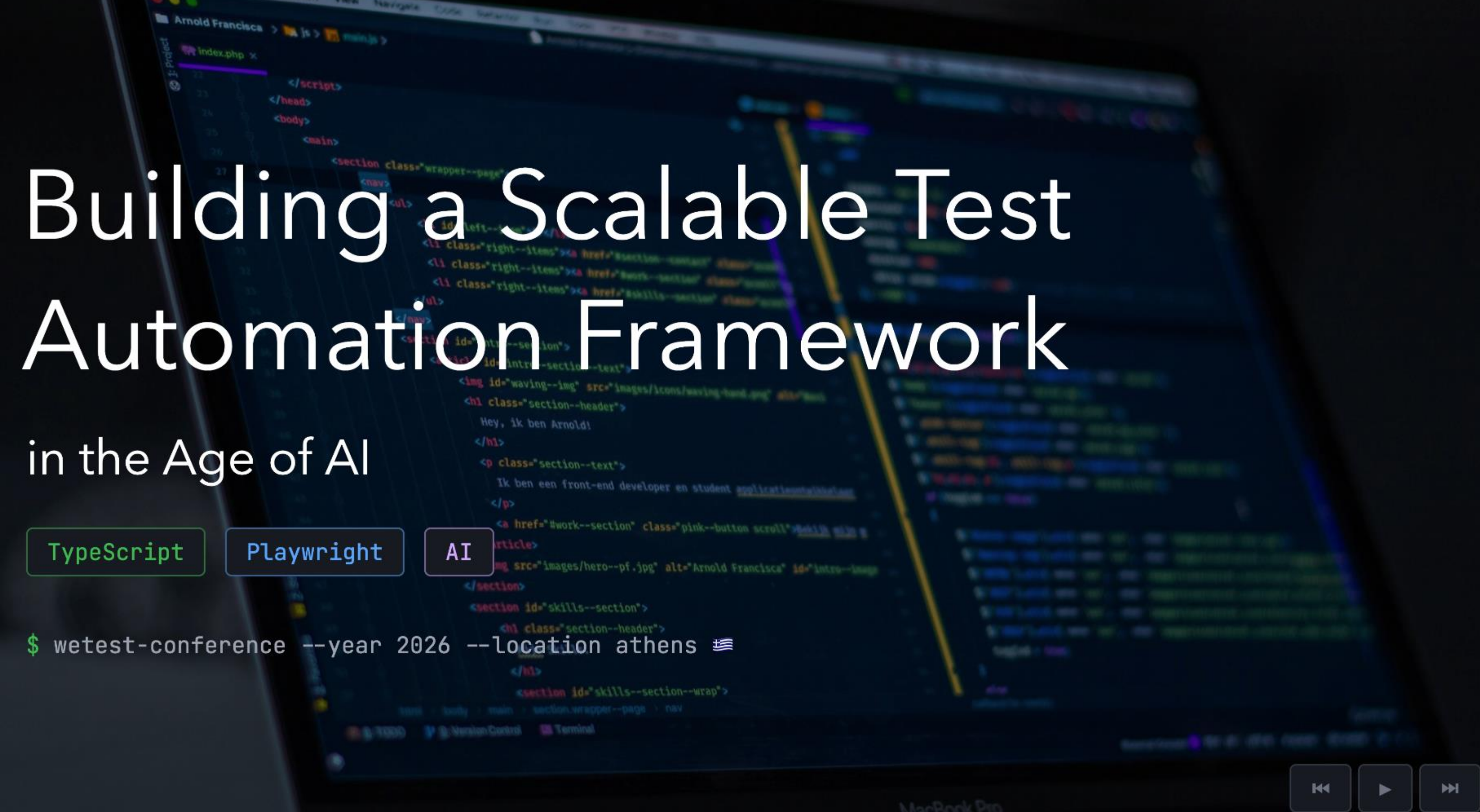
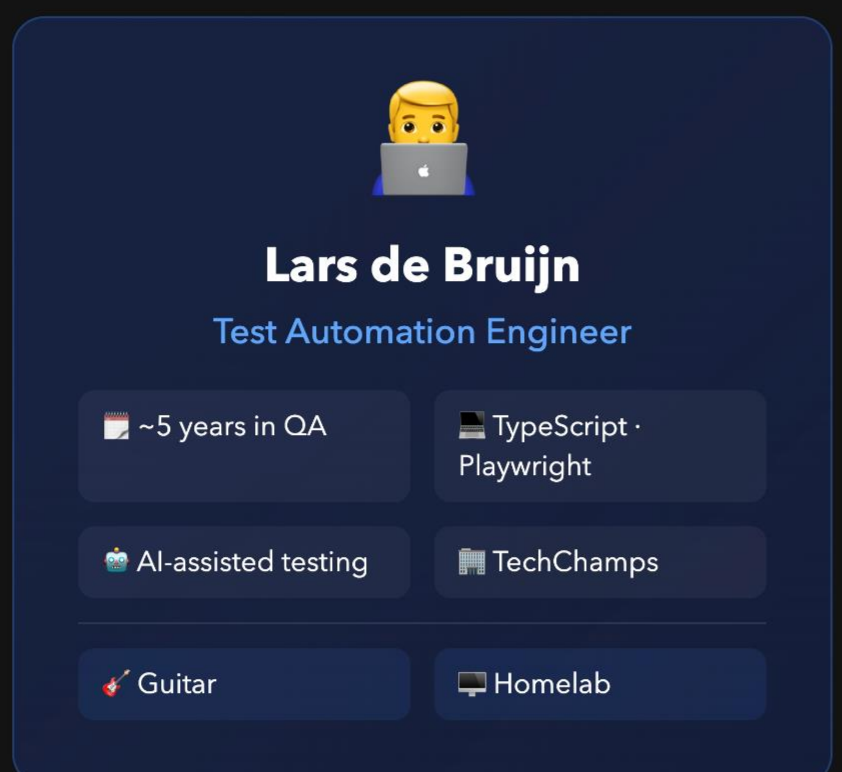

