13

Browser Test Automation

Playwright, Patterns, and the Death of Fragile Tests

THE MODERN QA ENGINEER SKILLSET

Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

From First Locator to Production Framework

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

From First Locator to Production Framework

Book 13 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

Edition: First Edition, February 2026

About This Book

This book exists because the browser automation landscape has fundamentally changed, and most books have not caught up.

For over a decade, Selenium was browser automation. If you wanted to automate a web browser, you learned Selenium. You learned WebDriver. You learned the Page Object Model from Selenium's documentation, you fought with implicit and explicit waits, you cursed at StaleElementReferenceException, and you accepted that a 30% flake rate was just how things worked.

That era is over.

Playwright has become the default choice for new browser automation projects. Not because of marketing. Not because it is newer. Because it solved the fundamental architectural problems that made Selenium tests fragile. Auto-waiting that actually works. Browser contexts that provide real isolation. A single installation command that handles browser binaries. Trace files that let you debug failures as if you were watching them happen in real time.

This book is not a Playwright tutorial. Tutorials teach you syntax. This book teaches you how to think about browser test automation in 2026 — and then gives you the tools, patterns, and production-ready code to implement that thinking.

If you have Selenium experience, that experience is valuable. The mental models you built — element location, synchronization, page abstrac-

tion — all transfer. But the implementation changes dramatically, and this book will show you exactly where and why. Every chapter includes "Old Way vs New Way" comparisons that respect your existing knowledge while showing you the better path.

If you are starting fresh, even better. You will learn the right patterns from day one, without habits to unlearn.

By the end of this book, you will have built a complete, production-grade test automation framework. Not a toy project. A real framework with parallel execution, CI/CD integration, visual regression testing, accessibility checks, and the kind of reporting that makes your entire team trust the test suite.

This book will make you dangerous in the best possible way.

About the Modern QA Course Series

This book is part of the Modern QA Course — a comprehensive series covering every aspect of quality engineering in 2026. Each book is self-contained but designed to complement the others.

Book 13 focuses exclusively on browser test automation: the tools, patterns, and architecture for testing web applications through the browser. It deliberately does not cover:

- **API testing in depth** (see Book 4: API Testing)
- **Mobile testing** (see Book 9: Mobile Testing)
- **Advanced visual and accessibility testing** (see Book 10: Visual & Accessibility Testing)
- **Programming fundamentals** (see Book 12: Programming for QA Engineers)
- **AI agent-driven testing** (see Book 3: Agentic Testing)

When these topics intersect with browser automation — like using API calls to set up test data, or running accessibility checks within Playwright — this book covers the intersection. For deep dives, it points you to the right companion book.

Who This Book Is For

This book is written for:

- **Junior QA engineers** learning browser automation for the first time who want to start with modern tools and patterns instead of legacy approaches.
- **Manual testers transitioning to automation** who need a structured, step-by-step path from zero to a working framework with career-relevant skills.
- **Mid-level automation engineers** currently using Selenium who want to understand why their tests are flaky, how to migrate to Playwright, and how to build frameworks that scale.
- **Senior engineers and leads** making tooling decisions who need an honest, technical comparison of Playwright, Cypress, and Selenium based on architecture, not blog posts.
- **Interview candidates** preparing for QA automation interviews where framework design, flakiness debugging, and tool selection are standard topics.

Prerequisites

To get the most from this book, you should have:

- **Basic TypeScript or JavaScript knowledge**. You do not need to be an expert. If you can write a function, use async/await, and understand objects and arrays, you have enough. All primary examples use TypeScript.
- **Basic Python knowledge** (optional). Secondary examples are provided in Python for teams in Python ecosystems. If you only know Python, you can still follow along — Playwright's Python API mirrors the TypeScript API closely.
- **Familiarity with HTML and CSS**. You need to understand what a DOM element is, what a CSS selector does, and what an HTML attribute looks like.
- **Node.js installed on your machine**. Version 18 or higher. If you do not have it, Chapter 3 walks you through setup.

- **A code editor**. VS Code is strongly recommended because Playwright's VS Code extension is excellent. But any editor works.
- **Basic command-line comfort**. You should be able to navigate directories, run commands, and install packages.

You do not need prior Playwright experience. You do not need prior Selenium experience. You do not need prior test automation experience. This book starts from the ground up.

How to Use This Book

This book is organized into eight parts, each building on the previous:

Part I: The Landscape sets the stage. Why did the industry shift from Selenium to Playwright? What architectural differences make Playwright more reliable? If you are skeptical about switching tools, this part will either convince you or give you a principled framework for staying.

Part II: Playwright Foundations covers installation, configuration, browser contexts, navigation, waiting, and assertions. By the end of this part, you will have Playwright running and understand the core concepts that make it different.

Part III: Locators and Selectors covers locator strategy in depth — the single most important skill in browser automation. User-facing locators, advanced filtering, frames, shadow DOM, and why your CSS selectors are probably wrong.

Part IV: Design Patterns covers the Page Object Model (done right), fixtures, hooks, and component testing patterns. This is where test code becomes maintainable.

Part V: The Hard Problems tackles network interception, visual testing, accessibility, file uploads, authentication flows, and the edge cases that break naive frameworks.

Part VI: Framework Architecture is the workshop section. You build a complete production framework from scratch, integrate it with CI/CD, and add reporting and debugging infrastructure.

Part VII: The Ecosystem covers Selenium in 2026 (when you still need it and how to migrate) and provides an honest comparison of Playwright, Cypress, and Selenium.

Part VIII: Capstone and Beyond brings everything together with a full e-commerce test suite project and a look at the future of browser testing.

Each chapter includes:

- **Learning objectives** at the top
- **Complete, runnable code examples** in TypeScript (primary) and Python (secondary)
- **"Common Mistakes" callout boxes** highlighting errors that waste hours
- **"Pro Tips" callout boxes** with expert-level advice
- **Hands-on exercises** with difficulty ratings
- **Self-assessment quizzes** to check your understanding
- **Key takeaways** summarizing essential points
- **Career Translation** sections with resume, cover letter, and interview phrasing
- **Interview Depth Check** sections with architect-level questions and model answers

My recommendation: If you are new to browser automation, read straight through. If you are migrating from Selenium, start with Part I for context, then jump to Part VII Chapter 18 for migration strategy, then come back to Part II to learn Playwright properly. If you are building a framework right now, Parts IV and VI are your priority.

Table of Contents

continued on next page

Part	Chapters	Focus
V: The Hard Problems	12–14	Network, visual testing, edge cases
VI: Framework Architecture	15–17	Production framework, CI/CD, reporting
VII: The Ecosystem	18–19	Selenium migration, tool comparison
VIII: Capstone and Beyond	20–21	Full project, future of testing
Appendices	A–E	Reference, cheat sheets, glossary

PART 0

QUICK START

This part is a fast, opinionated path from "I have no Playwright repo" to "I have an initial, application-specific Playwright + TypeScript automation harness scaffolded by Claude Code." It exists for readers who want to start building immediately and learn the deeper theory in Parts I through VIII afterward.

It is split into three chapters:

- **Chapter 0.1** — discover the application and produce the customization artifacts a scaffolding step needs.
- **Chapter 0.2** — turn those artifacts into a clean, scaffolding-ready package.
- **Chapter 0.3** — the actual copy-paste template you feed into Claude Code along with the artifacts.

If you are brand new to Playwright or to test architecture, you can skip this part on a first read and start at Chapter 1. If you already know roughly what you want and just need a strong starting repo, this part is the shortest path there.

CHAPTER 0.1

Pre-Requisites — Discover the Application and Produce Customization Artifacts

---

Why this chapter exists

Do **not** start a real Playwright automation repo from a generic harness template alone.

A generic template is useful only as a starting architecture. Before Claude Code creates folders, files, agents, page objects, API clients, and starter tests, it needs **application-specific truth**. That truth must be written into a small set of predictable artifacts.

This chapter is the **discovery and customization phase**.

Its job is to turn:

```
generic harness principles
```

into:

```
application-specific scaffolding inputs
```

Only after this chapter is complete should you move to the scaffolding chapter.

Set up Claude Code for discovery

Before you explore the application, configure Claude Code so it works safely and efficiently during the discovery phase.

Start in Plan Mode

Discovery is a read-only activity. You are learning, not building. Start Claude Code in **Plan Mode** so it can read files, run shell commands, and explore — but cannot edit your source code:

```
claude --permission-mode plan
```

You can also press `Shift+Tab` twice during an existing session to switch into Plan Mode. The status bar shows ⏸ `plan mode on` when active.

Plan Mode is ideal here because:

- Claude can navigate the running application, read its source, and explore routes without accidentally creating files
- you stay in control: Claude proposes a plan, and you approve before any changes happen
- if discovery spans multiple sessions, you can resume with `claude --continue`

Name the session

Give this session a descriptive name so you can resume it later:

```
claude --permission-mode plan -n discovery
```

Or during the session:

```
/rename discovery
```

If you close the terminal and need to come back, run:

```
claude --resume discovery
```

Let auto memory work for you

Claude Code has **auto memory** enabled by default. As you explore the application and correct Claude's assumptions, it saves notes for itself (build commands, architecture observations, naming patterns) in `~/.claude/projects/<project>/memory/`. These notes persist across sessions.

You do not need to configure this. Just be aware that Claude is already learning from your corrections during discovery. If it misnames a feature area and you correct it, it will remember the correction next session.

End state of this chapter

By the end of this chapter, you must have these files:

```
docs/specs/product-under-test.md
docs/specs/ui-journeys.md
docs/specs/api-endpoints.md
playwright-scaffolding.md
```

These files are not optional. They are the minimum durable handoff from discovery into scaffolding.

Core rule

> The application is the source of truth.
> The generic template is only the starting shape.

If the application structure conflicts with the generic template, keep the architectural principles, but replace the placeholders with the application's real structure.

Discovery priority order

When learning a product, use the richest source you have. In most cases, use this order:

1. Site Map page
2. Header navigation

3. Footer links
4. Authenticated navigation after login
5. Product docs, help center, onboarding flows
6. API docs, Postman collections, OpenAPI specs
7. Frontend route config or source code, if accessible
8. Exploratory traversal of the running app

You do not need every source. Use the best available sources and combine them where helpful.

