

Accelerating Enterprise Test Automation with Large Language Models: Building a Multi-Client Playwright Framework with Claude

A Practitioner Case Study on AI-Assisted Software Test Engineering

Rohit Kumar Sharma, Vidur Shukla, Jyoti Misra Parashar, Ritesh Kumar

Accenture — Marketing Analytics & Automation

Correspondence: rohit.b.kumar.sharma@accenture.com

vidur.shukla@accenture.com

jyoti.misra.parashar@accenture.com

ritesh.cr.kumar@accenture.com

Abstract

Anyone who has tried to build reliable, end-to-end test automation for a modern web application knows how quickly the complexity piles up — asynchronous rendering, enterprise auth, multi-tenancy, and fragile selectors all at once. This paper documents how I built a production-grade, multi-client test automation framework for a Marketing Mix Modelling platform, with Claude (Anthropic's LLM) as a genuine part of the development process. It wasn't a single-step effort. The work started as a rough Python script to prove the concept was even feasible, then evolved into a structured TypeScript/Node.js framework built around the Page Object Model. Along the way, a set of engineering patterns emerged from real debugging sessions and failures — a multi-selector fallback strategy, automated TOTP-based MFA, session-state reuse, a smart calendar-date algorithm, a clipboard-retry technique for asynchronous data exports, and a custom performance reporter. This paper describes the human-AI workflow behind all of that, the architecture it produced, and the lessons I'd pass on to any engineer considering a similar approach. The honest summary: LLM assistance genuinely compressed timelines and surfaced patterns I might have missed, but only because a human stayed in the loop throughout — reviewing every output, supplying ground truth, and making the final call on correctness.

Keywords

Test automation; Playwright; Large Language Models; Claude; AI-assisted software engineering; Page Object Model; Multi-factor authentication; Continuous Integration; Data validation; TypeScript.

1. Introduction

End-to-end test automation has a reputation problem. Everyone agrees it's valuable — until the maintenance bill arrives. Selectors break after a UI update, async timing issues cause intermittent failures, enterprise login flows refuse to be scripted cleanly, and the whole suite gradually becomes a burden rather than a safety net. For engineers working on analytics or enterprise platforms, the problem is sharper: domain experts who understand the business logic often aren't the same people who know their way around Playwright or Selenium, and that skills gap makes automated testing slow to adopt.

What changed for me was bringing an LLM into the development loop — not as a replacement for engineering judgement, but as a collaborator. Claude turned out to be genuinely useful for proposing structure, generating boilerplate, explaining Playwright idioms I hadn't encountered before, and diagnosing selector failures faster than I could triage them alone. This paper is an account of that experience: how I used Claude to build a full, production-oriented Playwright framework for an enterprise Marketing Mix Modelling (MMM) platform.

The paper covers four things:

1. **Workflow** — the actual human–AI development process I used, phase by phase;
2. **Architecture** — the framework structure and the specific engineering patterns that came out of it;
3. **Evaluation** — a before/after comparison between the initial script and the final framework;
4. **Guidance** — practical advice (and a few warnings) for engineers considering a similar approach.

2. Background and Related Work

2.1 Playwright and Modern E2E Testing

Playwright is Microsoft's browser automation framework — it runs Chromium, Firefox, and WebKit from a single API and handles a lot of the hard parts automatically: waiting for elements, network interception, video tracing, and parallel execution. Its TypeScript support is solid, which matters for maintainability in a large suite. The Page Object Model (POM) is the standard structural pattern for Playwright projects: each screen or component becomes a class that owns its locators and interactions, so when the UI changes, you update the page object and the tests themselves stay untouched.

2.2 LLM-Assisted Software Engineering

LLM coding tools have moved well past simple autocomplete. The ability to reason across multiple files, propose refactors, and iterate on failing code makes them particularly useful for test engineering, where a lot of the work is repetitive (writing locators, handling waits, structuring assertions) but the edge cases still need careful thought. Claude supports long context windows, which turned out to matter: I could paste the entire existing script plus a test plan and ask for a coherent migration proposal rather than working through it piecemeal.