Building a Scalable Test Automation Framework
in the Age of AI
TypeScript
Playwright
AI
$ wetest-conference --year 2026 --location athens 🇬🇷

# Meet the Speakers



THE PROBLEM

# Most test suites collapse because there's no scalable foundation to build on.

AI accelerates output, but it also accelerates technical debt.

> *"If you think good architecture is expensive, try bad architecture."*
>
> – Brian Foote & Joseph Yoder

# The Goal

**Maintainable**

Change one thing, break nothing

**Scalable**

100 tests or 10,000 – same confidence

**AI-Ready**

Patterns AI can follow and extend correctly

# What We'll Cover

Folder Structure

#1

Playwright Config

#2

Page Object Model

#3

Parallel Workers

#4

Storage State

#5

Hooks & Fixtures

#6

Global Setup & Teardown

#7

Test Sharding

#8

Design Patterns

#9

AI Tips & Tricks

#10

```
$ ls ./framework --topic
```

```
# "A Folder Layout that Shields Tests from Internal Complexity"
```

# Two Worlds, One Repo

**tests/**

WHAT TO TEST

The intent. The business flows.
A new team member reads this
and understands the product.

`ui/ · api/`

**support/**

HOW TO TEST IT

The mechanics. Selectors, helpers,
config, page objects, utilities.
The engine under the hood.

`pages/ · utils/ · config/ · environment/`

Tests should read like specifications. Support handles the rest.

