Cypress to Playwright migration guide

Ryan Song

ryan.song@iterable.com

Abstract

Cypress has served as our E2E testing tool for several years, offering simplicity and developer-friendly syntax. However, limitations such as performance issues with large test suites, and memory leak have surfaced as our testing needs evolved. To support a smooth transition from Cypress to Playwright as our end-to-end (E2E) testing framework, this proposal outlines the creation of a comprehensive Cypress to Playwright Migration Guide. The guide will serve as a hands-on reference for engineering teams, ensuring minimal disruption, knowledge transfer, and successful re-implementation of tests using Playwright.

Biography

Ryan Song is a Staff Test Engineer at Iterable with over 10 years of experience in the quality engineering field. He has contributed to projects across a wide range of environments—from startups and federal/defense sectors to Fortune 50 enterprises. Ryan is passionate about automation testing and continuous integration/continuous deployment (CI/CD), with a strong focus on system optimization and operations research. He holds a degree in Industrial and Systems Engineering from Texas A&M University and is currently based in Los Angeles.

Hackathon week

During Iterable's hackathon week, our quality engineering team explored whether a faster, more reliable alternative to Cypress could meet our growing testing needs. Cypress had served us well, but as our codebase expanded, we faced slower execution times for large test suites and recurring memory leaks that disrupted CI pipelines. Playwright quickly emerged as the top candidate thanks to its modern architecture, native parallel execution, robust cross-browser support, and strong community. To test its potential, we built a representative set of tests, ran them alongside Cypress, and tracked execution speed and memory usage. The results were clear: Playwright consistently ran faster, used fewer resources, and produced more stable results. This hackathon project provided the technical evidence and team confidence needed to form the cornerstone of our migration plan and begin a strategic, phased transition.

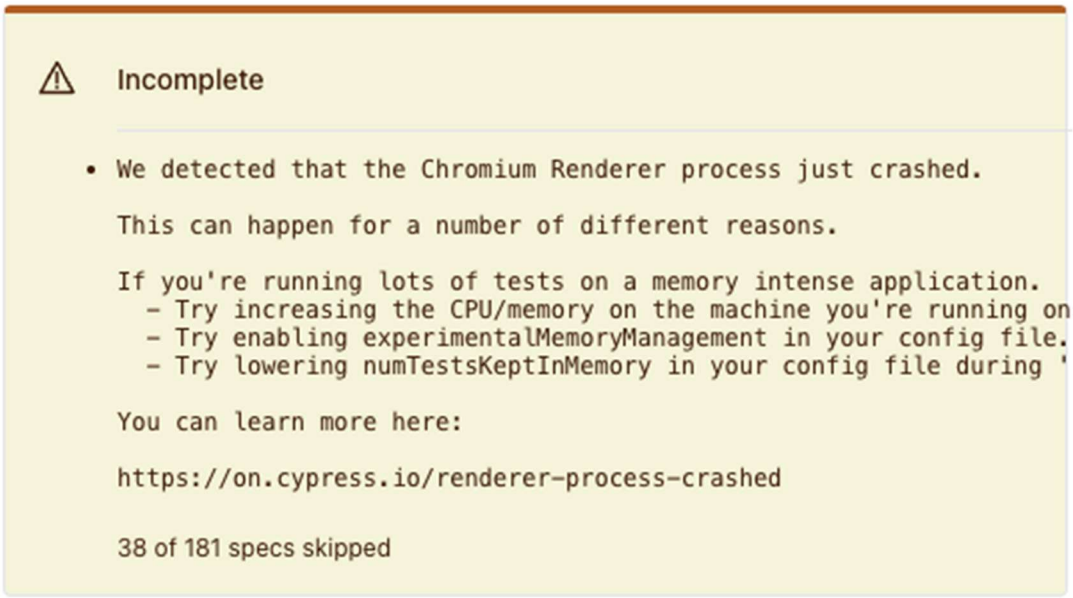

Figure 1 Example of Cypress out of memory issue

Introduction of the migration

Our transition from Cypress to Playwright is a deliberate, organization-wide effort to modernize our end-to-end testing framework, improve test performance, and reduce long-term maintenance costs. Cypress served us well for years with its developer-friendly syntax and ease of integration, but as our application and test coverage grew, we began encountering slower execution times, higher resource usage, and recurring memory leaks that affected CI stability. This migration guide is designed to support developers of all experience levels through a structured, practical approach. Migrating to a new framework is not just a technical change—it requires planning, shared understanding, and consistent best practices to ensure long-term success.