3. The System Under Test

The platform I was automating is an enterprise Marketing Mix Modelling tool. Analysts use it to create planning scenarios, run optimisations, execute simulations, and explore a range of insight charts — summary, trends, monetised and non-monetised details, time-series, halo, and network views. Several things about it made automation genuinely difficult:

- **Hardened login** — auth is delegated to an enterprise identity provider, with MFA on top;
- **Heavy async UI** — charts render asynchronously after a scenario is saved, and full population can take close to two minutes;
- **Multi-tenancy** — different clients have different credentials, base data, and expected values, so nothing could be hard-coded;
- **Clipboard-based export** — chart data comes via copy-to-clipboard, but the clipboard initially holds a placeholder string while the real payload loads asynchronously;
- **Data validation** — captured results had to be validated against a reference calculation derived from a source CSV, not just against arbitrary expected values.

The suite had to cover the full analyst workflow end-to-end: login, create a scenario plan, capture all insight charts into a multi-sheet Excel workbook, run goal-based and spend-based optimisations, and execute both effort- and spend-based simulations. Every run also had to emit performance metrics.

4. Methodology: The Human–AI Development Workflow

The framework didn't come out of a single design session. It was built through an iterative, back-and-forth process — I set direction, supplied ground truth, and validated results, while Claude proposed designs, generated code, and helped diagnose failures. It ran across five broad phases.

4.1 Phase 1 — Exploratory Automation (Python)

Before worrying about architecture, I needed to know whether the automation was even viable. The first iteration was a single Python script — around 1,400 lines — using Playwright for Python. The goal was pure discovery: learn the DOM structure, confirm that MFA could be handled with a TOTP secret, and prove the end-to-end journey could

actually be scripted. Claude helped draft the initial script and suggest locator strategies; I was also picking up Playwright idioms I hadn't used before, so having something explain them inline was useful.

4.2 Phase 2 — Architecture and Migration Planning

Once I was confident the automation was feasible, the question was how to structure it properly. I gave Claude the existing Python script and the multi-client test plan, and asked it to propose a maintainable architecture. The recommendation was a TypeScript/Node.js framework built on the Page Object Model, with clear separation between pages, tests, config, test data, reporters, and setup/teardown. What helped here was that Claude could read the entire existing script and map each procedural step onto the proposed structure — so before any TypeScript was written, there was already a clear picture of where each piece would live.

4.3 Phase 3 — Incremental Implementation

The actual build was incremental. I tackled one component at a time — base page, login page, scenarios page, optimisation and simulation pages, the reporter, one spec file per journey. Each piece was generated, run locally, debugged, and refined before moving on. The feedback loop with Claude was tight: when a selector failed or a chart returned placeholder data instead of real values, I'd paste the error or unexpected behaviour back in and get a targeted fix. Some of the most useful patterns came directly from those exchanges — the multi-selector fallback strategy and the clipboard-retry logic both emerged from debugging sessions, not upfront design.

4.4 Phase 4 — Hardening and Operationalisation

Once the happy path was solid, I turned to making the suite reliable in practice: session-state reuse so MFA was solved only once per run, sequential project dependencies to enforce execution order, automatic retries and traces on failure, a performance reporter writing JSON and CSV, and a GitHub Actions pipeline for scheduled and on-demand runs. Claude generated most of the reporter and the CI scaffold — this kind of boilerplate is where it's most efficient.

4.5 Phase 5 — Documentation

Finally, I had Claude generate the user guide, setup guide, and framework summary. Documentation is easy to skip when you're deep in implementation; having it produced alongside the code meant other engineers could pick up the framework without needing me to walk them through it.