What Claude Code should learn during discovery

Before any repo is scaffolded, Claude Code should extract:

- real page names
- route names or route patterns
- feature areas
- user roles
- auth/session model
- critical business flows
- API domain groupings
- endpoint families
- main data entities
- smoke-critical journeys
- risky or brittle areas worth prioritizing

This is what turns a generic harness into a project-specific one.

Step 1 — Create a discovery workspace

Before exploring the application, create the following folders locally in your future repo or in a scratch project folder:

```
docs/specs/
docs/discovery/
```

The `docs/discovery/` folder is temporary or semi-temporary working space.
The `docs/specs/` folder is durable project memory.

Suggested discovery files:

```
docs/discovery/site-map-notes.md
docs/discovery/navigation-notes.md
docs/discovery/api-notes.md
docs/discovery/auth-notes.md
docs/discovery/open-questions.md
```

You may later merge or archive these, but during discovery they help keep findings explicit.

Step 2 — Inspect the Site Map, if one exists

If the application has a visible **Site Map** page, start there. This is often the fastest way to capture the product's real information architecture.

A Site Map page may appear as:

- Site Map
- All Pages
- Browse All
- Documentation Index
- Footer link directory
- Admin navigation tree
- Help center directory

What to extract from the Site Map

Capture:

- page titles
- hierarchy
- grouping by section
- naming conventions
- likely user-facing vs admin-facing areas
- repeated patterns such as `/account/*`, `/settings/*`, `/orders/*`

Copy-paste prompt for Claude Code

```
Traverse the Site Map page and extract the real application structure.

Capture:
- all visible pages
- page hierarchy
- section names
- naming conventions
- route patterns if visible

Group the pages into logical product domains.

Write findings into:
- docs/discovery/site-map-notes.md
- docs/specs/product-under-test.md

Do not scaffold code yet.
```

What to do manually if Claude Code misses context

Review the generated notes and correct:

- duplicate pages
- public vs authenticated distinctions
- pages that are present but not test-worthy
- pages that exist only as support/legal content
- internal/admin areas that should be isolated into separate domains

Step 3 — If there is no Site Map, traverse the app deliberately

If no Site Map exists, discover structure from navigation and exploratory traversal.

Traverse these areas in order

1. landing page
2. primary nav
3. footer nav
4. auth entry points

5. post-login landing page
6. settings/account/admin menus
7. search/browse/detail pages
8. transaction flows
9. error or empty states that appear during use

Capture as you go

For each page or route, note:

- page name
- route or URL pattern
- purpose
- domain/feature area
- whether it belongs in smoke, regression, or low-priority coverage
- whether it is public, authenticated, or role-restricted

Copy-paste prompt for Claude Code

```
Explore the running application and infer its real structure.

From navigation, pages, and visible flows, extract:
- top-level pages
- nested pages
- route patterns
- major user journeys
- public vs authenticated areas
- likely role-specific areas

Write structured notes into:
- docs/discovery/navigation-notes.md
- docs/specs/product-under-test.md

Do not scaffold code yet.
```

Step 4 — Discover authentication and session behavior

A Playwright harness becomes much better when auth behavior is understood before scaffolding.

Determine the auth model

Capture:

- login page or entry point
- auth type: session cookies, token, OAuth, SSO, magic link, MFA, etc.
- logout mechanism
- session persistence expectations
- whether `storageState` is likely to help
- whether API and UI share auth context
- whether different roles need different saved states

Document edge cases

Note anything that affects automation, such as:

- CAPTCHA in lower environments
- MFA bypass in test environments
- login redirects
- region-specific or tenant-specific login pages
- expired sessions causing flake
- rate limits on auth endpoints

Copy-paste prompt for Claude Code

```
Inspect the authentication model of the application.

Document:
- login entry points
- auth type
- session persistence behavior
- likely Playwright storageState strategy
- role differences if visible
- known automation risks

Write findings into:
- docs/discovery/auth-notes.md
- docs/specs/product-under-test.md
```

Step 5 — Discover the UI journeys that actually matter

A good harness does not start by automating random pages. It starts with meaningful journeys.

Identify at least these categories

Smoke journeys

Short, critical checks that prove the application is alive.

Examples:

- sign in
- load dashboard
- browse catalog
- open detail page
- complete checkout
- create a simple record
- fetch a critical account page

Regression journeys

Broader user behavior and business logic.

Examples:

- edit profile
- update payment method
- create/edit/delete flow
- permission checks
- filtering/sorting/search
- validation and error handling

Accessibility targets

Pages worth early a11y coverage.

Examples:

- sign-in form
- checkout flow
- key dashboard
- search results

- important modal/dialog workflows

What to avoid at this stage

Do not over-specify every test case yet.
This chapter is for **journey mapping**, not full suite design.

Copy-paste prompt for Claude Code

```
Based on the discovered product structure, identify the most important UI journeys.

Organize them into:
- smoke journeys
- regression journeys
- accessibility targets

Write the result into:
- docs/specs/ui-journeys.md

Focus on business-critical user flows rather than exhaustive test cases.
```

Step 6 — Discover the API surface

The harness should support API automation from day one, ideally using Playwright `APIRequestContext`.

Learn the API structure

If you have API docs, Postman collections, or OpenAPI, capture:

- endpoint groups
- auth endpoints
- main resources
- create/read/update/delete patterns
- request/response formats
- special headers
- environment differences
- known rate limits
- unstable endpoints or async workflows

Group endpoints into client domains

Avoid dumping raw endpoint lists without structure.

Good grouping examples:

- Auth
- Users
- Orders
- Catalog
- Billing
- Reporting
- Admin

Copy-paste prompt for Claude Code

```
Discover the API structure for the application.

Capture:
- endpoint groups
- auth model
- key resources
- common request patterns
- important response contracts
- risks and special behaviors

Organize the output by domain and write it into:
- docs/specs/api-endpoints.md

Assume the future harness will use Playwright APIRequestContext unless there is a strong reason not to.
```

Step 7 — Write `docs/specs/product-under-test.md`

This file is the highest-level description of the system under test.

Required sections

Use this exact structure as a starting point (or update environments names if internally company calls them differently):

```
# Product Under Test

## Overview
Short description of the product.

## Environments
- local:
- qa:
```

```
- staging:
- production:

## User Roles
- guest
- authenticated user
- admin
- support
- other app-specific roles

## Feature Areas
- Authentication
- Search
- Catalog
- Checkout
- Account
- Reporting
- Admin
- app-specific modules

## Routing / URL Patterns
List important routes or patterns.

## Authentication Model
Describe login, session model, saved state strategy, and automation concerns.

## Major Risks for Automation
List risk areas that can cause flakiness or complexity.

## Notes
Anything else important for future agents.
```

Writing guidance

This file should be concise, but not vague.
Write for a future agent who lands in the repo with no chat history.

Step 8 — Write `docs/specs/ui-journeys.md`

This file defines where early UI automation should go. Talk to your product owner/product manager. Capture everything they respond to the question “Which pieces of functionality, if broken, would stop the release to Prod?” and put it under “Regression Journeys” section.

Recommended structure

```
# UI Journeys

## Smoke Journeys
- login
- open primary landing page
- complete business-critical short flow

## Regression Journeys
- account/settings behavior
- create/edit/delete flows
- validation scenarios
- permission boundaries

## Accessibility Targets
- forms
- modals
- key landing pages
- critical transaction flows

## Priority Notes
Explain which journeys matter most and why.
```

Writing guidance

Use product language, not generic language.

Bad:

- “dashboard stuff”
- “main flow”

Better:

- “Customer checkout with saved address”
- “Admin creates a new coupon”
- “Support user searches for an order and opens details”

Step 9 — Write `docs/specs/api-endpoints.md`

This file tells the scaffolding step how API clients should be organized.

Recommended structure

```
# API Endpoints

## Auth
- POST /login
- POST /logout
- refresh/session endpoints

## <Domain Name>
- GET /resource
- POST /resource
- PUT /resource/:id
- DELETE /resource/:id

## Contracts and Validation Notes
- required headers
- schema concerns
- pagination
- async processing
- rate limiting
```

Writing guidance

Do not try to document every single low-value endpoint. Capture the domain structure that will shape the harness.

Step 10 — Generate `playwright-scaffolding.md`

This is the most important artifact in the chapter.

`playwright-scaffolding.md` is the **custom blueprint** Claude Code will use in the next chapter to scaffold the repo.

It should be derived from:

- the generic harness principles
- `product-under-test.md`
- `ui-journeys.md`
- `api-endpoints.md`

What `playwright-scaffolding.md` must contain

At minimum, it should specify:

- real page-object names
- real feature domains
- real API client groupings
- fixture strategy
- likely flow helpers
- likely reusable component objects
- likely smoke/regression/a11y grouping
- auth strategy
- expected environments
- early plan and agent workflow expectations
- rules like “selectors live in page objects”
- rules like “use Playwright APIRequestContext for API clients”

Copy-paste prompt for Claude Code

```
Create playwright-scaffolding.md by reconciling the generic Playwright harness architecture with the real application structure
↪  captured in:

- docs/specs/product-under-test.md
- docs/specs/ui-journeys.md
- docs/specs/api-endpoints.md

Rules:
- replace generic names with real application terminology
- keep the harness architecture, but customize folder and file naming to the product
- define likely page objects, API clients, flows, reusable components, fixtures, and starter test groupings
- encode agent workflow expectations for planner, generator, and evaluator
- assume Playwright + TypeScript for UI and API automation
- assume Playwright APIRequestContext for API tests unless the app clearly requires otherwise
- assume selectors should live inside the owning page objects

Write the result to:
- playwright-scaffolding.md
```

Step 11 — Validate the artifacts before moving on

Before you continue to the next chapter, review the artifacts.

Checklist

- `product-under-test.md` names real feature areas
- `ui-journeys.md` uses real business flows
- `api-endpoints.md` is grouped by meaningful client domains
- `playwright-scaffolding.md` uses real app terminology
- generic names were replaced where real names are known
- auth/session assumptions are explicit
- the future harness shape is clear enough to scaffold

If this is not true, fix the artifacts before moving on.

What not to do in this chapter

Do not:

- generate code yet
- create page objects yet
- create API clients yet
- write detailed test cases yet
- keep generic names when the app provides better ones
- rely on chat memory instead of writing artifacts to disk

This is a discovery and customization chapter, not an implementation chapter.

