pie

White Paper · 2025

Hardware-in-the-Loop Testing Automation

How QA Teams Cut Testing Cycles by 70%

The Stakes

Automating QA for hardware-connected applications is difficult, but not impossible. This whitepaper documents the five strategies that helped Fi, Lumo, and a Smart Mattress Company transform their testing. Not theory. Implementation.

Each company had established QA processes. Each was looking to accelerate releases without sacrificing coverage. The strategies we developed together are now repeatable across hardware-connected applications.

> "The time between having a release candidate ready and being fully tested has gone from two to three days to a few hours."
>
> **— Philip Hubert, Director of Mobile Engineering, Fi**

Why Hardware Apps Break Test Automation

Test automation works well for pure software. For hardware-connected applications, the success rate drops. The challenges are architectural.

Problem 1: Device State Lives Outside the Application

Hardware state exists in firmware and device memory, not in the app's UI layer. Connection status, sensor readings, and device modes, the app just displays all the information but doesn't control it.

Test frameworks verify what's visible in the UI. Device state often isn't represented in ways automation can reliably detect. Teams end up using arbitrary timeouts and indirect indicators. Tests pass sometimes, fail others.

Problem 2: Hardware Introduces Timing Variability

Software responds in milliseconds. Hardware doesn't.

A wireless connection might establish in 2 seconds or 20. A sensor reading might arrive immediately or after a retry cycle. Firmware responds differently based on battery level, signal strength, or environmental conditions.

Test frameworks expect consistent timing. Hardware delivers variability. The mismatch produces flaky tests that erode confidence in the entire suite.

Problem 3: Multi-Layer Complexity

Hardware apps span three distinct layers: firmware running on the device, backend APIs processing device data, and frontend applications displaying results.

When tests fail, root cause analysis means investigating all three. A UI rendering issue looks identical to a backend data error which looks identical to a firmware timing problem. Teams spend more time diagnosing failures than fixing them.

Problem 4: Continuous Update Cycles

Hardware apps don't just update their own code. They respond to firmware changes, backend updates, and device capability changes.

When firmware updates ship, UI behavior often shifts. Timing changes. Response formats change. Tests that passed yesterday fail today, not because anything broke but because assumptions changed.

Problem 5: Manual Testing Becomes Default

When automation keeps failing, teams retreat to manual testing. Manual testing works at low release velocity. Ship monthly, and it's manageable. Ship weekly, and QA becomes the bottleneck.

Manual testing becomes the constraint that determines how fast you can move.

How Pie Approaches Hardware Testing

Pie is an AI-native QA platform. Our platform tests applications by interpreting screens visually rather than through code selectors. This vision-based approach sees your app like a human would.

We built Pie to handle the kinds of applications where traditional automation struggles. Hardware-connected apps fall squarely into that category.

The five strategies in the next section came from working with hardware companies on Pie implementations. Each strategy addresses one or more of the problems outlined above.

Some strategies are Pie-specific. Others are architectural approaches that work regardless of tooling.

What they share: they've been implemented in production and delivered measurable results.

AI detecting - UI elements

Custom test creation

Device Preview

Discovery Steps & Assertion

Strategy 1: Layered Testing Architecture

The insight: Most hardware apps are mostly software.

Most hardware apps are mostly software.

A GPS pet collar app tracks location. But, it also handles authentication, pet profiles, activity history, notification settings, and payment processing. The hardware-dependent functionality typically represents 20-30% of the application. The remaining 70-80% is pure software that never touches a physical device.

The Strategy: Segment every user flow by hardware dependency, then automate systematically from least dependent to most dependent.

Tier	Definition	Examples	Automation Approach
Tier 1: No Hardware	Flows that never interact with device state	Login, profile management, settings, payment	Full automation, no special handling
Tier 2: Simulated Hardware	Flows that need device data but not live devices	Activity dashboards, historical charts, device status displays	Automation with API-based data seeding
Tier 3: Physical Hardware	Flows requiring actual device interaction	Bluetooth pairing, initial setup, physical button presses	Manual testing, tightly scoped

How Pie implements this:

When we onboard a hardware app, the first step is mapping every user flow to these tiers. Pie's discovery process explores the application and identifies which screens and interactions involve device state. The mapping reveals what's immediately automatable (Tier 1), what becomes automatable with API integration (Tier 2), and what stays manual (Tier 3).

Results from Fi: When we mapped Fi's mobile app, we found that 80%+ of user flows—authentication, pet profiles, activity dashboards, walk history, notification settings—required no physical collar.