The split of responsibility across the workflow looked roughly like this:

Responsibility	Human Engineer	Claude (LLM)
Intent & requirements	Owens	Clarifies
Architecture design	Approves	Proposes
Code generation	Reviews	Drafts
Ground-truth data & expected values	Owens	Consumes
Debugging failing selectors	Observes & reports	Diagnoses & fixes
Security & credential handling	Owens	Advises
Final validation & sign-off	Owens	—

5. Framework Architecture

The framework ended up with a layered structure where each layer has a single clear job. Tests describe intent; page objects handle the interaction details; config and data files drive per-client behaviour; and cross-cutting concerns like reporting, setup, and utilities sit in their own modules. The goal was that changing one thing — a locator, a client's base URL, the reporter format — should touch exactly one place.

```

automation-framework/
├── playwright.config.ts      # projects, sequencing, reporters
├── src/
│   ├── pages/
│   │   ├── BasePage.ts      # shared interaction primitives
│   │   ├── LoginPage.ts     # enterprise login + TOTP MFA
│   │   ├── ScenariosPage.ts  # scenario creation, chart capture
│   │   ├── OptimizationPage.ts # goal/spend optimisation
│   │   └── SimulationPage.ts # effort/spend simulation
│   ├── reporters/performance-reporter.ts
│   ├── setup/{global-setup,global-teardown}.ts
│   └── utils/csv-helper.ts
├── tests/                    # one spec per analyst journey
├── config/{client1,client2}.config.ts
└── data/{client1,client2}/*.json # credentials & expected values

```

5.1 Layered Responsibilities

- **Base page** — shared primitives like navigation, screenshots, waits, and the fallback interaction helpers. Anything used by more than one page object lives here.
- **Page objects** — one class per screen; owns all the locators and interaction logic for that screen, so tests never contain raw selectors.
- **Specs** — structured as Playwright `test.step()` blocks, which makes the execution log read like a plain-English checklist of what happened and where it failed.
- **Config & data** — per-client base URLs, expected investment totals, and file paths. This is what lets the same codebase serve multiple tenants without hard-coded values.

5.2 Sequential, Dependency-Aware Execution

The analyst journey has a natural order — you can't create a scenario before logging in, or run an optimisation before creating a scenario. The Playwright config reflects this: each stage is a project with explicit dependencies on the ones before it. Authentication is handled once in the first project, the session state is serialised to a file, and every downstream project loads from that file. In practice, MFA is solved exactly once per run regardless of how many projects follow.

```

projects: [
  { name: 'test0-verify-login', testMatch: 'verify-login.spec.ts' },
  { name: 'test2-create-scenario',
    dependencies: ['test0-verify-login'],
    use: { storageState: SESSION_FILE } },
  { name: 'test3-run-optimization',
    dependencies: ['test2-create-scenario'],
    use: { storageState: SESSION_FILE } },
  // spend optimisation, spend & effort simulation ...
]

```

6. Reusable Engineering Patterns

Six patterns came out of this project that I think are broadly applicable beyond this specific platform. Each is described below with the problem it addresses.

6.1 Multi-Selector Fallback Strategy

Enterprise UIs are inconsistent about test identifiers, and identity-provider login pages vary between tenants. Binding to a single locator means a minor markup change breaks the test. Instead, each interaction tries an ordered list of

candidate selectors, falls back down the list on failure, and uses a forced click as a last resort. It's a simple idea, but it makes a real difference to flakiness in practice.

```
async clickWithFallback(selectors: string[]): Promise<boolean> {
  for (const selector of selectors) {
    for (const el of await this.page.locator(selector).all()) {
      if (await el.isVisible().catch(() => false)) {
        await el.click(); return true;
      }
    }
  }
  // last resort: force-click first match
}
```

6.2 Automated TOTP Multi-Factor Authentication

MFA is one of the first things people cite when they say UI automation is 'impossible' on a given platform. The approach here is straightforward: store the TOTP secret as an environment secret (never in source control), use the otpauth library to generate valid one-time codes programmatically, and the identity provider can't tell the difference from a real user. It requires access to the raw TOTP secret rather than just an authenticator app, which needs organisational sign-off — but once that's in place, it works reliably.

```
const totp = new OTPAuth.TOTP({
  secret: OTPAuth.Secret.fromBase32(process.env.TOTP_SECRET),
});
const code = totp.generate(); // 6-digit, time-synchronised
```

6.3 Session-State Reuse

Running through MFA on every test is slow and risks hitting rate limits on the identity provider. Instead, the first test captures the authenticated browser context as a session file, and every subsequent project loads from it. This shaved significant time off each full run and removed a whole category of intermittent auth failures.