Deliverables from Chapter 0.1

You are ready for the next chapter only when these files exist and are usable:

```
docs/specs/product-under-test.md
docs/specs/ui-journeys.md
docs/specs/api-endpoints.md
playwright-scaffolding.md
```

Mental model

```
generic harness architecture
        +
real product structure
        =
custom blueprint for scaffolding
```

That custom blueprint is `playwright-scaffolding.md`.

CHAPTER 0.2

Quick Start Preparation — Turn Discovery into a Scaffolding-Ready Package

---

Why this chapter exists

Chapter 1 produced the raw customization artifacts.

This chapter prepares them for deterministic scaffolding.

The goal here is not to build the repo yet. The goal is to make sure Claude Code can take the discovery output and generate a strong **initial repo state** with:

- structure
- standards
- plans
- agent roles
- starter fixtures
- starter tests
- validation scripts
- durable documentation

Think of this chapter as the **assembly step** between discovery and generation.

Inputs required from Chapter 0.1

These must already exist:

```
docs/specs/product-under-test.md
docs/specs/ui-journeys.md
docs/specs/api-endpoints.md
playwright-scaffolding.md
```

If they do not exist, stop and complete Chapter 1 first.

The main rule of this chapter

> `playwright-scaffolding.md` is now the application-specific source of truth.

Not the generic harness template.
Not memory.
Not assumptions.

The remaining quick-start process should treat the artifacts from Chapter 0.1 as durable repo memory.

What this chapter prepares Claude Code to do

After this chapter, Claude Code should be ready to scaffold:

- root repo files
- Playwright config
- docs structure
- plans and templates
- `CLAUDE.md`
- `AGENTS.md`
- agent role docs
- page-object folders
- API-client folders
- fixture folders
- sample UI/API/hybrid tests
- reusable utilities

- reports/artifact folders
- validation scripts
- CI workflow skeletons

But before it does any of that, you should package the instructions cleanly.

Step 1 — Review the four required artifacts together

Open these side by side:

```
docs/specs/product-under-test.md
docs/specs/ui-journeys.md
docs/specs/api-endpoints.md
playwright-scaffolding.md
```

Ask these questions:

- Do the feature areas in `product-under-test.md` match the page and client domains in `playwright-scaffolding.md`?
- Do the journeys in `ui-journeys.md` map cleanly to likely test folders and flow helpers?
- Do the API domains in `api-endpoints.md` map cleanly to likely API client classes?
- Is auth handled consistently across all four files?
- Is the terminology consistent?

Fix mismatches now. Do not expect Claude Code to resolve ambiguous language perfectly.

Step 2 — Normalize naming before scaffolding

The scaffolding phase gets much better when naming is normalized first.

Normalize these categories

Page names

Choose one canonical name per page or screen.

Examples:

- `CheckoutPage`

- `OrderHistoryPage`
- `AccountSettingsPage`

Avoid drift like:

- “Account Page”
- “Settings”
- “Profile screen”
- “User settings page”

if they really mean the same thing.

API client domains

Choose one canonical name per endpoint family.

Examples:

- `AuthClient`
- `OrdersClient`
- `CatalogClient`
- `UsersClient`

Flow names

Choose names that represent business behavior.

Examples:

- `checkout.flow.ts`
- `login.flow.ts`
- `account-recovery.flow.ts`

Component object names

Use these only where repeated UI fragments justify them.

Examples:

- `Header.ts`
- `Sidebar.ts`
- `Toast.ts`
- `CartDrawer.ts`

Why this matters

If names are unstable during scaffolding, the repo will start with drift and future agents will amplify it.

Step 3 — Decide the initial harness boundaries

A good initial repo is not the whole universe. It is a clear starting slice.

Decide what the initial scaffold must include

Usually:

- smoke UI coverage for 1–3 critical journeys
- smoke API coverage for auth and one major domain
- one hybrid test showing API-assisted UI setup
- basic auth strategy
- basic fixtures
- page objects only for the earliest journeys
- component objects only where immediately useful
- reusable API clients for the earliest endpoint groups

Decide what is out of scope for the first scaffold

Usually:

- full regression suite
- complete page-object coverage of the entire app
- every API endpoint
- every role
- visual regression
- contract-testing sophistication beyond simple schema checks
- performance testing
- mobile emulation unless essential

Write these boundaries into your scaffolding expectations.

Step 4 — Decide the initial directory naming

Before Claude Code scaffolds, be explicit about the major top-level areas.

A strong default is:

```
src/
  core/
  ui/
    pages/
    components/
    flows/
    assertions/
  api/
    clients/
    builders/
    helpers/
    schemas/
    assertions/
  shared/
    data/
    factories/
    utils/
    state/
  fixtures/

tests/
  ui/
    smoke/
    regression/
    accessibility/
  api/
    smoke/
    regression/
    contract/
  hybrid/
  setup/

docs/
  architecture/
  standards/
  runbooks/
  specs/
  plans/
  agents/
  decisions/

scripts/
artifacts/
```

```
.github/workflows/
```

This structure is opinionated but practical. It keeps reusable code in `src/`, specs in `tests/`, and durable repo memory in `docs/`.

Step 5 — Decide the opinionated defaults

Before scaffolding, make these rules explicit.

Preferred stack

- Node.js
- TypeScript
- Playwright Test
- npm
- ESLint
- Prettier
- dotenv
- zod

Preferred API harness style

Use Playwright `APIRequestContext` by default for API automation.

Preferred pattern:

- request context per appropriate scope
- domain-specific wrapper clients
- shared auth setup
- schema validation with `zod`
- explicit assertions for status, headers, and important body fields

Preferred UI harness style

- selectors live inside the owning page object
- page objects hold reusable page-specific actions

- flows handle multi-page business behavior
- assertion helpers hold reusable domain expectations
- component objects exist only for reused page fragments

Preferred agent loop

At minimum:

```
Planner → Generator → Evaluator
```

Secondary roles can be added later:

- Reviewer
- Doc-gardener

Note that Playwright itself now ships first-party Test Agents (planner, generator, healer) since v1.56, scaffolded with:

```
npx playwright init-agents --loop=claude
```

This generates agent definitions plus a `specs/` folder for Markdown test plans and a `seed.spec.ts` entry point; generated tests land in `tests/`. They are a drop-in replacement for the plan-and-generate part of this loop, and the healer is worth adopting either way. The custom loop above remains useful when you need project-specific evaluator rules.

Preferred repo-memory style

- short root instruction files
- deep guidance in `docs/`
- plans written to files
- handoffs written to files
- evaluator reports written to files
- future agents should be able to continue from repo state alone

These defaults should be visible in the final template.

Step 6 — Prepare the root instruction strategy

Claude Code reads `CLAUDE.md` at the start of every session. It is the most important file for shaping Claude's behavior, but it must stay **under 200 lines**. Longer files get ignored.

The solution is to keep `CLAUDE.md` short and navigational, and use two complementary mechanisms:

- **`@import` syntax** — `CLAUDE.md` can reference other files with `@path/to/file.md`. Claude expands them automatically at session start.
- **`.claude/rules/`** — path-scoped rules that only load when Claude is working with matching files.

What `CLAUDE.md` should contain

```
# Playwright Automation Harness — [Product Name]

## What this repo is
One-sentence description.

## Source of truth
@playwright-scaffolding.md overrides generic assumptions.

## Key docs
- Product spec: @docs/specs/product-under-test.md
- UI journeys: @docs/specs/ui-journeys.md
- API endpoints: @docs/specs/api-endpoints.md
- Architecture: @docs/architecture/repo-map.md

## Validation
Run before committing:
- `npm run validate` (format, lint, typecheck, smoke)
- `npm run test:smoke`

## Standards
- @docs/standards/coding-standards.md
- @docs/standards/test-writing-standards.md
- @docs/standards/selector-strategy.md

## Work log — IMPORTANT
ALWAYS update CHANGELOG.md before committing. Every entry must include:
- date
- what was done (one-line summary)
- which files were created or changed
```

```
- why the change was made

## Agent workflow
Planner → Generator → Evaluator. See @AGENTS.md.

## Plans
- Active plans: docs/plans/active/
- Completed plans: docs/plans/completed/
```

The `@` syntax makes the referenced files available to Claude without repeating their contents in `CLAUDE.md`. This keeps the root file navigational while giving Claude deep context on demand.

Use `.claude/rules/` for path-scoped rules

Instead of putting every coding rule in `CLAUDE.md`, create rule files in `.claude/rules/` that only apply when Claude is working with matching files:

```
.claude/
└── rules/
    ├── page-objects.md      # rules for src/ui/pages/**
    ├── api-clients.md       # rules for src/api/**
    ├── test-writing.md      # rules for tests/**
    └── selectors.md         # rules for src/ui/**
```

Each rule file uses YAML frontmatter to specify which paths it applies to:

```
---
paths:
  - "src/ui/pages/**/*.ts"
---

# Page Object Rules

- Selectors live inside the owning page object, never in test files
- Page objects expose actions, not raw locators
- Use `getByRole`, `getByText`, or `getByTestId` — avoid CSS selectors
- Every page object must have a `waitForReady()` method
```

```
---
paths:
  - "tests/**/*.ts"
---

# Test Writing Rules

- One assertion concept per test
- Use descriptive test names that read as sentences
- Never use `page.waitForTimeout()` — rely on auto-waiting
- Smoke tests must complete in under 10 seconds
```

Path-scoped rules are powerful because they:

- do not consume context tokens until Claude actually works with matching files
- prevent rule conflicts (UI rules cannot confuse API work)
- scale as the repo grows without bloating `CLAUDE.md`

`AGENTS.md` — the multi-agent operating model

`CLAUDE.md` imports `AGENTS.md` via `@AGENTS.md`. This file defines role responsibilities and handoff rules:

- role responsibilities
- handoff rules
- when the planner must be consulted
- when the evaluator must fail work
- when docs must be updated

Use `.claude/agents/` for native subagents

Claude Code supports **native subagents** — specialized assistants that run in their own context window with their own set of allowed tools. Instead of only documenting agent roles in `docs/agents/`, create actual subagent definitions in `.claude/agents/`:

```
<!-- .claude/agents/evaluator.md -->
---
name: evaluator
description: Verifies implementation against sprint contract. Use after completing a sprint.
tools: Read, Grep, Glob, Bash
---

You are the evaluator. Your job is to verify implementation quality.

1. Read the sprint contract from docs/plans/active/
2. Run all validation commands listed in the contract
3. Check that acceptance criteria are met
4. Write a PASS/FAIL report to docs/plans/handoffs/
5. If FAIL, list specific issues with file paths and line numbers
```

```
<!-- .claude/agents/planner.md -->
---
name: planner
description: Creates sprint contracts and execution plans before implementation work begins.
tools: Read, Grep, Glob
---

You are the planner. Your job is to define scope before coding.

1. Read the request or spec
2. Explore the codebase to understand current state
3. Write a sprint contract to docs/plans/active/
4. The contract must include: scope, out of scope, acceptance criteria, validation commands, expected files to change, risks
```

Native subagents are better than plain documentation because:

- Claude Code delegates tasks to them automatically based on the `description` field
- they run in a **separate context window**, so investigation work does not clutter your main session
- you can scope their tools (e.g., the evaluator can run Bash commands, but a reviewer might only need Read)
- you can request a specific one: `"use the evaluator subagent to verify my last change"`

Keep the role docs in `docs/agents/` as human-readable references, but create the `.claude/agents/` definitions as the actual working subagents.