Strategy 2: API-Driven Device Simulation

The insight: If the app needs device data to function, you don't need a device to provide that data.

If the app needs device data to function, you don't need a device to provide that data.

Hardware apps display information that originates from devices—GPS coordinates, sleep metrics, sensor readings, status updates. The app receives this data through APIs, processes it, and renders it in the UI.

The app doesn't know or care whether that data came from a physical device or was injected directly into the backend. It processes and displays whatever it receives.

The Strategy

Work with backend teams to enable test data injection, then seed test accounts with the device states and historical data needed for comprehensive testing.

Test Account Seeding: Create test accounts pre-configured with specific device states and data histories. Instead of connecting a device and generating data organically, the backend populates the account with exactly what the tests need.

Hardware Blocker Bypass: Some flows have a single hardware dependency that blocks everything downstream. Scanning a QR code to unlock a bike. Tapping an NFC chip to start a session. These physical interactions can't be automated—but the result of the interaction can be simulated.

How Pie implements this:

Pie's scripting layer can call customer APIs before, during, or after test execution. A typical pattern: Pre-test API call seeds test account with required device state → Test execution navigates the UI and verifies behavior → Post-test API call resets account for next test run.

Strategy 3: Pre-Configured Test Accounts

The insight: Each test account can represent a different device state, usage pattern, or edge case.

Hardware testing traditionally requires manipulating physical devices to create test scenarios. Want to test what happens with a low battery? Drain the battery. Want to test behavior after 30 days of usage? Wait 30 days.

With pre-configured accounts, the backend represents whatever state you need. The account is the test fixture.

The strategy: Create a library of test accounts, each configured to represent a specific scenario. Run tests across accounts to validate behavior under different conditions.

Account Type	Represents	Tests
New user, no device	Fresh signup, pre-pairing	Onboarding flows, setup prompts
Device connected, no data	Just paired, awaiting first use	Empty states, data collection prompts
Active user, 7 days	Regular usage, recent data	Standard dashboards, recent activity
Power user, 90+ days	Long-term usage, rich history	Historical views, trend analysis
Edge case: gaps in data	Sporadic usage, missing days	Gap handling, interpolation logic
Edge case: extreme values	Unusual readings, outliers	Boundary conditions, error states

How Pie implements this:

Pie's credential management allows unlimited test accounts. Each account maps to a scenario. Tests specify which account to use: "Login as new-user-no-device and verify onboarding prompt appears" or "Login as power-user-90-days and verify trend chart displays correctly."

Strategy 4: Visual AI for Hardware State Detection

The insight: When hardware state shows up in the UI, vision beats selectors.

Hardware apps often display device status through visual indicators: icons, colors, badges, animations. A green dot means connected. A red icon means error. A pulsing animation means syncing.

Traditional automation struggles with these indicators. The element might exist in the DOM, but its meaning depends on visual properties that selectors can't interpret.

The strategy: Use Pie's visual AI to interpret hardware state indicators the way a human would - by recognizing what the visual elements represent.

Symbol recognition over color detection: Color is unreliable. Different devices render colors differently. Themes change palettes. Symbols are stable. An icon showing a sun with a line through it means "off." An icon showing a sun without a line means "on." The shape conveys meaning regardless of color.

Text confirmation as backup: When hardware state changes, apps often display confirmation text: "Light turned on," "Device connected," "Sync complete." This text is unambiguous and easily verified.

How Pie implements this:

Test instructions describe what to verify in human terms: "Verify the device status shows connected" or "Verify the light toggle shows ON state." The platform interprets these instructions by analyzing what's visually present on screen.

Strategy 5: Read-Only Testing Against Live Data

The insight: Some data can't be faked convincingly. Real usage produces real edge cases.

Seeded test data covers known scenarios. But hardware apps encounter patterns that nobody anticipated - unusual usage sequences, rare environmental conditions, data combinations that only emerge over months of real-world operation.

The strategy: Use real user accounts (with permission) for read-only validation. Tests observe and verify but never modify.

How it works:

- Internal team members volunteer accounts linked to real devices they actually use
- Tests are constrained to read-only operations - viewing screens, checking data displays, verifying calculations
- Tests run during safe windows (e.g., daytime for sleep tracking apps, when devices aren't actively in use)
- No test ever modifies data, triggers device actions, or alters account state

How Pie implements this:

Pie's test constraints ensure read-only behavior. Tests navigate and observe but don't click action buttons. Verification is visual: "Does this chart render correctly?" not "Does clicking this button work?" Test accounts are flagged as read-only, and tests fail if they attempt modifications.