6.4 Smart Calendar-Date Algorithm

The planning form has an unintuitive date requirement: the end date must fall exactly one year after the start date, aligned to the same weekday — specifically, the last occurrence of that weekday in the target month. It's the kind of rule that's easy to get wrong manually and impossible to hard-code reliably across clients. The framework reads the form's start date, computes the correct target algorithmically, and includes retry logic to handle the calendar widget's occasional quirks. Without this, you'd get silent failures every time the month boundary shifted.

6.5 Resilient Clipboard Capture with Retry and DOM Fallback

The platform's export mechanism is copy-to-clipboard, but clicking the copy button immediately gives you a placeholder string (like 'KPICardClipBoardData') while the real data finishes loading asynchronously. The framework detects known placeholder values, re-clicks and waits, and retries. There's a wrinkle with the KPI card specifically: re-clicking there resets the copy state rather than retrying it, so for that element, the fallback is scraping the value directly from the DOM. Once all the clipboard data is collected, it goes into a multi-sheet Excel workbook.

```
const isPlaceholder = t =>
  t.length === 0 || ['KPICardClipBoardData','ClipBoardData',
    'Loading...'].includes(t.trim());
// retry: re-click + wait until real data arrives,
// else fall back to DOM scraping for the KPI card
```

6.6 Custom Performance-Metrics Reporter

Rather than relying on Playwright's built-in reporters alone, I built a custom one that captures per-test and per-step timings, status, retry counts, client, and browser, then writes JSON and CSV artefacts alongside a console summary

that flags the slowest tests. The schema was designed to be database-ready from the start — the idea being that over time, these timings could feed a trend dashboard and turn the test suite into a source of continuous performance telemetry, not just a pass/fail indicator.

7. Results and Discussion

Moving from the exploratory Python script to the TypeScript framework produced real, visible improvements across almost every dimension. This is an experience report rather than a controlled experiment, so I'm not claiming precise productivity measurements — but the structural differences speak for themselves.

Aspect	Exploratory Python Script	LLM-Assisted TS Framework
Structure	One file (~1,400 lines)	20+ modular files
Design pattern	Procedural	Page Object Model
Multi-client support	Hard-coded	JSON data per client
Type safety	None	TypeScript, end to end
Reporting	Console only	HTML + JSON + CSV + custom
Performance tracking	None	Automatic, per-step
CI/CD	None	GitHub Actions pipeline
Failure diagnostics	Manual	Trace + video + screenshot
Reusability	Low	High (base classes, configs)

7.1 Observed Benefits

- **Speed** — scaffolding and refactoring that would have taken days took hours. Page objects and the reporter are boilerplate-heavy by nature; having Claude generate first drafts was a real time save.
- **Quality** — idiomatic Playwright patterns like auto-waiting, storage state, and test.step were applied consistently because Claude surfaced them proactively, rather than me having to remember each one.
- **Knowledge transfer** — because documentation was generated as each component was built, the suite came with a user guide and setup instructions from day one. No knowledge siloed in my head.
- **Upskilling** — the debugging conversations were also learning sessions. My understanding of Playwright improved considerably through those exchanges.

7.2 Where the Human Was Indispensable

Claude was useful, but it couldn't replace the engineering judgement that made the suite actually correct. The reference investment values came from the source data, not from anything the model could infer. The decision about how to handle credentials securely was mine. The business date rule needed me to read the spec and verify the computed result was right — not just plausible-sounding. And every generated output needed review; the model produces confident code that can be subtly wrong in ways that only show up when you run it against the real system. The most reliable pattern was treating Claude as a fast, knowledgeable collaborator and reviewing everything it produced.

8. Best Practices for LLM-Assisted Test Automation

Based on this experience, here's what I'd tell someone starting a similar project:

5. **Prototype first** — write a small, throwaway script to learn the DOM and prove the automation is feasible before asking Claude to design a framework. You'll learn things from that script that should inform the architecture.

