

AI-Assisted Testing of Personal Injury Management Platform

Overview:

We developed an AI-assisted test automation program for an enterprise platform designed to manage personal-injury cases. The platform connects medical providers, law firms, and internal staff, supporting key processes such as medical record management, appointment requests, medical billing, and plaintiff funding. The platform comprises a core application, a modernized legacy case-management system, and a legacy appointment-tracking API, requiring test automation across multiple interconnected components.

In its earlier state, every release depended heavily on manual regression: 2.5 days to 2 weeks per cycle, 2–4 rounds per release, and 6–8 QA engineers plus developers getting pulled in to “swarm.” In a certain instance, one of the sprints delivered only 44% of its committed story points because regression consumed QA bandwidth, and a scheduled release slipped a full cycle after regression found blocking bugs. In order to address these cases, the QA team introduced an AI-assisted testing system: specialized agents for test authoring, live-fixture discovery, pull-request review, and regression reporting — each guided by written repository steering documents and backed by deterministic Node.js scripts, so that the AI agents are never left to guess.

The agents now assist engineers in drafting and extending Playwright test specs, resolving live test data per environment tier, reviewing every pull request against a fixed checklist, and narrating root causes in sprint regression reports. All this runs inside one shared automation repository covering every product, with results reported automatically to Microsoft Teams.

This AI-assisted workflow helped grow the automated test suite by 174% over 12 months (2.7×), while a full regression of 2,187 tests now completes in under 30 minutes at a 95.6% pass rate — work that earlier took days and multiple manual rounds.

Client details:

Name: Confidential | **Industry:** Healthcare, Software | **Location:** USA

Technologies:

QA Automation & AI Enablement

TypeScript, Playwright Test, Claude Code, steering documents; Bitbucket Pipelines, Azure DevOps, Azure Playwright Service; Jira, Xray, Confluence; C# / .NET 8 (xUnit, NUnit, Moq, FluentAssertions, NetArchTest); Angular, React, Jest, React Testing Library, MSW; Page Object Model; Microsoft SQL Server.

AI-Assisted Testing of Personal Injury Management Platform

Project Description:

Problem Statement:

In its existing state, QA was a manual, multi-day, multi-round bottleneck that competed with feature delivery for the same engineers.

- Manual regression cycles took **2.5 days to 2 weeks** per release, needing **2 to 4 rounds** and **6–8 QA engineers** plus developers pulled in to swarm on testing.
- Regression consumed QA bandwidth so completely that one sprint delivered only **44% of its committed story points**; a scheduled release slipped a full cycle after regression found blocking bugs.
- Environment and login outages blocked all testing for **20 hours to 41 days** at a time, across **5 pre-production tiers** (now 4) that routinely drifted from each other.
- CI incidents in the automation itself — a browser-leak freeze, a CI runner unable to reach UAT, a certificate-provisioning race — each took **1.7 to 32 days** to permanently fix.
- Test management lagged automation: **99.7% of 4,917 Xray test cases** were still typed Manual, and next-generation-platform automated coverage was **~1% UI, ~72% BFF API, 0% core API**.
- Delivery was hotfix-heavy: **85% of 2026 core-application production releases** were unplanned patches or off-cycle releases rather than the scheduled minor.
- A push-triggered CI pipeline ran for **3.5 months at a 0% success rate** before being retired — noise, not signal.

Solution :

This AI-assisted testing system is built around specialized agents operating inside one shared Playwright/TypeScript automation repository. Each agent is powered by Claude Code, guided by written steering documents that encode the team's standards and real-incident "never-do" rules, and backed by deterministic Node.js scripts that handle the work that must never be guessed: parsing raw results, computing pass/fail/flaky metrics, a 14-run flake history, and Teams notifications. The agents form a traceable pipeline:

- **Agent: Test Authoring** — drafts and extends Playwright specs against the repository's Page Object Model and tag conventions (@smoke / @regression / @destructive / @smokeFixMe), for API, UI and cross-system integration suites.
- **Agent: Fixture Discovery** — finds live, valid test data on the target environment tier instead of hardcoding IDs, tokens or emails; hardcoded exceptions must be tagged and justified.
- **Agent: PR Review** — runs automatically on every pull request: a fixed checklist, a pattern sweep across the whole diff, a sibling-file parity check, and a mandatory type check, before a human approves and merges.