Our approach spans five key areas:

1. Proof-of-Concept Validation – Using a PoC to compare Playwright and Cypress head-to-head in speed, memory usage, and stability, providing the technical evidence to secure leadership buy-in.

2. Migration Tooling – Leveraging AI-assisted translation and specialized tools like Cursor.io to accelerate conversion while handling framework-specific differences.

3. Folder & Test Structure Improvements – Refactoring from ad-hoc selectors to a Page Object Model (POM) for cleaner, more maintainable, and scalable tests.

4. Timeline & Strategy – Executing migration in deliberate phases, starting with the most complex tests and including buffer time for staffing changes.

5. Final Touch-Up & Ongoing Support – Delivering documentation, utilities, and stability improvements to ensure long-term adoption and measurable ROI.

1. Proof-of-Concept: Validating Playwright

When adopting a new framework like Playwright, the first and most critical step is to create a proof-of-concept (PoC) that demonstrates clear, measurable value. A migration of this magnitude is more than a tooling swap—it represents an organizational shift that impacts workflows, long-term maintainability, and CI/CD stability. Building trust and confidence early is essential to secure buy-in from both leadership and developers.

Our PoC began after the Iterable's Hackathon Week, when the quality engineering team used the opportunity to evaluate whether Playwright could not only match Cypress's capabilities but also address its long-standing weaknesses. We designed the PoC to cover a range of scenarios, from basic functional tests that confirmed Playwright's syntax and API suitability, to complex cases involving database reads and writes through third-party libraries, and API response interception with mocked data. Each Playwright test was paired with an equivalent Cypress test, run side-by-side, and measured on execution time, CPU usage, and memory consumption.

The results were compelling: Playwright matched all of Cypress's features, performed just as well without running into memory issues, and executed faster in local environments. Its open-source nature also presented cost savings by eliminating hosting fees. We shared these findings in a demo to the engineering organization, emphasizing both the performance improvements and the potential reduction in CI resource usage. To validate the developer experience, we invited several front-end engineers to write Playwright tests and share feedback. Their responses were overwhelmingly positive, further convincing engineering leadership to support the migration.

2. Migration Tooling

Our next focus is migration tooling, because moving large amounts of tests to a new framework is time consuming and complicated. Cypress and Playwright differ substantially in syntax, selector strategies, and architectural patterns, which makes direct translation challenging and prone to subtle errors. We evaluated several migration methods, starting with manual rewriting, which provided high accuracy but was too slow to be practical for hundreds of tests. We also experimented with AI-assisted translation, leveraging GitHub Copilot and ChatGPT to generate Playwright equivalents from Cypress examples. While this approach accelerated the process, it still required manual adjustments to account for framework-specific behaviors—particularly replacing Cypress's implicit assertions with Playwright's explicit expectations.

Ultimately, we adopted Cursor.io, a specialized migration tool that outperformed other options by parsing page structures and auto-generating Playwright code directly from existing Cypress

files. Cursor.io proved especially effective at importing selectors and producing robust test files, though it initially struggled with the complex, custom Cypress utilities our team had developed. Since these utilities had no direct Playwright equivalents, we improved the tool's accuracy by providing targeted training examples and iteratively refining its output.

