iot-48f9bbae10df. Accessed: 2024-12-23.

Thule Group (n.d.). Thule Group. https://www.thulegroup.com/en. Accessed: 2024-12-23.

Tretmans, Jan & Belinfante, A. (1999). Automatic testing with formal methods. Technical Report 99-17, Centre for Telematics and Information Technology (CTIT), Enschede.

Zampetti, Fiorella & Tamburri, D., Panichella, S., Panichella, A., Canfora, G., and Di Penta, M. (2023). Continuous integration and delivery practices for cyber-physical systems: An interview-based study. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–44. https://doi.org/10.1145/3571854.