# Let's see the code

SWITCHING TO THE IDE

// 📁 FOLDER STRUCTURE

```
$ cat playwright.config.ts
```

# ⚙️ Playwright Config

```
# "Centralized Config for Multi-Project Scaling"
```

# The Config is the Contract

// one file defines the rules every test plays by

**Test Behaviour**

Timeouts · Retries
Parallelism · Reporters

**Browser & App**

Base URL · Devices
Selectors · Tracing

**Projects**

Auth setup → Chrome
Multi-browser · Roles

```
// key decisions made here
testIdAttribute: 'data-test'
trace: 'on'
retries: CI ? 1 : 0
```

// project dependency chain

- `setup` project runs first
- Saves auth state to disk
- Browser projects depend on it

# Let's see the code

SWITCHING TO THE IDE

// ⚙ PLAYWRIGHT CONFIG

```
$ tsc --pattern "support/pages/**/*.ts"
```

```
# "Reusable Page Classes for Scalable UI Tests"
```

# What is the Page Object Model?

## Without POM

Tests contain selectors directly.
The same locator appears in 20 files.
One UI change → update 20 tests.

Fragile · Duplicated · Hard to read

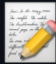
## With POM

Each page/component = one class.
Selectors live in one place.
Tests call methods, not locators.

Maintainable · Reusable · Readable

UI changes → update **one class**, not dozens of tests

# POM Structure



### Page Class

- Constructor receives `page`
- Locators as class properties
- Actions as methods
- No assertions inside

### Shared Components

- Navbar, modals, toasts
- Used across page objects
- Composed, not duplicated
- One place to maintain

The test says **"what"** happens. The Page Object knows **"how"** to do it.

# Let's see the code

SWITCHING TO THE IDE

```
bash — wetest-slidev

$ cd support/pages/

$ ls

  login.page.ts · home.page.ts · base.page.ts
```

// PAGE OBJECT MODEL

```
$ npx playwright test --workers=50%
```

# ⚡ Parallel Tests & Workers

```
# "More Workers, Less Waiting. Scale Without Chaos"
```

# Parallelism: Speed With Control

Playwright distributes tests across **workers** with separate processes, each with its own browser instance. No shared state. No interference.

### Workers run in parallel

Each worker picks up the next available test. The more workers, the faster the suite capped by your CPU limit.

### Full isolation by default

Each worker gets its own browser context. Login state, cookies, and storage are never shared between tests.

### Serial when needed

Some flows depend on order: e.g. create → edit → delete. Group those explicitly to run sequentially on a single worker.

**Rule of thumb:** run more workers on CI where resources are dedicated, fewer locally to keep your machine responsive. Playwright auto-scales to half your CPU cores if you let it.

# Let's see the code

SWITCHING TO THE IDE

// ⚡ PARALLEL TESTS & WORKERS

```
$ cat tests/.auth/customer.json | jq '.origins | length'
```

```
# "Log In Once. Test Everything."
```

# The Login Tax

WITHOUT STORAGE STATE

**Login on every test**

- ✕ Every test logs in
- ✕ Every test waits for auth
- ✕ Every test hits the auth server

**100 tests × 3s login**

= 5 minutes wasted, every run

WITH STORAGE STATE

**Login once, reuse everywhere**

- ✓ Login runs **once** before the suite
- ✓ Auth state saved to a JSON file
- ✓ All tests load it instantly

**1 login, 100 tests benefit**

Setup project → all browser projects

# Multiple Roles, Multiple States

REAL APPS HAVE MULTIPLE USER TYPES, SO YOUR FRAMEWORK SHOULD TOO

### SETUP PROJECT

Runs once.
Authenticates all roles.
Saves state files.

### ADMIN-TESTS PROJECT

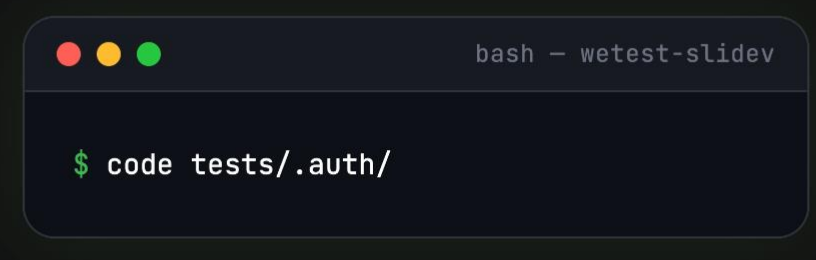
Loads admin.json.
All tests start
already authenticated.