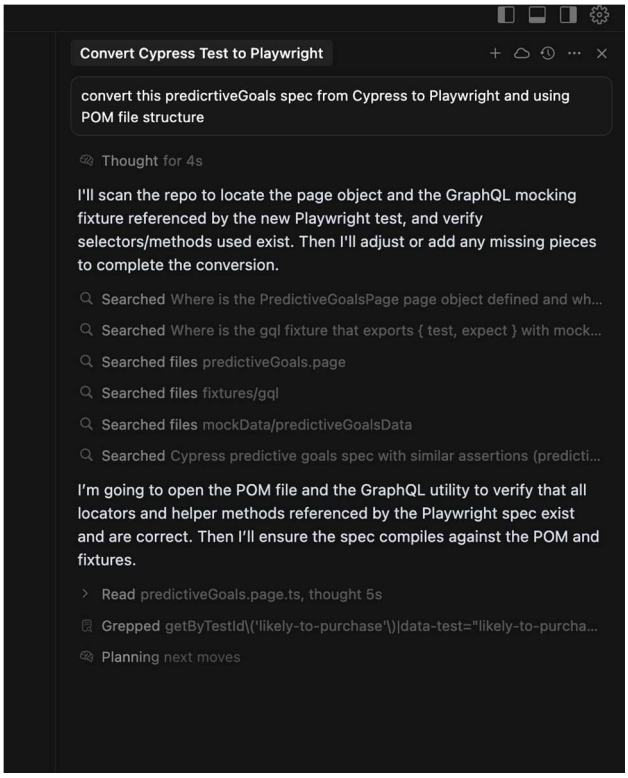

Figure 2 Ex of using Cursor.io

Notes:

- The POM `PredictiveGoalsPage` provides all locators used by the spec (`full*` cards, `andQueryLocator`, `occurrencesInput`, popover, etc.). Methods `getEventText`, `getPurchaseEventText`, `getComparisonOperator`, and `getTimeText` line up with assertions.
- The test uses the `gql` fixture's `mockGQL` correctly with the existing mock JSONs in `playwright/mockData/predictiveGoalsData`.

predictiveGoals.page.ts

playwright > pages > insights > predictiveGoals.page.ts > ...

```ts
import { Page, Locator, expect } from '@playwright/test';

export class PredictiveGoalsPage {
  readonly page: Page;

  // Template cards
  readonly fullSubscribeTemplateCard: Locator;
  readonly fullChurnTemplateCard: Locator;
  readonly fullPurchaseTemplateCard: Locator;
  readonly subscribeTemplateCard: Locator;
  readonly churnTemplateCard: Locator;
  readonly purchaseTemplateCard: Locator;
```

predictiveGoals.spec.ts M

playwright > e2e > insights > predictiveGoals.spec.ts > ...

```ts
import { test, expect } from '../../fixtures/gql';
import { PredictiveGoalsPage } from '../../pages/insights/predictiveGoals.page';

```

3. Folder & Test Structure Improvements

Migrating frameworks gave us the perfect opportunity to eliminate technical debt and design a cleaner, more scalable test architecture. In our Cypress implementation, selectors were often hardcoded directly in test files, reusable commands were scattered or duplicated, and updating a single selector could require changes across multiple files—driving up maintenance costs. For Playwright, we adopted the Page Object Model (POM), consolidating selectors and reusable methods into dedicated page classes. This structure centralized selectors by page or component, encapsulated common workflows in utility functions, and maintained a clear separation between test logic and page interaction details. As a result, duplication was significantly reduced, maintainability improved, and scaling the test suite became far more manageable. Now, updating a selector requires changing only one file rather than dozens, lowering the risk of regressions and easing developer frustration.