AI-Assisted Testing of Personal Injury Management Platform

- **Agent: Regression Reporter** — narrates root causes and ticket mismatches on top of the deterministic sprint-regression report; fail-soft by design, so a narration failure never blocks the metrics or the Teams post.

The suite spans 747 spec files and roughly 8,500 declared tests across 12 active suites — API, UI, integration and cross-system data reconciliation — for the core application, the next-generation platform, the legacy appointment API, and their surrounding integrations. Daily QA-check pipelines and a sprint-aligned full regression run give both release rhythms (monthly for the core application, roughly biweekly for the next-generation platform) continuous, scheduled evidence instead of a single pre-release push. Throughout the process, Jira remains the single source of truth: every branch, commits and pull requests carry the originating work-item key, so every artifact traces back to it.

Methodology:

The program keeps AI output consistent and trustworthy by separating deterministic work from creative work, and by making humans sign-off the gate before anything ships:

- **Steering documents:** repository guidance files and agent definitions encode naming conventions, tag rules and explicit “never-do” rules — drawn from real incidents — constraining the AI to the team’s standards.
- **Deterministic scripts:** Node.js parses raw test results, computes every release metric (pass/fail/flaky, duration anomalies, run-over-run history), and posts to Teams; the AI only interprets those facts and never invents a number.
- **Two-pass review:** scripted pre-checks (stale-branch, tag-hygiene) run first on every pull request and must be treated as confirmed findings; the AI reviewer then works a fixed checklist — code quality, test design, hardcoded data, security, tags, execution evidence — and the cycle repeats until the review comes back clean.
- **Quality gates:** every pull request must state the environment tested, attach run evidence, and list known gaps not covered — AI-assisted PRs leave cases unautomated rather than faking them with a mocked response.
- **Human sign-off & traceability:** a human approves and merges every pull request, and a human closes every Jira ticket after re-testing the fix live — a status comment claiming “deployed” is treated as a claim, not evidence. Traceability runs work item → test → test data → execution → evidence.
- **Sensitive-data controls:** automated tests run only against non-production tiers; database checks use a read-only, write-free login; credentials are masked in logs and come from secured CI variables; state-changing tests are tagged @destructive and clean up after themselves.

AI-Assisted Testing of Personal Injury Management Platform

Architecture:

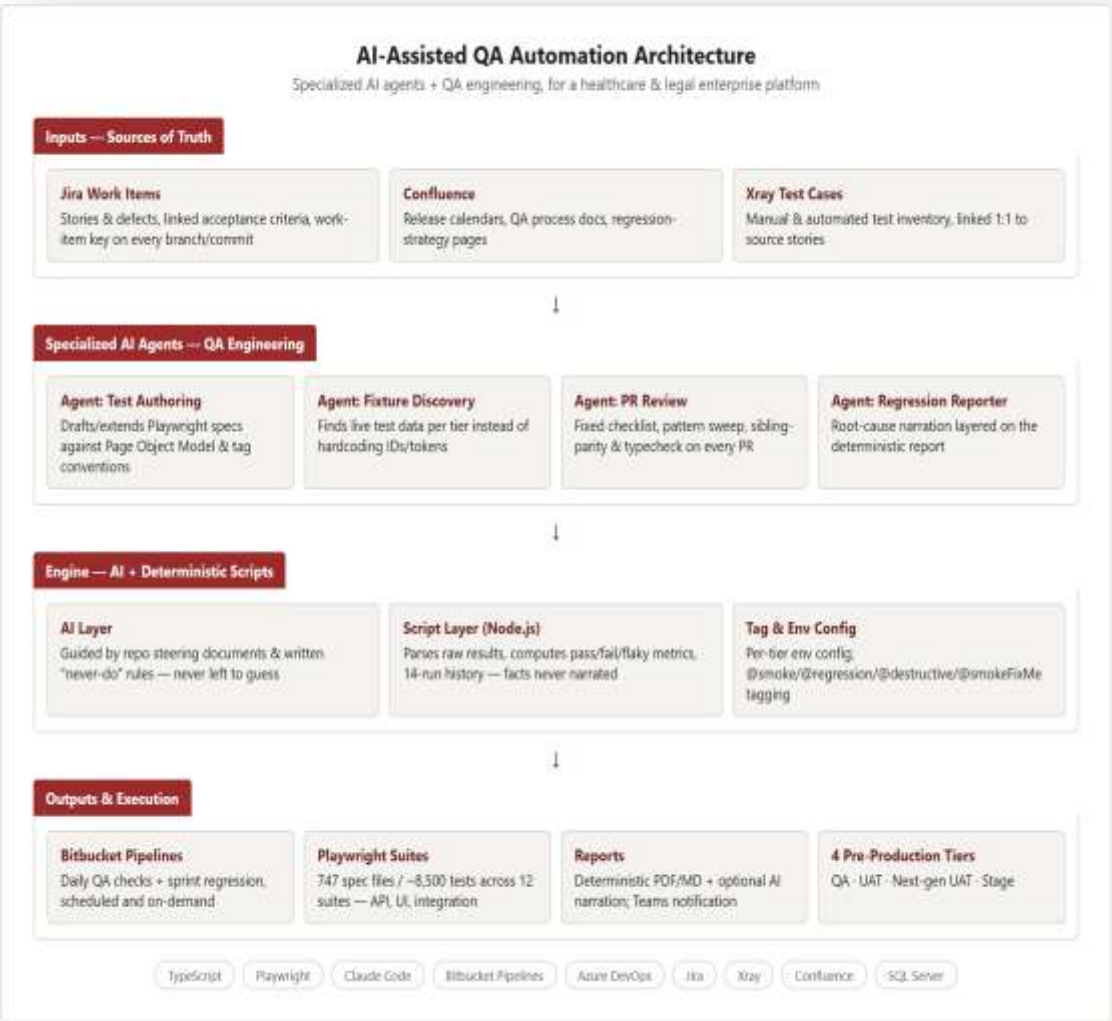


Figure 1. Architecture — Jira/Confluence/Xray inputs feed four specialized AI agents, backed by deterministic scripts, producing scheduled pipeline execution across four pre-production tiers.

AI-Assisted Testing of Personal Injury Management Platform

Workflow:

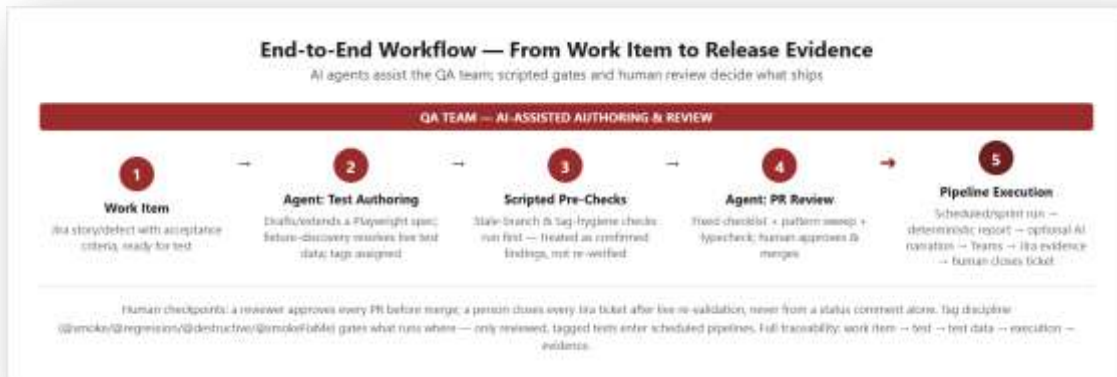


Figure 2. End-to-end workflow — the QA team authors and reviews with AI assistance; scripted gates and human sign-off decide what reaches a scheduled pipeline and what closes a ticket.

Screenshots:

The following views show the chain in action — from a real passing end-to-end test result to a real scheduled pipeline’s failure handling and reporting.

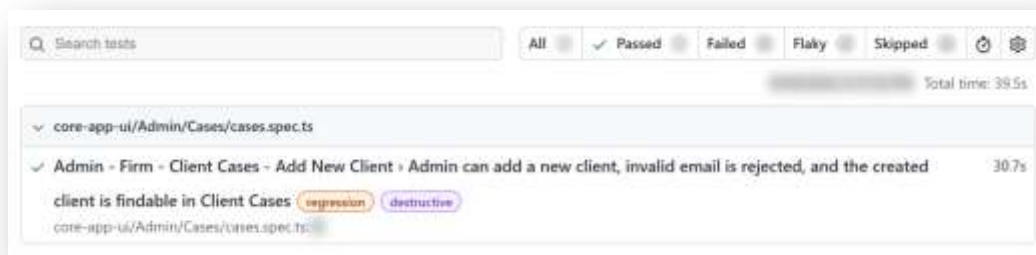


Figure 3. A real, passing end-to-end UI test result (Playwright report) — an Admin “add new client” flow, tagged regression and destructive.