Case Studies

Fi: From 2-3 Day Releases to Same-Day

Company : Fi designs GPS smart collars for dogs. Their app displays location, activity, sleep patterns, and lets owners control the collar's LED light.

The challenge : Every release required 2-3 days of testing. The QA process involved 12+ engineers manually verifying device interactions, data displays, and feature functionality. Release velocity was constrained by testing capacity.

The Solution : We mapped Fi's app against the layered testing architecture. The analysis revealed that 80%+ of user flows had no hardware dependency. For flows requiring device data, we worked with Fi's backend team to enable test account seeding. For hardware state verification (LED status), we implemented visual AI detection.

Metric	Before Pie	After Pie
Release validation time	2-3 days	Same day
Engineers involved in testing	12+	1 QA lead + Pie automation
Automated test coverage	Minimal	Hundreds of tests
Test maintenance burden	Constant firefighting	Near-zero (self-healing)

"Pie is now an integral piece of our release process. If we were to split ways or something were to happen and we weren't able to get coverage for a week, I'm really not sure what we would do."

- Phillip Hunt, QA Lead, Fi

Lumo: Untangling a Testing Architecture

Company: Lumo develops agricultural IoT—soil moisture sensors, irrigation controllers, and farm monitoring systems for farmers managing water usage.

The Challenge: Testing was tangled. Firmware bugs, backend issues, and frontend problems surfaced together. Failure origin was unclear.

The Solution: Testing was separated into three distinct layers with defined responsibilities and approaches:

Layer	Responsibility	Testing Approach
Firmware	Device-level behavior	Dedicated hardware team, physical devices
Backend	API logic, data processing	API testing tools (Postman, Runscope)
Frontend	UI rendering, user experience	Pie automation

> "Creating a test prompt took one minute. If done manually, checking different webpage elements would take 30 minutes to one hour."
>
> **— Senior Software Engineer, Lumo**

Smart Mattress Company: 111 Test Cases in 2 Hours

Company : The smart mattress firm manufactures temperature-controlled smart mattresses. Their app displays sleep scores, temperature graphs, health insights, and enables user settings adjustment.

The Challenge : Testing sleep data visualizations required actual sleep data—someone sleeping on connected mattresses awaiting data accumulation. Manual test creation couldn't match development pace.

The Solution : Pie's discovery process was run against their app without hardware context. The AI autonomously explored applications, identified features, and generated test cases. Discovery produced 111 test cases in approximately 2 hours.