Use `.claude/skills/` for repeatable workflows

Skills are reusable prompts that Claude applies automatically when relevant, or that you can invoke directly with `/skill-name`:

```
<!-- .claude/skills/new-page-object/SKILL.md -->
---
name: new-page-object
description: Create a new page object for a given page
---

Create a new page object for: $ARGUMENTS

1. Read docs/specs/product-under-test.md for context
2. Read docs/standards/selector-strategy.md for selector rules
3. Look at existing page objects in src/ui/pages/ for patterns
4. Create the page object file in src/ui/pages/
5. Add a `waitForReady()` method
6. Export it from the pages index
7. Run `npm run typecheck` to verify
```

```
<!-- .claude/skills/new-test-spec/SKILL.md -->
---
name: new-test-spec
description: Create a new test specification file
---

Create a test spec for: $ARGUMENTS

1. Read docs/specs/ui-journeys.md for journey context
2. Look at existing tests in the matching test folder for patterns
3. Create the test file following docs/standards/test-writing-standards.md
4. Run the new test with `npx playwright test <file> --reporter=line`
5. Fix any failures
```

```
<!-- .claude/skills/log/SKILL.md -->
---
name: log
description: Update CHANGELOG.md with a summary of recent work
---

Update CHANGELOG.md with what was done in this session.

1. Check git diff and git status for all changes since the last changelog entry
2. Summarize what was accomplished, grouped by feature or concern
3. List all files created or modified with a brief note for each
4. Add the entry at the top of CHANGELOG.md under today's date
5. Stage CHANGELOG.md
```

Invoke skills directly: `/new-page-object CheckoutPage`, `/new-test-spec checkout smoke`, or `/log` after completing work.

Skills differ from `.claude/rules/` in an important way:

- **Rules** load automatically when Claude touches matching files — use them for standards and constraints
- **Skills** load on demand when invoked or when Claude determines they are relevant — use them for workflows and procedures

Step 7 — Prepare the planning system

The first scaffold should not only create code folders. It should create an operating system for future work.

That means Claude Code should create at least:

```
docs/plans/active/
docs/plans/completed/
docs/plans/handoffs/
docs/plans/templates/
```

Required templates

The scaffold should include templates for:

- sprint contract
- execution plan
- evaluator report

What each should contain

Sprint contract

- task name
- linked request or spec
- scope
- out of scope
- acceptance criteria
- validation commands
- expected files to change
- risks and assumptions

Execution plan

- problem statement
- approach
- phases
- progress log
- decisions
- follow-up debt

Evaluator report

- contract reviewed
- commands run
- environment
- pass/fail summary
- evidence
- missing coverage
- recommendation

This makes future long-running agent work much more reliable.

Step 8 — Decide the initial validation gate

The initial repo should be easy to validate frequently.

That means the scaffold should support scripts like:

```
npm run bootstrap
npm run lint
npm run format
npm run format:check
npm run typecheck
npm run test
npm run test:ui
npm run test:api
npm run test:smoke
npm run test:contract
npm run test:a11y
npm run docs:check
npm run validate
npm run report
npm run plan:init
npm run new:spec
```

Recommended meaning of `validate`

Keep it concise but useful.

A good default is:

- format check
- lint
- typecheck
- smoke tests

Do not make the local validation gate so heavy that agents stop using it.

Step 9 — Decide the artifact and debugging conventions

The scaffold should create predictable local outputs.

Use:

```
artifacts/
  reports/
  screenshots/
  videos/
  traces/
  logs/
```

And the future runbooks should explain:

- how to run one test
- how to open the HTML report
- how to inspect traces
- how to regenerate auth state
- how to run only API tests
- how to switch environment

Redaction rule

The scaffold should encode:

- never log raw secrets
- never log full tokens
- redact auth headers
- redact cookies
- redact PII
- redact session values in logs and artifacts where practical

Step 10 — Decide the initial CI expectations

The first scaffold should be CI-aware even if CI is still minimal.

A good default is to prepare:

```
.github/workflows/ci.yml
.github/workflows/smoke.yml
.github/workflows/docs-check.yml
```

Expectations

`ci.yml`

- install dependencies
- lint
- typecheck
- run smoke UI and API tests
- upload Playwright report and failure artifacts when needed

`smoke.yml`

- manual or scheduled smoke run

`docs-check.yml`

- verify required docs exist
- verify key links or file presence where practical

These do not need to be perfect from day one, but the structure should be in place.

Step 11 — Choose the right permission mode for scaffolding

Claude Code has several permission modes that control how often it pauses to ask for approval. The default mode asks before every file edit and every shell command. During scaffolding, that means approving dozens of file creations one by one — tedious and unnecessary.

Recommended: `acceptEdits` mode

For the scaffolding step, start Claude Code in `acceptEdits` mode:

```
claude --permission-mode acceptEdits -n scaffolding
```

In this mode, Claude creates and edits files without prompting. Shell commands (like `npm install`) still require approval. This is the right balance: you let Claude build the file structure uninterrupted, but stay in control of anything that installs packages or modifies system state.

Alternative: `auto` mode

If your account supports it (Team, Enterprise, or API plan), `auto` mode eliminates nearly all prompts. A separate classifier model reviews each action and blocks anything that looks risky:

```
claude --permission-mode auto -n scaffolding
```

Auto mode is ideal for long scaffolding runs where you want to step away and let Claude work.

Alternative: worktree isolation

For extra safety, scaffold in a Git worktree. Claude works in an isolated copy of the repo, and you review the result before merging:

```
claude --worktree scaffolding
```

This creates `.claude/worktrees/scaffolding/` with its own branch. If you do not like the result, the worktree is discarded cleanly.

If the scaffold needs `.env` files from your main repo, create a `.worktreeinclude` file in the project root:

```
.env
.env.local
```

What not to use

Do **not** use `--dangerously-skip-permissions` (bypass mode) unless you are inside an isolated container or VM. It disables all safety checks.

Step 12 — Prepare hooks for automated quality checks

Claude Code supports **hooks** — shell commands that run automatically at specific points in Claude's workflow. Unlike `CLAUDE.md` instructions (which are advisory), hooks are deterministic. If a hook fails, Claude's action is blocked.

Recommended hooks for the scaffold

Add these to `.claude/settings.json` (or ask Claude to write them: `"Write a hook that runs eslint after every file edit"`):

Lint after every file edit

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "npx eslint --fix $CLAUDE_FILE_PATH 2>/dev/null || true"
          }
        ]
      }
    ]
  }
}
```

Notify when Claude needs attention

If you step away during scaffolding, get a desktop notification:

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "osascript -e 'display notification \"Claude Code needs your attention\" with title \"Claude Code\"'"
          }
        ]
      }
    ]
  }
}
```

Block commits without a changelog entry

Claude can update `CHANGELOG.md` when asked, but it does not do it consistently. A `CLAUDE.md` instruction helps (~70–80% of the time), but the only guaranteed enforcement is a hook that blocks the commit if `CHANGELOG.md` is not staged:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash(git commit*)",
        "hooks": [
          {
            "type": "command",
            "command": "git diff --cached --name-only | grep -q 'CHANGELOG.md' || (echo 'BLOCKED: Update CHANGELOG.md before
            ↪  committing. Summarize what was done, why, and which files changed.' && exit 1)"
          }
        ]
      }
    ]
  }
}
```

When Claude tries to commit without updating the changelog, the hook fails with a clear message. Claude sees the error, goes back, updates `CHANGELOG.md`, stages it, and retries the commit. This works in every permission mode, including `auto`.

Pair this hook with the `/log` skill from Step 6 and the `CLAUDE.md` rule below for a three-layer approach:

Layer	Mechanism	Reliability
Hook (PreToolUse on commit)	Blocks the commit — Claude **cannot** skip it	100%
CLAUDE.md rule	Reminds Claude to update the log throughout the session	~70–80%
`/log` skill	Manual trigger when you want an entry without committing	100% (you invoke it)

Why hooks matter for autonomous work

When Claude runs in `acceptEdits` or `auto` mode, it makes many changes without pausing. Hooks ensure that standards are enforced on every edit, even when you are not watching. This is especially important during scaffolding, where dozens of files are created in sequence.

Step 13 — Prepare the exact prompt package for Claude Code

Before moving to Chapter 0.3, prepare the final set of inputs Claude Code will read.

These are:

```
playwright-scaffolding.md
docs/specs/product-under-test.md
docs/specs/ui-journeys.md
docs/specs/api-endpoints.md
```

Optionally, you may also include:

```
docs/discovery/site-map-notes.md
docs/discovery/navigation-notes.md
docs/discovery/api-notes.md
docs/discovery/auth-notes.md
docs/discovery/open-questions.md
```

These optional files are useful when the app has complexity or ambiguity.

Step 14 — Final pre-scaffolding checklist

Do not move to Chapter 0.3 until all of the following are true:

- the application's real structure is captured
- names are normalized
- auth/session behavior is explicit
- journey priorities are explicit
- API client domains are explicit
- initial scope is bounded
- opinionated defaults are explicit
- the required input files exist
- permission mode decision is made (`acceptEdits` or `auto`)
- hook strategy is decided (at minimum: lint-on-edit, changelog-on-commit, notifications)

- work log strategy is decided (CLAUDE.md rule + commit hook + `/log` skill)
- `.claude/rules/` path-scoped rules are planned
- `.claude/agents/` subagent definitions are planned
- you are ready for Claude Code to scaffold from those files

What not to do in this chapter

Do not:

- write the actual scaffold prompt yet
- start implementation yet
- create placeholder-heavy files with no real product content
- let generic names survive when real names are known
- add every possible future framework to the planned scaffold

This chapter is about making the handoff to scaffolding clean and deterministic.

Deliverable from Chapter 0.2

You should now have:

- the four required source artifacts from Chapter 0.1
- a clear set of opinionated harness defaults
- a `CLAUDE.md` strategy that uses `@imports` and stays under 200 lines
- `.claude/rules/` path-scoped rules planned for UI, API, and test code
- `.claude/agents/` native subagent definitions planned (planner, generator, evaluator)
- `.claude/skills/` repeatable workflow definitions planned
- a hook strategy for automated quality checks and work log enforcement
- a work log strategy (CLAUDE.md rule + commit hook + `/log` skill)
- a permission mode decision (`acceptEdits` or `auto`) for uninterrupted scaffolding
- a clear plan/template strategy

- a clear validation and CI expectation
- a clear understanding of what the initial scaffold must include

Now Claude Code can scaffold an initial repo state without guessing too much.

CHAPTER 0.3

Copy-Paste Template — Feed This and the Customization Artifacts into Claude Code

---

Purpose

This chapter is the **actual scaffolding template**.

It expects the application-specific artifacts from Chapters 0.1 and 0.2 to already exist.

You will copy-paste this template into Claude Code together with the required artifacts. Claude Code should then create an initial Playwright + TypeScript automation repo that is:

- customized to the application
- agent-ready
- planning-heavy
- validation-oriented
- ready to begin building test cases, page objects, reusable components, API clients, fixtures, and starter tests

This is not a discovery prompt.
Discovery is already done.
This is a **repo initialization prompt**.

Required inputs

Before using this template, all of these must already exist:

```
playwright-scaffolding.md
docs/specs/product-under-test.md
docs/specs/ui-journeys.md
docs/specs/api-endpoints.md
```

Optional but useful:

```
docs/discovery/site-map-notes.md
docs/discovery/navigation-notes.md
docs/discovery/api-notes.md
docs/discovery/auth-notes.md
docs/discovery/open-questions.md
```

Non-negotiable rule

> If `playwright-scaffolding.md` exists, it overrides generic assumptions.

Use real application structure.
Use real application terminology.
Do not keep generic placeholders where the product-specific artifacts provide better names.

Before you paste the template

Choose a permission mode

The scaffolding step creates dozens of files. In the default permission mode, Claude will pause and ask you to approve every single file creation. That defeats the purpose of autonomous scaffolding.

Recommended: `acceptEdits` mode. Claude creates and edits files freely. Shell commands still require approval.

```
claude --permission-mode acceptEdits -n scaffolding
```

Alternative: `auto` mode (requires Team, Enterprise, or API plan). A classifier model reviews every action. You can step away and let Claude work.

```
claude --permission-mode auto -n scaffolding
```

Alternative: worktree isolation. Scaffold in a disposable copy of the repo. Review the result, then merge or discard.

```
claude --worktree scaffolding --permission-mode acceptEdits
```

Name the session

Always name the scaffolding session so you can resume it if interrupted:

```
claude -n scaffolding --permission-mode acceptEdits
```

If Claude is interrupted mid-scaffold, resume with:

```
claude --resume scaffolding
```

Set up hooks (optional but recommended)

If you prepared hooks in Chapter 0.2 Step 12, confirm they are in `.claude/settings.json` before starting. At minimum, a lint-on-edit hook keeps the generated code clean as it is created.

Copy-paste template for Claude Code

```
You are scaffolding an initial Playwright + TypeScript UI and API automation harness for mostly autonomous agentic development.

Read these files first and treat them as the source of truth:

Required:
- playwright-scaffolding.md
- docs/specs/product-under-test.md
- docs/specs/ui-journeys.md
- docs/specs/api-endpoints.md

Optional, if present:
- docs/discovery/site-map-notes.md
- docs/discovery/navigation-notes.md
- docs/discovery/api-notes.md
- docs/discovery/auth-notes.md
- docs/discovery/open-questions.md

Your job is to create an initial repository state that is customized to the real application under test.

Important rules:
- If playwright-scaffolding.md exists, it overrides generic assumptions.
- Use real page names, real route groupings, real feature domains, real API client domains, and real journey names.
- Keep architectural principles from the generic harness, but customize naming and structure to the actual application.
- Do not preserve generic placeholder names if the artifacts provide real names.
- Prefer a repo that another agent can resume from after context loss.

Tech and tooling defaults:
```

```
- Node.js
- TypeScript
- Playwright Test
- npm
- ESLint
- Prettier
- dotenv
- zod

Primary harness requirements:
- UI automation with Playwright
- API automation with Playwright APIRequestContext by default
- shared auth/session helpers where appropriate
- environment-aware configuration
- local development support
- CI-aware structure
- artifact and debugging support
- strong repo memory via docs and plans

Create the repository with this top-level structure as a strong default, adapting names where the application-specific artifacts
↪  justify it:

.
├── AGENTS.md
├── CHANGELOG.md
├── CLAUDE.md
├── README.md
├── package.json
├── tsconfig.json
├── playwright.config.ts
├── .gitignore
├── .editorconfig
├── .env.example
├── .worktreeinclude
├── .claude/
│   ├── settings.json
│   ├── rules/
│   │   ├── page-objects.md
│   │   ├── api-clients.md
│   │   ├── test-writing.md
│   │   └── selectors.md
│   ├── agents/
│   │   ├── planner.md
│   │   ├── generator.md
│   │   ├── evaluator.md
│   │   └── reviewer.md
│   └── skills/
│       ├── new-page-object/SKILL.md
│       ├── new-test-spec/SKILL.md
│       ├── run-evaluator/SKILL.md
│       └── log/SKILL.md
├── scripts/
│   ├── bootstrap.ts
│   ├── validate.ts
│   ├── smoke-ui.ts
│   ├── smoke-api.ts
│   ├── doc-check.ts
│   ├── plan-init.ts
│   └── new-spec.ts
├── config/
│   ├── env/
│   ├── projects/
│   └── test-data/
├── src/
│   ├── core/
│   ├── ui/
│   │   ├── pages/
│   │   ├── components/
│   │   ├── flows/
│   │   └── assertions/
│   ├── api/
│   │   ├── clients/
│   │   ├── builders/
│   │   ├── helpers/
│   │   ├── schemas/
│   │   └── assertions/
│   ├── shared/
│   │   ├── data/
│   │   ├── factories/
│   │   ├── utils/
│   │   └── state/
│   └── fixtures/
├── tests/
│   ├── ui/
│   │   ├── smoke/
│   │   ├── regression/
│   │   └── accessibility/
│   ├── api/
│   │   ├── smoke/
│   │   ├── regression/
│   │   └── contract/
```

```
│   ├── hybrid/
│   └── setup/
├── artifacts/
│   ├── reports/
│   ├── screenshots/
│   ├── videos/
│   ├── traces/
│   └── logs/
├── docs/
│   ├── index.md
│   ├── architecture/
│   ├── standards/
│   ├── runbooks/
│   ├── specs/
│   ├── plans/
│   │   ├── active/
│   │   ├── completed/
│   │   ├── handoffs/
│   │   └── templates/
│   ├── agents/
│   └── decisions/
└── .github/
    └── workflows/

Create meaningful starter content, not empty placeholders, for these files:
- README.md
- CLAUDE.md
- AGENTS.md
- docs/index.md
- docs/architecture/repo-map.md
- docs/architecture/test-layering.md
- docs/architecture/ui-harness.md
- docs/architecture/api-harness.md
- docs/architecture/data-and-state.md
- docs/standards/coding-standards.md
- docs/standards/test-writing-standards.md
- docs/standards/selector-strategy.md
- docs/standards/api-contract-strategy.md
- docs/standards/logging-and-redaction.md
- docs/standards/flaky-test-policy.md
- docs/runbooks/local-dev.md
- docs/runbooks/ci.md
- docs/runbooks/debugging.md
- docs/runbooks/traces-and-screenshots.md
- docs/runbooks/incident-response-for-failing-tests.md
- docs/agents/planner.md
- docs/agents/generator.md
- docs/agents/evaluator.md
- docs/agents/reviewer.md
- docs/agents/doc-gardener.md
- docs/plans/templates/sprint-contract-template.md
- docs/plans/templates/execution-plan-template.md
- docs/plans/templates/evaluator-report-template.md
- docs/decisions/0001-initial-harness-architecture.md
- .claude/settings.json
- .claude/rules/page-objects.md
- .claude/rules/api-clients.md
- .claude/rules/test-writing.md
- .claude/rules/selectors.md
- .claude/agents/planner.md
- .claude/agents/generator.md
- .claude/agents/evaluator.md
- .claude/agents/reviewer.md
- .claude/skills/new-page-object/SKILL.md
- .claude/skills/new-test-spec/SKILL.md
- .claude/skills/run-evaluator/SKILL.md
- .claude/skills/log/SKILL.md
- .worktreeinclude
- CHANGELOG.md

Root instruction rules:
- CLAUDE.md must be under 200 lines. Keep it navigational — use @path/to/file.md imports to reference detailed docs.
- CLAUDE.md should import key docs: @playwright-scaffolding.md, @docs/specs/product-under-test.md,
↪  @docs/standards/coding-standards.md, @AGENTS.md, etc.
- CLAUDE.md must include a "Work log" section near the top: "ALWAYS update CHANGELOG.md before committing" with the required entry
↪  format (date, summary, files changed, why). This advisory rule is backed by the PreToolUse commit hook.
- AGENTS.md must define the collaboration model and handoff rules. CLAUDE.md imports it via @AGENTS.md.
- Do not create one giant monolithic instruction file.
- Future agents should be able to continue from repo files alone.