### BUYER-TESTS PROJECT

Loads buyer.json.
Same pattern,
different permissions.

# Let's see the code

SWITCHING TO THE IDE




// 💾 STORAGE STATE

```
$ grep -r "beforeAll\|beforeEach\|test.extend" support/
```

# 🪝 Hooks & Fixtures

```
# "Stop Repeating Setup. Inject What Tests Need"
```

# Hooks – The Lifecycle of a Test

Suite starts — **beforeAll** – run once. Expensive setup: seed data, start services, create shared resources that are specific to that test suite.

Each test — **beforeEach** – reset state, navigate to starting page, prepare fresh context.

🧪 **TEST RUNS**

Each test — **afterEach** – clear cookies, reset local storage, capture screenshots on failure.

Suite ends — **afterAll** – cleanup: delete test data, close connections, remove created resources that are specific to that test suite.

DESIGN PRINCIPLE

*"Good hooks keep your tests clean, fast, and independent."*

# Fixtures – How They Work

Test declares — **Declare by name** – a test says what it needs in its arguments. No manual setup.

Playwright — **Setup runs automatically** – Playwright resolves dependencies and prepares the fixture before the test body.

🧪 **TEST RUNS – fixture is ready and injected**

Always — **Teardown is guaranteed** – cleanup runs after the test, whether it passed or failed.

**FIXTURES COMPOSE**

A fixture can depend on other fixtures. Playwright resolves the full chain – in order, every time.

```
browser → page → loginPage
          ↳ homePage
          ↳ chatPage
```

**A FIXTURE CAN BE ANYTHING**

Page Object | API Client | Auth Context | Test Data

# Hooks vs Fixtures – When to Use Which

## Hooks

- Scoped to a single `describe` block
- You write both setup *and* cleanup
- Not reusable across files

**Best for**

Quick, one-off setup that only one test group needs – e.g. navigate to a specific page before each test in a suite

## Fixtures

- Defined once, injected anywhere by name
- Playwright guarantees teardown. No leaks
- Composable: fixtures can use other fixtures

**Best for**

Page objects, API clients, authenticated contexts. Anything shared across multiple test files or the whole team

**Reusability**

Hooks: file-local

Fixtures: project-wide

**Cleanup safety**

Hooks: manual

Fixtures: guaranteed

**At scale**

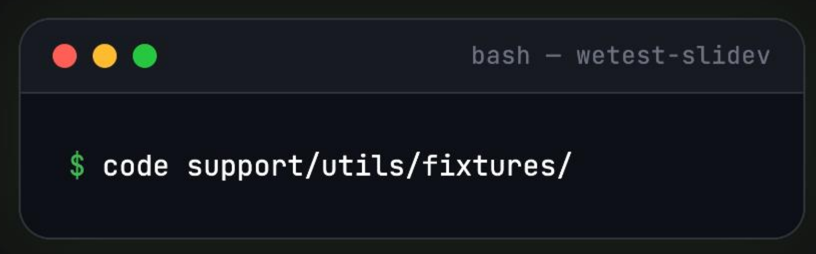
Hooks: get messy

Fixtures: stay clean

**Default to fixtures. Reach for hooks only when the setup truly belongs to one test group.**

# Let's see the code

SWITCHING TO THE IDE




// 🪝 HOOKS & FIXTURES

```
$ node support/globalSetup.ts && node support/globalTeardown.ts
```

```
# "Prepare the World Once. Then Let Tests Focus on Tests"
```