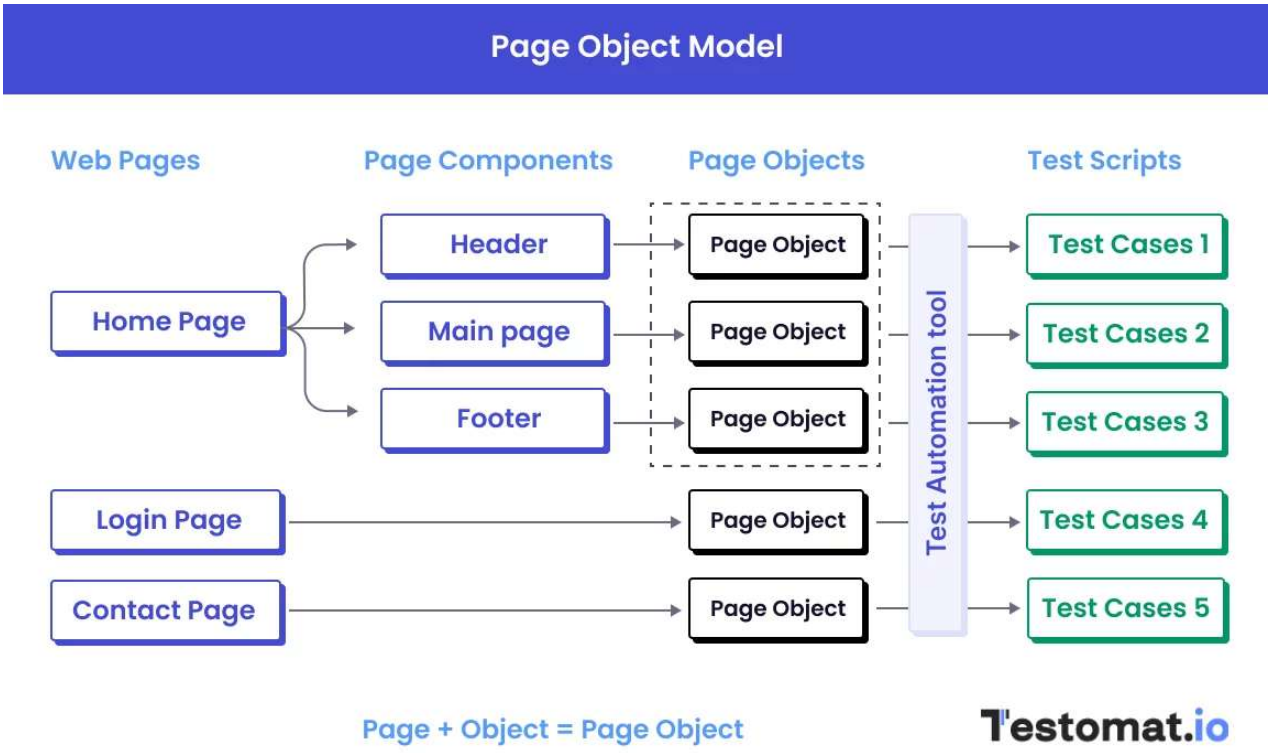

Figure 3 Example of Page Object Model

4. Migration Timeline & Strategy

A successful migration requires phased execution and proactive risk management. Our Playwright migration began in Q3 2024 with over ten specs migrated—specifically those that had been problematic in Cypress. Tackling the most challenging tests first gave us confidence that Playwright could handle the scenarios where Cypress struggled. During this phase, we also completed draft knowledge transfer documentation and saw the first frontend developers contribute Playwright tests. Once a test was migrated, it was skipped in the Cypress suite to avoid duplication, and by this point, all new end-to-end development had shifted to Playwright with no new Cypress tests being created.

In Q4 2024, we migrated another ten-plus specs, focusing on those that were particularly flaky in Cypress so we could compare their stability in Playwright. By Q1 2025, we had completed migrating all specs owned by embedded teams, meaning the most complex tests were already in Playwright and only stable Cypress tests remained to be moved. Q2 2025 was dedicated to migrating the remaining “other” team-owned specs, but limited QE resources made it challenging to complete the work on schedule.

With our Cypress contract ending in August 2025, we prioritized removing dependency on the Cypress Dashboard by switching to CircleCI’s native parallel test execution. Q3 2025 focused on final cleanup to ensure all remaining Cypress dependencies were removed, and our goal for Q4 2025 is to fully deprecate Cypress and complete the migration.

This phased approach allowed us to validate Playwright's capabilities early, maintain momentum by starting with QE-owned tests, and collaborate effectively with frontend teams on shared cases. Running both frameworks in the CI pipeline throughout the process ensured stability, and adopting CircleCI's test-splitting functionality allowed us to move away from the Cypress Dashboard ahead of schedule. Spanning more than a year, the migration was completed smoothly through structured execution, close collaboration, and strategic use of tooling—without major disruption to delivery timelines.