Path-scoped rules (.claude/rules/):
Create rule files with YAML frontmatter `paths:` fields so rules only load when Claude works with matching files:
- .claude/rules/page-objects.md (paths: src/ui/pages/**/*.ts) — selector strategy, page object patterns, waitForReady requirement
- .claude/rules/api-clients.md (paths: src/api/**/*.ts) — APIRequestContext usage, schema validation, redaction
- .claude/rules/test-writing.md (paths: tests/**/*.ts) — test naming, assertion style, no hardcoded waits, smoke test time limits
- .claude/rules/selectors.md (paths: src/ui/**/*.ts) — getByRole first, getByTestId second, never CSS selectors for user-facing
↪  elements

Native subagents (.claude/agents/):
Create subagent definitions so Claude can delegate specialized tasks:
- .claude/agents/planner.md — tools: Read, Grep, Glob. Creates sprint contracts before implementation.
```

```
- .claude/agents/generator.md — tools: Read, Grep, Glob, Edit, Write, Bash. Implements the current sprint scope.
- .claude/agents/evaluator.md — tools: Read, Grep, Glob, Bash. Verifies implementation against sprint contract, writes PASS/FAIL
↪  report.
- .claude/agents/reviewer.md — tools: Read, Grep, Glob. Reviews code for quality, patterns, and security.
Each subagent must have a description field that tells Claude when to delegate to it automatically.

Skills (.claude/skills/):
Create reusable workflow definitions:
- .claude/skills/new-page-object/SKILL.md — create a new page object from a page name
- .claude/skills/new-test-spec/SKILL.md — create a new test file from a journey description
- .claude/skills/run-evaluator/SKILL.md — invoke the evaluator against the current sprint contract
- .claude/skills/log/SKILL.md — update CHANGELOG.md with a summary of recent work (invoke with /log)

Hooks (.claude/settings.json):
Configure hooks for automated quality enforcement:
- PostToolUse hook on Edit|Write: run eslint --fix on changed files
- PreToolUse hook on Bash(git commit*): block commits unless CHANGELOG.md is staged — this is the only guaranteed way to enforce
↪  work logging
- Notification hook: notify when Claude needs attention (useful during long scaffolding runs)

Worktree support:
- Create .worktreeinclude listing .env and .env.local so environment files are copied into worktrees automatically
- Add .claude/worktrees/ to .gitignore

Agent workflow requirements:
- Make Planner → Generator → Evaluator the default core loop.
- Planner defines scope and acceptance criteria before coding.
- Generator implements only the current sprint scope.
- Evaluator verifies against the sprint contract and writes PASS/FAIL with evidence.
- Also include Reviewer and Doc-gardener role docs as secondary roles.

Planning requirements:
Create a planning system with:
- docs/plans/active/
- docs/plans/completed/
- docs/plans/handoffs/
- docs/plans/templates/

The plan templates must support:
- sprint contract
- execution plan
- evaluator report

Code design requirements:
- Use Playwright with TypeScript for UI and API automation.
- Use Playwright APIRequestContext as the default API transport layer.
- Keep selectors inside the owning page objects by default.
- Keep reusable page behavior in page objects.
- Keep multi-page business behavior in flow helpers.
- Use component objects only for repeated page fragments.
- Use assertion helpers for reusable expectations.
- Prefer simple fixtures over clever abstractions.
- Prefer explicit readable code over heavy indirection.

UI harness requirements:
- Create page-object files named from the actual application pages where possible.
- Create flow helpers for the most important journeys from docs/specs/ui-journeys.md.
- Create starter smoke coverage for the most critical UI paths.
- Add at least one accessibility-oriented starter test where the product makes sense for it.

API harness requirements:
- Organize API clients by the real endpoint domains from docs/specs/api-endpoints.md.
- Use wrapper client classes and schema validation.
- Add starter smoke coverage for auth and one major business domain.
- Add one contract-style example using schema validation.
- Add explicit status/body/header assertions where meaningful.

Hybrid test requirements:
- Add one example test that seeds or prepares state via API and validates via UI.

Fixtures and auth requirements:
- Create environment-aware fixtures.
- Include storageState support where appropriate.
- Reflect the auth/session model from docs/specs/product-under-test.md.
- Do not hardcode secrets.
- Create .env.example with representative non-secret placeholders.

Validation requirements:
Add npm scripts for:
- bootstrap
- lint
- format
- format:check
- typecheck
- test
- test:ui
- test:api
- test:smoke
- test:contract
- test:a11y
- docs:check
```

```
- validate
- report
- plan:init
- new:spec

Recommended validate behavior:
- format check
- lint
- typecheck
- smoke tests

Playwright config requirements:
- separate UI and API projects
- environment selection through env vars
- trace retention on failure
- screenshots on failure
- video retention on failure for UI
- HTML report output under artifacts/reports/playwright-report
- test results output under artifacts/reports/test-results
- sensible retries
- base URL configurable via env
- storageState support
- tags or project targeting for smoke, regression, contract, and accessibility
- global setup / teardown hooks

Artifacts and debugging requirements:
Use predictable artifact locations:
- artifacts/reports
- artifacts/screenshots
- artifacts/videos
- artifacts/traces
- artifacts/logs

Write runbooks that explain:
- how to run a single test
- how to open the Playwright report
- how to inspect traces
- how to regenerate auth state
- how to run only API tests
- how to switch environments

Redaction requirements:
- never log raw secrets
- never log raw auth tokens
- redact auth headers
- redact cookies
- redact PII and session values where practical

CI requirements:
Create:
- .github/workflows/ci.yml
- .github/workflows/smoke.yml
- .github/workflows/docs-check.yml

Expected CI behavior:
- lint
- typecheck
- run smoke UI and API tests
- upload Playwright report and failure artifacts when needed
- validate docs structure where practical

Starter test requirements:
Create a minimal but meaningful starter suite aligned to the application-specific artifacts:
- UI smoke test(s)
- UI regression sample
- one accessibility sample
- API auth smoke test
- API major-domain sample
- API contract-style sample
- one hybrid example

Do not generate generic nonsense tests. Use the discovered journeys and feature names.

Definition of done for the scaffold:
The scaffold is not done unless:
- required structure exists
- key docs contain meaningful starter content
- CLAUDE.md exists and is under 200 lines, using @imports for detailed docs
- .claude/rules/ contains path-scoped rules for page objects, API clients, tests, and selectors
- .claude/agents/ contains native subagent definitions for planner, generator, evaluator, and reviewer
- .claude/skills/ contains at least new-page-object, new-test-spec, and run-evaluator workflows
- .claude/settings.json contains hooks for lint-on-edit, changelog-on-commit, and notifications
- CHANGELOG.md exists with the initial scaffold entry
- .worktreeinclude lists .env files for worktree support
- root instruction files exist
- plan templates exist
- package scripts exist
- Playwright config exists and is coherent
- starter tests exist
- docs and naming reflect the real application where known
- no secrets are committed
```

```
- the repo is understandable to a future agent with no chat history

Bootstrap order:
1. create folder structure
2. create root config files
3. create package scripts and dependencies
4. create Playwright config
5. create core utilities and fixtures
6. create sample UI/API/hybrid tests aligned to the product
7. create docs index and architecture docs
8. create agent role docs
9. create plan templates
10. create CI workflows
11. run validation and fix scaffold issues
12. update README with exact getting-started steps

At the end, provide:
1. a concise summary of what was created
2. a short list of follow-up tasks
3. any assumptions that still need human confirmation

Do not ask to continue later. Perform the scaffolding now from the provided artifacts.
```

How to use the template in practice

You will usually give Claude Code:

1. the chapter prompt above
2. `playwright-scaffolding.md`
3. `docs/specs/product-under-test.md`
4. `docs/specs/ui-journeys.md`
5. `docs/specs/api-endpoints.md`

If the application is complex, also provide the optional discovery notes.

This gives Claude Code both:

- the **what** to build
- the **application context** to customize it correctly

Why this template is structured this way

This template intentionally separates:

- product discovery
- quick-start preparation
- actual scaffolding

That separation makes the resulting repo better because Claude Code is not forced to invent the product structure while also inventing the harness.

The repo should start with:

- structure

- standards
- plans
- agent roles
- starter automation patterns
- documentation that matches reality

That is much better than generating a bag of generic files.

After the scaffold exists

Once Claude Code has created the initial repo state, the normal loop becomes:

```
Planner → Generator → Evaluator
```

At that point, the repo should already contain:

- `CLAUDE.md` (under 200 lines, with `@imports`)
- `AGENTS.md`
- `.claude/rules/` path-scoped rules
- `.claude/agents/` native subagent definitions
- `.claude/skills/` repeatable workflows
- `.claude/settings.json` with hooks
- architecture docs
- standards docs
- plan templates
- starter page objects
- starter API clients
- starter fixtures
- starter tests
- CI skeletons
- artifact conventions

How to work after scaffolding

Start new sessions in `acceptEdits` mode for day-to-day work:

```
claude --permission-mode acceptEdits
```

Use skills to create new artifacts consistently:

```
/new-page-object ProductDetailPage
/new-test-spec "user adds item to cart"
```

Delegate investigation to subagents to keep your main context clean:

```
use a subagent to investigate how the checkout flow handles payment errors
```

Run the evaluator after completing a sprint:

```
use the evaluator subagent to verify the current sprint contract
```

Clear context between unrelated tasks with `/clear`. Long sessions with accumulated irrelevant context degrade Claude's performance.

Auto memory keeps learning. As you correct Claude's code style, naming, or patterns, it saves those corrections automatically. Future sessions start with that knowledge built in.

Then the agents can begin creating:

- new test cases
- new page objects
- new reusable components
- new API clients
- new fixtures
- deeper regression coverage
- refactors guided by the standards and plans

Final rule

> If `playwright-scaffolding.md` exists, it is the truth.
> Not generic examples.
> Not memory.

> Not assumptions.

That is what makes the scaffold customized instead of generic.

PART I

THE LANDSCAPE

CHAPTER 1

The State of Browser Automation in 2026

Learning Objectives

By the end of this chapter, you will be able to:

- Trace the evolution of browser automation from Selenium RC to Playwright
- Explain why Playwright displaced Selenium as the default choice for new projects
- Define test flakiness and articulate why it is the central problem in browser automation
- Identify the role of WebDriver BiDi in the current landscape
- Make informed tool selection decisions based on architecture, not popularity

1.1 A Brief History of Breaking Browsers on Purpose

Browser automation has always been a fight against the browser itself.

In 2004, Jason Huggins created Selenium at ThoughtWorks. The original Selenium — later called Selenium RC (Remote Control) — worked by injecting JavaScript into the browser. Your test code talked to a Selenium server, which injected commands into the page via JavaScript. It was clever, it was hacky, and it was the only game in town.

The problem was the Same-Origin Policy. Browsers do not let JavaScript from one domain manipulate content from another domain. Selenium RC worked around this by proxying everything through a local server, effectively pretending all content came from the same origin. It worked, barely, with constant edge cases and browser-specific hacks.

In 2009, Simon Stewart at Google created WebDriver — a completely different approach. Instead of injecting JavaScript, WebDriver controlled the browser through its native automation interface. Each browser provided a driver binary (chromedriver, geckodriver, msedgedriver) that accepted HTTP commands and translated them into browser-native actions.

WebDriver was better. Your tests sent HTTP requests to a driver, the driver talked to the browser natively, and the Same-Origin Policy was irrelevant. In 2011, Selenium and WebDriver merged into Selenium 2.0. In 2018, WebDriver became a W3C standard. The architecture looked like this:

```
Test Code  →  HTTP + JSON  →  Browser Driver  →  Native Protocol  →  Browser
```

This architecture served the industry for over a decade. It was language-agnostic (any language that could make HTTP requests could drive a browser), it was standardized (W3C meant browser vendors had to support it), and it scaled horizontally through Selenium Grid.

But it had problems. Fundamental, architectural problems that no amount of clever wait strategies could fix.

The Cracks in the Foundation

WebDriver's HTTP-based architecture means every command is a separate HTTP request-response cycle. Finding an element is one request. Clicking it is another. Checking if it is visible is a third. A simple login test might execute 50+ HTTP round-trips.

This has three consequences:

Latency accumulates. Each HTTP request has connection overhead — DNS resolution, TCP handshake, serialization, deserialization. For a test with 200 commands, this overhead adds seconds of pure protocol latency before you account for the browser doing anything.

Race conditions are structural. Between the moment your test checks "is this element visible?" and the moment it clicks the element, the page can change. The check and the click are separate HTTP requests with a

gap between them. In that gap, React can re-render a component, Angular can update the DOM, and your element reference becomes stale. This is not a bug — it is the architecture.

Debugging is opaque. When a test fails, you get an HTTP error response with a JSON body. "Element not interactable." Why? The protocol does not tell you. Was it hidden? Was it behind a modal? Was it detached from the DOM? You screenshot the page and guess.

The Playwright Revolution

In 2020, Microsoft released Playwright. The team behind it included engineers who had built Puppeteer at Google (Chrome's headless automation library) and understood the WebDriver architecture's limitations intimately.

Playwright took a fundamentally different approach. Instead of communicating over HTTP, Playwright connects to the browser's debugging protocol via a persistent WebSocket connection. For Chromium, this is the Chrome DevTools Protocol (CDP). For Firefox and WebKit, Playwright developed custom protocols that provide equivalent capabilities.

The architecture looks like this:

```
Test Code  →  WebSocket (persistent)  →  Browser Process  →  Renderer
```

This single change eliminates the three problems above:

No per-command overhead. The WebSocket connection stays open for the entire test. Commands are small JSON messages on an already-open channel. The protocol overhead that dominated Selenium tests effectively disappears.

Auto-waiting is built into the protocol. When you tell Playwright to click an element, it does not send a "find" request and then a "click" request with a gap between them. It sends a single "click this selector" command that the browser processes atomically — finding the element, waiting for it to be visible and stable, scrolling it into view, and clicking it, all in one operation. The race condition gap does not exist because there is no gap.

Debugging is rich. Playwright captures traces — complete recordings of every network request, DOM snapshot, console message, and action. When a test fails, you open the trace in Trace Viewer and watch exactly

what happened, step by step, with before-and-after screenshots for every action. This is not a screenshot on failure. This is a time-travel debugger.

Why Playwright Won

By 2025, Playwright had become the dominant choice for new browser automation projects. The reasons were not subtle:

Factor	Selenium	Playwright
Installation	Download browser driver binaries separately, match versions manually	`npx playwright install` — one command, all browsers
Waiting	Mix of implicit waits, explicit waits, Thread.sleep, and prayer	Auto-waiting built into every action
Isolation	Shared browser state, clean up cookies manually	Browser contexts provide true isolation
Parallel execution	Selenium Grid, complex infrastructure	Built-in workers, no external infrastructure
Debugging	Screenshots on failure, console logs	Trace Viewer, video recording, step-by-step replay
Speed	HTTP round-trips per command	WebSocket, near-zero protocol overhead
Multi-browser	Chromium, Firefox, WebKit (via separate drivers)	Chromium, Firefox, WebKit (all built in)
Language support	Java, Python, C#, Ruby, JavaScript	TypeScript/JavaScript, Python, Java, .NET

This is not a marketing comparison. It is an architectural analysis. Playwright's advantages are not features bolted on top — they are consequences of a fundamentally better architecture.

> **Common Mistake:** Evaluating tools based on GitHub stars, npm downloads, or blog post frequency. These measure marketing and community momentum, not technical merit. Evaluate based on architecture, reliability, and how the tool handles failure — because failure is where you spend most of your debugging time.

1.2 The Flakiness Problem

If there is one word that defines the browser automation industry's central failure, it is "flaky."

A flaky test is a test that sometimes passes and sometimes fails without any change to the code under test or the test itself. Run it once, it passes. Run it again, it fails. Run it a third time, it passes. The result is nondeterministic.

Flakiness is not a minor inconvenience. It is an existential threat to test automation's value proposition. Here is why:

Flaky tests destroy trust. When a test fails, the team needs to know: is this a real bug, or is the test flaky? If they cannot answer that question quickly, they stop trusting the test suite. Once trust is gone, people stop looking at test results. Once people stop looking at test results, the test suite is dead — it costs money to run and provides no value.

Flaky tests are contagious. One flaky test in a suite of 500 is annoying. Twenty flaky tests make the entire suite unreliable. Engineers start adding `retry: 3` to everything. They add sleep statements "just in case." They skip intermittent failures in CI. Each workaround makes the problem worse.

Flaky tests have root causes, and most of them are architectural. The industry treats flakiness as an inevitable cost of browser automation. It is not. Most flakiness comes from three sources:

1. **Race conditions between test code and application code.** Your test checks for an element before the application has rendered it. This is the most common source and is largely eliminated by Playwright's auto-waiting.
2. **Shared state between tests.** Test A creates a user. Test B assumes that user exists. Test C deletes all users. Run them in order, they pass. Run them in parallel, they fail. Browser contexts solve this.
3. **External dependencies.** Your test hits a real API that is slow or intermittent. Network interception and mocking solve this.

Playwright does not magically eliminate flakiness. But it provides architectural solutions to the most common causes. Selenium leaves you to solve these problems yourself, with varying degrees of success.

> **Pro Tip:** Track your flake rate. Seriously. Run your suite 10 times without changing anything. If any test fails even once, that test is flaky. A healthy suite has a flake rate below 1%. Most Selenium suites run between 5% and 30%. If

> you do not know your flake rate, you do not know how much of your debugging time is wasted.

1.3 WebDriver BiDi: The Standard Catches Up

The W3C did not ignore Playwright's success. In 2023, they began work on WebDriver BiDi (Bidirectional) — a new protocol that combines WebDriver's standardization with the bidirectional communication model that makes Playwright fast.

WebDriver BiDi uses WebSocket connections instead of HTTP. It supports event subscriptions (the browser can push events to your test, rather than your test polling the browser). It provides richer debugging information. In many ways, it is the W3C acknowledging that Playwright's architectural choices were correct and standardizing them.

As of July 2026, WebDriver BiDi is no longer a future promise — it is the present. Selenium 4.x exposes low-level BiDi APIs covering roughly 70% of the CDP surface, working in Chrome and Firefox. WebdriverIO v9 uses BiDi as its default protocol. Playwright uses parts of it internally for Firefox and WebKit.

Selenium itself is still on the 4.x line (around 4.41 as of July 2026). Selenium 5 has not shipped — when it does, its headline feature will be high-level BiDi APIs that wrap today's low-level ones. So the standard is catching up in real, usable increments. But the high-level developer experience — auto-waiting, browser contexts, the trace viewer — is still Playwright's advantage. Today, Playwright's custom protocols provide a richer and more reliable experience than what BiDi-based stacks expose.

If you are starting a new project in 2026, use Playwright. If you are maintaining a Selenium codebase, Chapter 18 covers when and how to migrate. If you are evaluating whether to wait for WebDriver BiDi to mature before deciding, do not wait — the architectural advantages of Playwright are available now.

1.4 The Tool Landscape in 2026

Browser automation in 2026 has three serious contenders and a long tail of niche tools.

Playwright

Architecture: Direct browser protocol communication via WebSocket. Controls Chromium, Firefox, and WebKit through browser-specific protocols.

Strengths: Auto-waiting, browser contexts, trace viewer, built-in test runner, cross-browser from a single API, excellent TypeScript support, fast release cycle (monthly — current stable is 1.61 as of July 2026), first-party Test Agents (planner, generator, healer) since v1.56.

Weaknesses: Relatively young ecosystem compared to Selenium, smaller (but growing) hiring pool, limited mobile support (browser-only, no native mobile).

Best for: New projects, teams that value reliability and developer experience, organizations willing to invest in modern tooling.

Cypress

Architecture: Runs inside the browser as JavaScript. Commands execute in the same event loop as the application.

Strengths: Excellent developer experience for single-page applications, real-time reload during test development, time-travel debugging, strong community. Cypress v15 adds `cy.prompt()` natural-language test authoring, Cypress Studio, and self-healing capabilities — making Cypress a real player in AI-assisted authoring.

Weaknesses: Chromium-only for most of its history (Firefox support is still experimental), cannot handle multiple browser tabs, cannot test across multiple domains easily, slower test execution than Playwright, less suitable for complex enterprise applications.

Best for: Frontend developer-written tests for SPAs, teams already invested in the Cypress ecosystem.

Selenium

Architecture: HTTP-based WebDriver protocol, plus low-level WebDriver BiDi APIs in the 4.x line. Language-agnostic. Scales via Selenium Grid. Still on 4.x (around 4.41) as of July 2026 — Selenium 5 has not shipped.

Strengths: W3C standard, widest language support, massive community and hiring pool, works with every browser that implements WebDriver, decades of accumulated knowledge and tooling.

Weaknesses: HTTP round-trip overhead, manual wait management, complex setup, fundamentally fragile architecture for modern SPAs.

Best for: Large organizations with existing Selenium investments, teams requiring languages not supported by Playwright (Ruby, for example), organizations mandating W3C-standard protocols.

> **Pro Tip:** Tool selection is a business decision, not a religious one. The "best" tool depends on your team's skills, your application's architecture, your CI/CD infrastructure, and your timeline. This book teaches Playwright because it is the best default choice for new projects in 2026. But Chapter 19 provides an honest comparison to help you make the right decision for your specific situation.

1.5 What This Book Will Build

Over the next 20 chapters, you will build a complete browser test automation system. Not example by example in isolation — a cohesive, production-grade framework.

Here is what you will have by the end:

- A **Playwright test framework** with TypeScript, organized by the Page Object Model with composition (not inheritance)
- **Fixtures** for dependency injection, authentication, and environment-specific configuration
- **Network interception** for mocking APIs and testing error states without depending on external services
- **Visual regression tests** with configurable thresholds and baseline management
- **Accessibility tests** integrated into your existing test suite, not as a separate afterthought
- **A CI/CD pipeline** in GitHub Actions with parallel execution, artifact collection, and intelligent retry
- **Reporting** with Playwright's HTML reporter, trace viewer integration, and optionally Allure
- A **migration guide** for moving an existing Selenium suite to Playwright without rewriting everything at once

Each chapter builds on the previous one. By Chapter 15, when you build the full framework, you will understand every piece because you will have learned it individually first.

Exercises

Exercise 1.1: Tool Audit (Beginner)

Look at your current or most recent project. Answer these questions: 1. What browser automation tool does it use? 2. How are browser drivers managed (manually downloaded, package manager, built-in)? 3. What is the approximate flake rate? (Run the suite 5 times and count inconsistent results.) 4. How long does the full suite take to run? 5. What happens when a test fails? (Screenshot? Video? Trace? Nothing?)

Exercise 1.2: Architecture Comparison (Intermediate)

Draw two sequence diagrams: 1. A Selenium test that navigates to a page, finds an input, types text, clicks a button, and verifies the result. Show every HTTP request. 2. The same test in Playwright. Show the WebSocket messages.

Count the number of protocol round-trips in each. What is the ratio?

Exercise 1.3: Flakiness Analysis (Advanced)

If you have access to a CI system with browser tests: 1. Export the last 30 days of test results. 2. Identify every test that had at least one failure and at least one pass in that period. 3. Calculate the flake rate for each test and the suite overall. 4. Categorize the root causes: race condition, shared state, external dependency, or other.

What percentage of failures were real bugs vs. flakiness?

Quiz

1. **What was the fundamental architectural difference between Selenium RC and WebDriver?**
   a) Selenium RC was slower
   b) Selenium RC injected JavaScript; WebDriver used native browser automation interfaces
   c) Selenium RC only supported Firefox

   d) WebDriver required a paid license

2. **Why does Playwright's WebSocket-based architecture reduce flakiness compared to Selenium's HTTP-based architecture?**
   a) WebSocket is a newer technology
   b) It eliminates the gap between checking element state and acting on it
   c) WebSocket is faster at downloading web pages
   d) It uses less memory

3. **What are the three most common root causes of flaky browser tests?**
   a) Slow computers, bad code, poor documentation
   b) Race conditions, shared state, external dependencies
   c) Wrong browser, wrong driver, wrong version
   d) Network latency, CPU usage, disk space

4. **What is WebDriver BiDi?**
   a) A new browser developed by the W3C
   b) A bidirectional protocol that combines WebDriver standardization with WebSocket communication
   c) Playwright's proprietary protocol
   d) A replacement for HTTP

5. **When should you choose Selenium over Playwright for a new project in 2026?**
   a) Always, because it is the W3C standard
   b) When you need languages not supported by Playwright, or when organizational mandates require W3C-standard protocols
   c) When you want faster tests
   d) When you need cross-browser testing

Answers: 1-b, 2-b, 3-b, 4-b, 5-b

Key Takeaways

- Browser automation evolved from JavaScript injection (Selenium RC) to HTTP-based native control (WebDriver) to persistent WebSocket-based control (Playwright).
- Playwright's architectural advantages — auto-waiting, browser contexts, trace viewer — are consequences of its protocol design, not features bolted on top.
- Flakiness is the central enemy of browser automation. Most flakiness is architectural, not inevitable.
- WebDriver BiDi is shipping today — low-level in Selenium 4.x (roughly 70% of the CDP surface) and the default protocol in WebdriverIO v9 — but Selenium is still on the 4.x line as of July 2026, and Playwright's high-level advantages remain available now.
- Tool selection should be based on architecture and team context, not popularity or tradition.

Career Translation

Resume phrasing

- "Led evaluation and adoption of Playwright for browser test automation, reducing test suite flake rate from 18% to under 1% and cutting execution time by 65%"
- "Architected migration strategy from Selenium WebDriver to Playwright, enabling parallel execution without Selenium Grid infrastructure"
- "Established browser automation standards and tool selection criteria based on architectural analysis of WebDriver vs CDP protocols"

Cover letter framing

"I approach tool selection as an architectural decision, not a preference. When our team's Selenium suite reached a 20% flake rate, I did not add more retries — I analyzed the root causes, identified that 80% of failures were WebDriver protocol race conditions, and led the evaluation and migration to Playwright. The result was not just fewer flakes, but a fundamental shift in how our team thought about test reliability."

Interview framing

"When asked about browser automation tools, I start with architecture. Selenium uses HTTP round-trips per command — find element, click element, check state, each a separate request. Playwright uses a persistent WebSocket connection with atomic operations. That architectural difference is why Playwright auto-waiting works and Selenium's explicit waits are a workaround. I have used both extensively, and the tool choice should depend on the team's context, but for new projects in 2026, the architectural case for Playwright is strong."

What not to say

- "Selenium is dead" — it is not, and saying so shows you do not understand the legacy ecosystem
- "Playwright is better because it is newer" — recency is not an argument; architecture is
- "I do not like Selenium because it is slow" — this is vague; explain the HTTP round-trip overhead specifically
- "We should switch to Playwright because everyone is switching" — bandwagon reasoning does not impress interviewers

Interview Depth Check

Question 1

Prompt: "Walk me through the architectural differences between Selenium WebDriver and Playwright. Why does it matter?"

What a strong answer should cover:

- WebDriver's HTTP request-response model vs Playwright's persistent WebSocket connection
- The per-command overhead in Selenium (DNS, TCP, serialization for each action)
- The race condition gap between "check state" and "act on state" in Selenium
- Playwright's atomic operations (find + wait + act in one command)
- How this architectural difference leads to Playwright's auto-waiting
- The implications for debugging (opaque HTTP errors vs rich traces)

Example answer:

- "Selenium sends a separate HTTP request for every action — finding an element is one request, clicking it is another, checking if it is visible is a third. Each request has connection overhead. More importantly, between those requests, the page can change. That gap is where flaky tests are born."
- "Playwright connects via WebSocket and keeps the connection open for the entire test. When you call page.click(), it sends a single command that atomically finds the element, waits for it to be actionable, and clicks it. There is no gap for a race condition."
- "For debugging, Selenium gives you an HTTP error response with a message like 'element not interactable.' Playwright gives you a trace file where you can see the DOM state, network requests, and console logs at the exact moment of failure."

Question 2

Prompt: "What is test flakiness, and how would you systematically reduce it in a browser test suite?"

What a strong answer should cover:

- Definition: tests that produce inconsistent results without code changes
- The three main root causes: race conditions, shared state, external dependencies
- Why flakiness destroys trust in the test suite
- Measurement: how to calculate flake rate
- Architectural solutions: auto-waiting, browser contexts, network mocking
- Process solutions: flake tracking, quarantine, root cause analysis

Example answer:

- "A flaky test is one that sometimes passes and sometimes fails with no changes to the application or test code. I start by measuring — run the suite 10 times, identify every test with inconsistent results, and calculate the flake rate."

- "Most flakiness falls into three categories. Race conditions: the test checks for something before the app is ready. Shared state: tests depend on each other's side effects. External dependencies: the test hits a real API that is slow or unreliable."
- "For each category, there is an architectural solution. Auto-waiting for race conditions. Browser contexts or isolated environments for shared state. Network interception and mocking for external dependencies. Retries are a symptom treatment, not a cure."

Question 3

Prompt: "Your organization uses Selenium for 2,000 browser tests. Leadership asks if you should migrate to Playwright. What is your recommendation framework?"

What a strong answer should cover:

- Assessment of current pain points (flake rate, execution time, maintenance cost)
- Cost of migration (rewriting tests, retraining team, dual infrastructure)
- Incremental migration strategy (run both tools in parallel, migrate by feature area)
- Risk analysis (what breaks during migration, rollback plan)
- Decision criteria that could lead to "stay with Selenium" (stable suite, team expertise, no pain points)

Example answer:

- "I would not recommend migrating based on tool popularity alone. I would start by measuring the current pain: what is the flake rate, what is the execution time, how much time does the team spend debugging infrastructure vs writing tests?"
- "If the suite is stable and the team is productive, migration has a cost with limited benefit. If the flake rate is above 5%, execution time is a bottleneck, or the team spends more time fighting Selenium than writing tests, migration becomes compelling."

- "For 2,000 tests, I would recommend an incremental approach: write all new tests in Playwright while maintaining the Selenium suite. Migrate existing tests feature by feature, starting with the flakiest ones. Run both suites in CI until the Selenium suite is empty. This avoids a big-bang rewrite and lets the team learn Playwright gradually."

About This Sample

You have just read **Chapter 1 of Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-13-browser-automation-playwright/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.