# Before Any Test Runs

Global setup runs once before any test starts

**Load Environment**

Read .env files
Set process.env
Global then env-specific

**Check Infrastructure**

Is Docker running?
Is the app reachable?
Start if needed

**Reset Database**

Fresh migrations
Seed test data
Known starting state

**Clean Stale State**

Remove old auth files
Clear temp data
Start clean

**Global Setup**

Environment-level, not test-level · Setup test data · Runs once before all tests

**Global Teardown**

Restores environment to baseline · Clean test data · Runs once after all tests

# Let's see the code

SWITCHING TO THE IDE

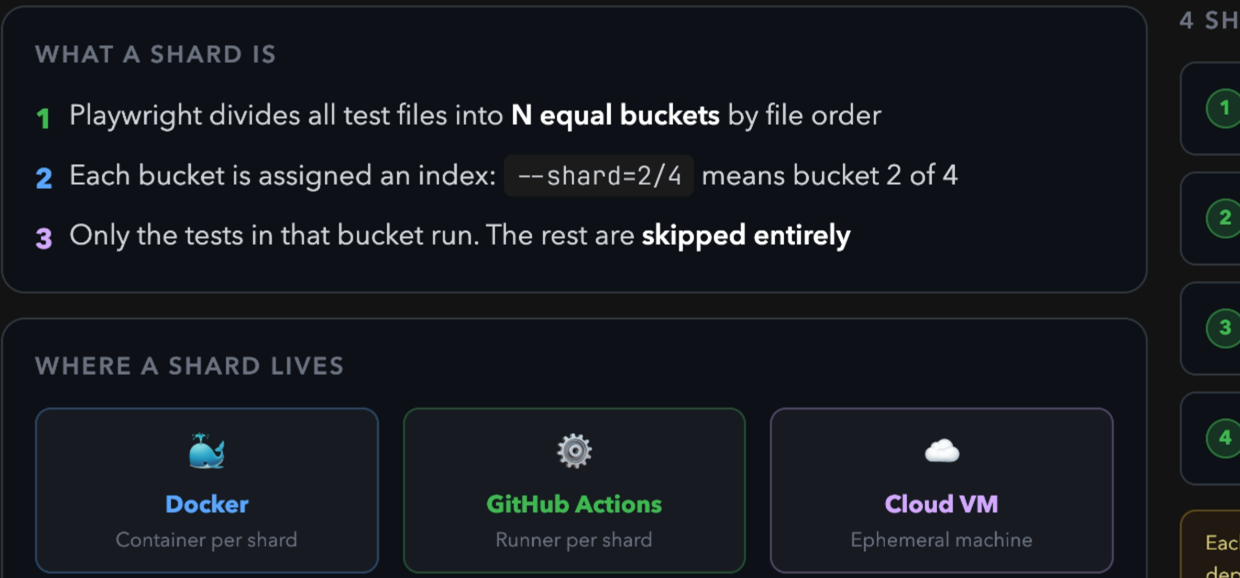
// 🌍 GLOBAL SETUP & TEARDOWN

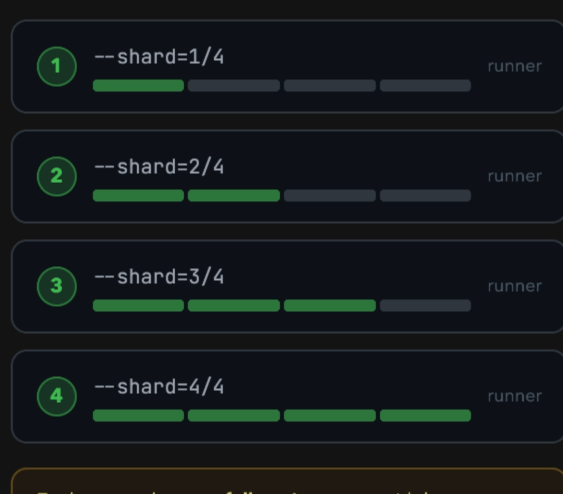
```
$ npx playwright test --shard=1/4
```

```
# "Split the Suite, Multiply the Speed"
```