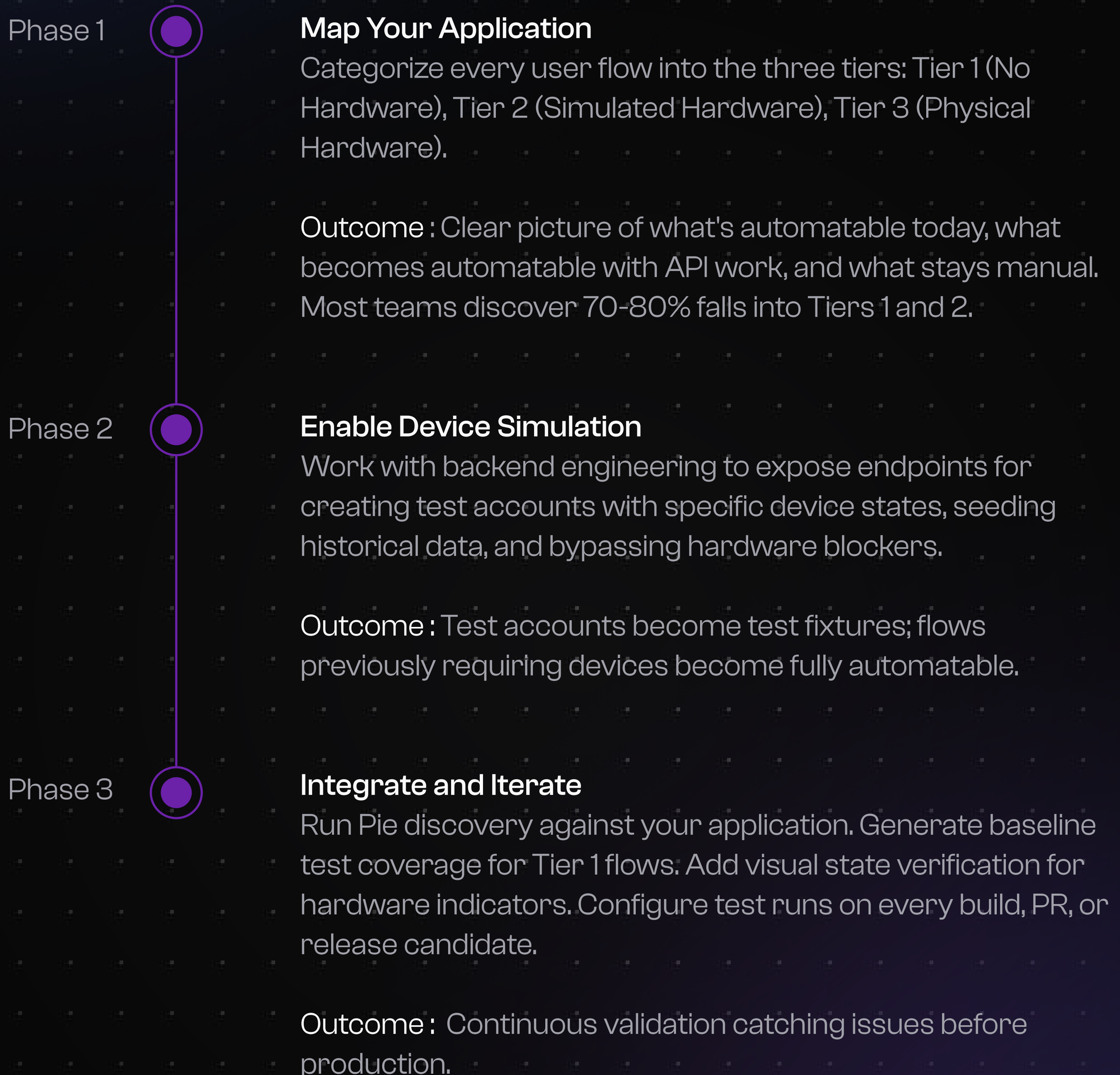
111
Test Cases Generated

~2hrs
Discovery Time

> "If we can get accounts for just looking at metrics without tapping action buttons, just clicking graphs... that could be valuable because it eliminates one entire testing class of graph verification."
>
> **- Engineering Lead, Smart Mattress Company**

Getting Started

Implementation Steps



Phase 1

Map Your Application

Categorize every user flow into the three tiers: Tier 1 (No Hardware), Tier 2 (Simulated Hardware), Tier 3 (Physical Hardware).

Outcome : Clear picture of what's automatable today, what becomes automatable with API work, and what stays manual. Most teams discover 70-80% falls into Tiers 1 and 2.

Phase 2

Enable Device Simulation

Work with backend engineering to expose endpoints for creating test accounts with specific device states, seeding historical data, and bypassing hardware blockers.

Outcome : Test accounts become test fixtures; flows previously requiring devices become fully automatable.

Phase 3

Integrate and Iterate

Run Pie discovery against your application. Generate baseline test coverage for Tier 1 flows. Add visual state verification for hardware indicators. Configure test runs on every build, PR, or release candidate.

Outcome : Continuous validation catching issues before production.

Key Takeaways

Five Strategies Summary

- Layered testing architecture separates device-dependent from device-independent functionality. Most hardware apps are 70-80% software fully automatable today.
- API-driven simulation provides device data without devices. Applications don't know or care data origins.
- Pre-configured test accounts turn scenarios into fixtures without device manipulation.
- Visual AI interprets hardware state indicators through screen observation like humans do.
- Read-only testing validates against real data patterns synthetic data cannot replicate.

Results Across Companies

Company	Responsibility	Testing Approach
Fi: Release Validation	2-3 days	Same day
Lumo: Architecture	Tangled layers	Clean separation
Smart Mattress App: Test Generation	Manual creation	111 tests in 2 hrs

The Future is Autonomous

Hardware apps have earned their reputation as automation-resistant. Device state, timing variability, multi-layer complexity—these are real challenges.

But the teams we worked with proved something important: the right strategies make hardware testing automatable. Fi and Lumo didn't accept manual testing as inevitable. They found a different way.

The question isn't whether your hardware app can be automated. It's whether you're ready to try.

See Pie on Your App

Drop in a URL. Get a Readiness Score. No sales call required.

Book a demo   pie.inc