6. **Feed rich context** — the quality of Claude’s output is roughly proportional to the quality of context you give it. Share your test plan, existing code, and the actual errors you’re seeing; vague questions get vague answers.
7. **Work incrementally** — one page object, one spec, one reporter at a time. Run each piece before moving to the next. Large one-shot code generations are harder to validate and harder to debug when something goes wrong.
8. **Close the loop with reality** — when a test fails, paste the actual error message and unexpected behaviour back to Claude. This is where LLM diagnosis tends to be strongest; it can often spot what went wrong faster than manual triage.
9. **Own security yourself** — credentials, TOTP secrets, and session files should never enter source control. Keep them in environment variables and secret stores. Have Claude read from those; don’t let it handle or store the actual values.
10. **Validate every assertion** — never let the model invent ground truth. Expected investment values, chart totals, and business rule outcomes should come from you, derived from the actual source data.
11. **Design for resilience up front** — ask for fallback selectors, retry logic, and explicit waits from the start. Enterprise UIs are flaky by nature; building resilience in later is more painful than building it in from the beginning.
12. **Document as you go** — have Claude generate the README, setup guide, and inline comments alongside the code. Documenting at the end is a chore and it often doesn’t happen.

9. Limitations and Threats to Validity

- **Single case** — this is one system, one platform, one team. The patterns here may or may not apply elsewhere, and the productivity improvements I observed aren’t measured under controlled conditions.
- **Model fallibility** — generated code can look correct and still be wrong. Selectors that match in development may be fragile in production. Logic that sounds right in prose may have an off-by-one error hiding in it. Disciplined review is not optional.
- **Brittleness** — the framework reduces maintenance overhead compared to an unstructured script, but UI automation is always sensitive to application changes. When the product team redesigns a screen, the page objects will need updating.
- **Security & compliance** — automation of this kind touches credentials, session files, and potentially sensitive chart data. How those are stored, transmitted, and retained needs to align with your organisation’s policies. It’s easy to widen the attack surface if this isn’t thought through.
- **Reproducibility** — Claude’s capabilities and defaults change across versions. The specific prompts that worked well here may produce different results with a later model.

10. Future Work

- **Telemetry** — persist the performance metrics to a database and build a trend dashboard with regression alerting. The reporter already emits the right data; it just needs somewhere to land.
- **Coverage** — extend to additional tenants and add API-level validation alongside the UI checks. Some validations are more reliable and faster at the API layer.
- **Self-healing locators** — worth exploring: when a test fails due to a changed selector, can the LLM propose a repair automatically? The tooling for this is maturing.
- **Controlled study** — a proper comparison between LLM-assisted and traditional development on matched tasks would strengthen the claims here. Experience reports are useful; controlled measurements are more convincing.

11. Conclusion

The honest answer to ‘can an LLM help you build a serious test automation framework?’ is yes — with caveats. Starting from a messy exploratory Python script, Claude helped produce a layered, multi-client Playwright framework in TypeScript, with a proper Page Object Model, automated MFA, session reuse, resilient data capture, performance

telemetry, and a CI/CD pipeline. That's a real outcome, not a demo. But what made it work wasn't handing the problem to an AI — it was staying in control of the parts that matter. I owned the intent, the ground truth, the security decisions, and the final validation. Claude handled the acceleration: design proposals, code generation, debugging suggestions, and documentation. For engineers facing the recurring cost of E2E test automation, this kind of collaboration is worth trying — but only if you're prepared to review everything carefully and keep the human judgement firmly in the loop.

References

- [1] Anthropic. Claude (Large Language Model). <https://www.anthropic.com/claude>
- [2] Microsoft. Playwright: Fast and reliable end-to-end testing for modern web apps. <https://playwright.dev/>
- [3] M. Fowler. PageObject. <https://martinfowler.com/bliki/PageObject.html>
- [4] M. M. Rahman et al. "An Empirical Study on the Usage of Large Language Models for Software Testing." (survey literature on LLM-assisted testing).
- [5] IETF RFC 6238. TOTP: Time-Based One-Time Password Algorithm.
- [6] TypeScript. <https://www.typescriptlang.org/>
- [7] GitHub Actions Documentation. <https://docs.github.com/actions>