# What is a Shard?

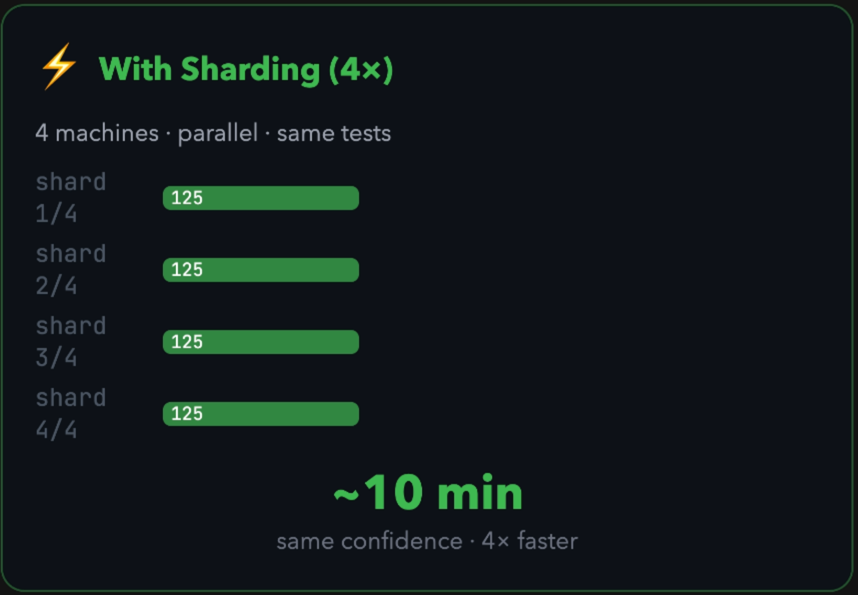
`A shard = one isolated slice of your test suite running on its own machine`



# Why Sharding?

Your suite grows. CI gets slower. Developers wait. Feedback loops break.




| 1 shard | 2 shards | 4 shards | 8 shards |
|---|---|---|---|
| **~40 min** | **~20 min** | **~10 min** | **~5 min** |
| baseline | 2× faster | 4× faster | 8× faster |

# Sharding vs Workers – Two Levels of Speed

## Workers – within one machine

- → Multiple threads inside one Playwright process
- → Tests run simultaneously, not one by one
- → Each worker gets its own browser context
- → Limited by your machine's CPU cores

## Sharding – across machines

- → Splits the suite across multiple CI jobs
- → Each machine runs a slice of the test files
- → No communication between shards
- → Scales beyond what one machine can do

### THEY WORK TOGETHER

Each shard can still use multiple workers internally. A suite of 400 tests split into 4 shards gives each machine 100 tests. Then workers on each machine run those 100 in parallel. You get horizontal scale **(shards)** and vertical scale **(workers)** at the same time.

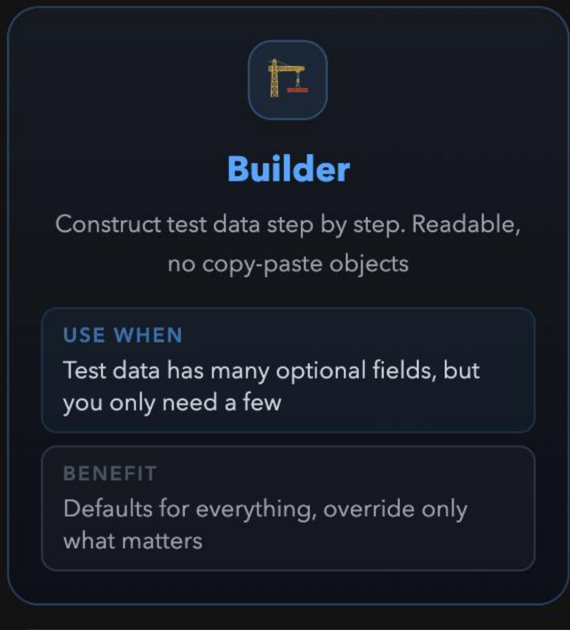
```
$ grep -r "Builder\|Factory\|Facade\|Strategy" support/
```

# Design Patterns

```
# "Proven Solutions to Problems Every Test Suite Eventually Hits"
```

# Why Design Patterns in Tests?

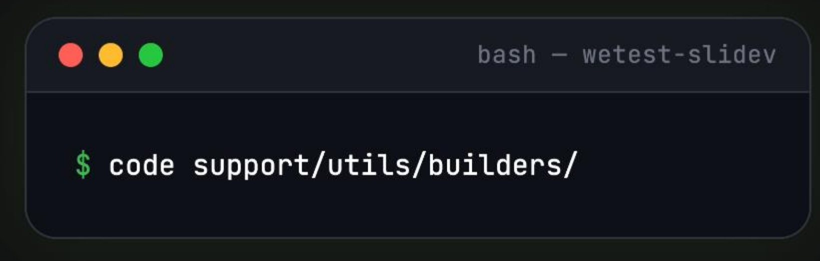
TEST CODE IS PRODUCTION CODE. IT DESERVES THE SAME RESPECT.