AI-Assisted Testing of Personal Injury Management Platform

What a daily check reports: real pipeline log excerpt

Most recent next-gen UI daily check. Product names are redacted and absolute counts are masked (■); the percentages and messages are original.

```
...  
[run-with-retry] At least one test actually ran - this is a real test failure, not retrying  
                (never mask real failures behind an automatic retry).  
  
[generate-report] ■ tests - ■ passed - ■ failed - ■ skipped - ■ known issue  
X DocumentCenter ■ passed ■ failed (38%)  
X BillOfSale      ■ passed ■ failed (8%)  
X Firms           ■ passed ■ failed (8%)  
  
[generate-report] Known issues that are currently FAILING (tagged, but not passing as expected):  
X Creating a Branch Firm via the plain Add form succeeds without any Enterprise link [■■■■■]  
  
[generate-report] Failure groups:  
■ X Test timeout of 60000ms exceeded.  
■ X Test timeout of 120000ms exceeded.  
■ X Test timeout of 300000ms exceeded.  
■ X Test timeout of 300000ms exceeded while running "beforeEach" hook.  
■ X Error: expect(locator).toBeVisible() failed  
  
[regression-summary] ■ tests - ■ passed - ■ failed - ■ skipped - 14.93% pass rate  
[notify-teams] Posted regression summary to Teams.
```

Source: step log of the scheduled next-gen UI daily check, retrieved through the Bitbucket Pipelines API. The run shows the safeguards working as designed: the retry wrapper refused to hide a real failure, known issues were listed separately, failures were grouped by cause, and the summary was posted to Teams.

Figure 4. A real daily-check pipeline logs the retry wrapper refusing to mask a real failure, known issues listed separately, failures grouped by cause, and the summary posted to Teams.

AI-Assisted Testing of Personal Injury Management Platform

Key Outcomes & Business Impact:

- **Dramatically faster regression:** a full 2,187-test regression now completes in under 30 minutes at a 95.6% pass rate, versus 2.5 days to 2 weeks and 2–4 manual rounds before automation.
- **Growth, not just speed:** the automated QA suite grew 174% in 12 months (2.7×); core-application backend unit tests grew 216% and frontend test files grew 451% over the same period.
- **Every failure explained, not just counted:** 100% of the failures in a recent full regression were traced to a root cause — none was an unexplained backend regression — and flaky tests were about 0.05% of the suite.
- **Quality gates that block:** an 85% line-coverage gate and an architecture-conformance gate now fail the build, where no such gate existed roughly 12 months ago.
- **Consistent, auditable review:** steering documents, scripted pre-checks and a two-pass AI-plus-human PR review enforce the same standard on every change, regardless of who opens it.
- **Full traceability with human sign-off:** every artifact links work item → test → test data → execution → evidence, and a human — never the AI — approves the merge and closes the ticket.
- **Honest about what's unfinished:** scheduled daily checks are still red more often than not (20% fully green over a recent five-day window), and human approval on merges is practiced but not yet enforced by branch policy — both tracked openly on the roadmap rather than hidden.

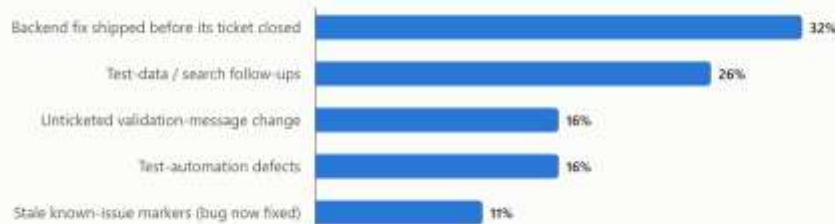
A recent full regression run: fast, stable, fully explained

Next-gen platform BFF API suite, QA environment, CI-equivalent retries



Where the failures came from

Share of that run's failures by root cause



Source: the automated regression report for that run (deterministic metrics + AI-narrated root-cause analysis)

Figure 5. A recent full regression run executed by the QA team — results captured, every failure root-caused, and reported.