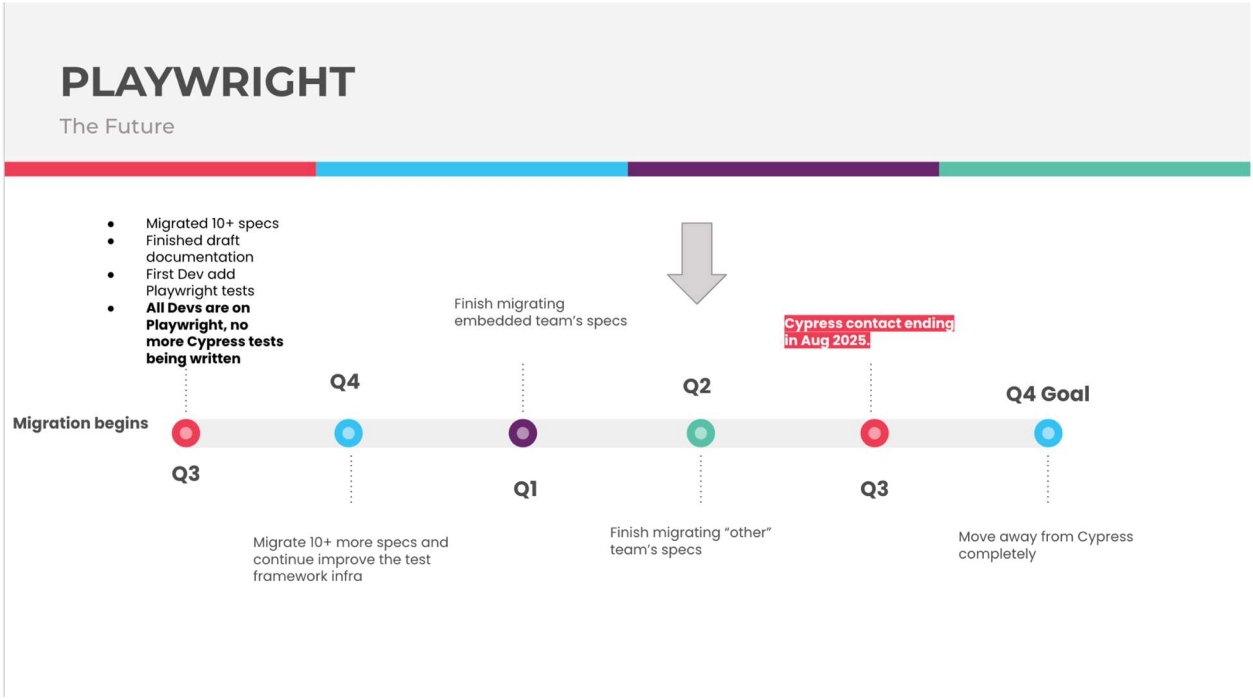

Figure 4 Timeline of the migration

5. Final Touch-Up and Ongoing Support

Migration completion was not the end of the journey—it marked the beginning of long-term adoption. To ensure Playwright became an integral part of our workflow, we prioritized enablement, stability, and continuous feedback. We created comprehensive documentation covering syntax, best practices, advanced scenarios, and troubleshooting guides to help engineers ramp up quickly. To improve reliability, frontend engineers were encouraged to add data-test IDs to elements, significantly reducing test flakiness. We also developed reusable utilities for complex actions such as file uploads, multi-tab workflows, and conditional waits, enabling teams to write cleaner, more maintainable tests.

To demonstrate the value of the migration, we tracked key metrics including run times, flakiness rates, CI resource usage, and total cost savings. This data provided clear evidence of efficiency gains and return on investment for engineering leadership. An open feedback channel allowed us to refine utilities and documentation based on real-world usage, ensuring Playwright became not just a replacement for Cypress, but a sustainable and scalable foundation for our test automation strategy.

SPEC FILES	TOTAL RUNS	FAILURE RATE
195 specs	61	5%
195 specs	604	8%
195 specs	481	11%
195 specs	535	11%
195 specs	531	14%
195 specs	411	13%
195 specs	603	15%
196 specs	515	8%
188 specs	515	21%
188 specs	516	16%

Figure 5 Flaky test percentage when running only running Cypress

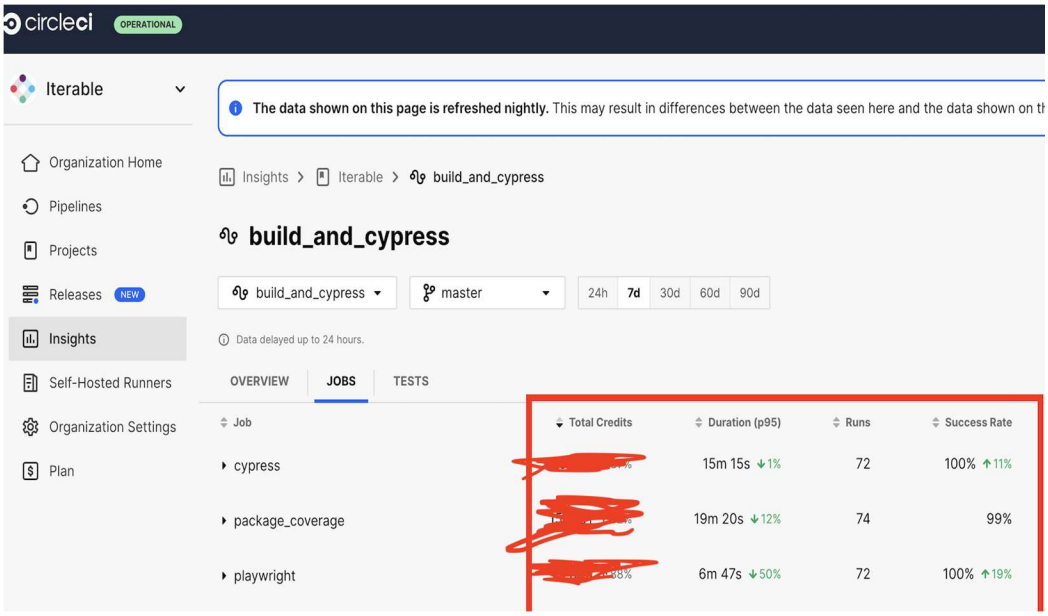

Figure 6 After migrating the flaky tests to Playwright, both jobs are running 100% success rate on the master branch

Highlight of the comparison

Cypress offers strong debugging capabilities through its interactive Test Runner, which includes time-travel debugging and detailed DOM snapshots. The paid version also provides enhanced customer support, making it appealing for teams that value responsive assistance. However, Cypress can face performance bottlenecks at scale, particularly with large test suites or memory-intensive tests. Additionally, advanced features such as test recording, parallelization, and analytics are locked behind the paid Cypress Dashboard, adding ongoing costs.

Playwright, on the other hand, is fully open-source with no paid tiers, providing all core features, including parallelization and reporting, at no additional cost. Its powerful API supports network interception, native event handling, file uploads, geolocation simulation, and more, while also delivering better performance at scale with efficient parallel execution and lower memory overhead compared to Cypress. That said, Playwright offers fewer built-in debugging tools, lacking Cypress's time-travel DOM snapshots, though its trace viewer provides an alternative. It is also less opinionated, requiring teams to make more decisions about test structure, best practices, and configuration, which can lengthen the onboarding process for new users.

Conclusion

Our migration from Cypress to Playwright was more than a framework switch—it was a transformation in how we approach end-to-end testing. By starting with a proof-of-concept,

leveraging migration tooling, and refactoring our test architecture, we built a foundation that is faster, more stable, and easier to maintain. Careful planning and phased execution allowed us to manage risks, handle staffing changes, and avoid disruptions to delivery timelines, while continuous documentation, enablement, and feedback ensured that Playwright adoption was both smooth and sustainable.

The results speak for themselves: reduced flakiness, faster execution times, improved maintainability, and tangible cost savings in CI resources. More importantly, the migration set the stage for long-term scalability, enabling our quality engineering team to keep pace with product growth and evolving business needs. This initiative was not simply about replacing a tool—it was about investing in a testing culture that prioritizes reliability, performance, and adaptability, ensuring our organization remains well-positioned for the future.

Reference

1. Cypress Documentation – Why Cypress? https://docs.cypress.io/app/get-started/why-cypress

2. Playwright Documentation – Getting Started with Playwright https://playwright.dev/docs/intro

3. CircleCI Documentation – Test Splitting and Parallelism https://circleci.com/docs/parallelism-faster-jobs

4. Testomat.io – Page Object Model Pattern in JavaScript with Playwright https://testomat.io/blog/page-object-model-pattern-javascript-with-playwright/