✦ Consistent patterns let AI generate code that **fits your framework on the first try**

# Let's see the code

SWITCHING TO THE IDE




// 🎨 DESIGN PATTERNS

```
$ cat agent.md | grep -i "pattern\|fixture\|page-object"
```

```
# "Teach the AI Your Patterns. It Will Follow Them at Scale"
```

# Teaching AI Your Framework

GIVE AI THE RIGHT FILES AND IT GENERATES CODE THAT FITS ON THE FIRST TRY

### MCP

Model Context Protocol. Gives AI direct access to tools, APIs, and live data. No copy-pasting context, no stale snapshots.

`real-time context for AI tooling`

### skills.md

Reusable task recipes and playbooks. Step-by-step instructions for common tasks: what to create, where to put it, and which patterns to follow.

`reusable task definitions for AI`

### agent.md

Runtime behaviour config. Autonomy level, tool access, confirmation thresholds and escalation rules. How the agent decides and acts.

`runtime behaviour of the AI agent`

### instructions.md

Project conventions and constraints. Naming rules, architecture patterns and dos and don'ts. Everything AI needs to generate code that fits your codebase.

`project conventions AI must follow`

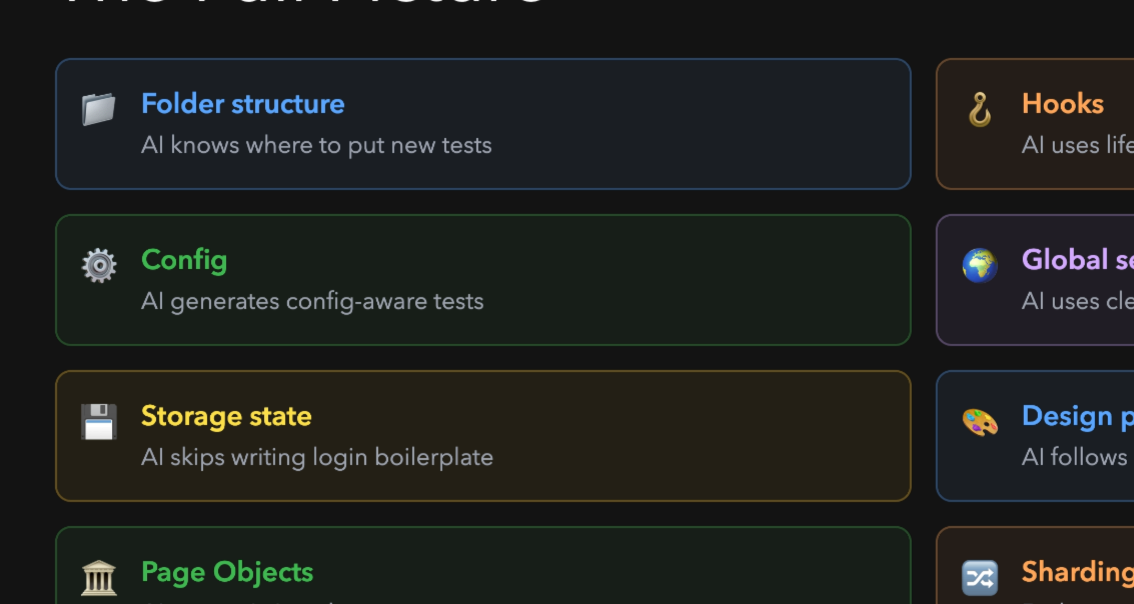
AI CONTEXT LOADED

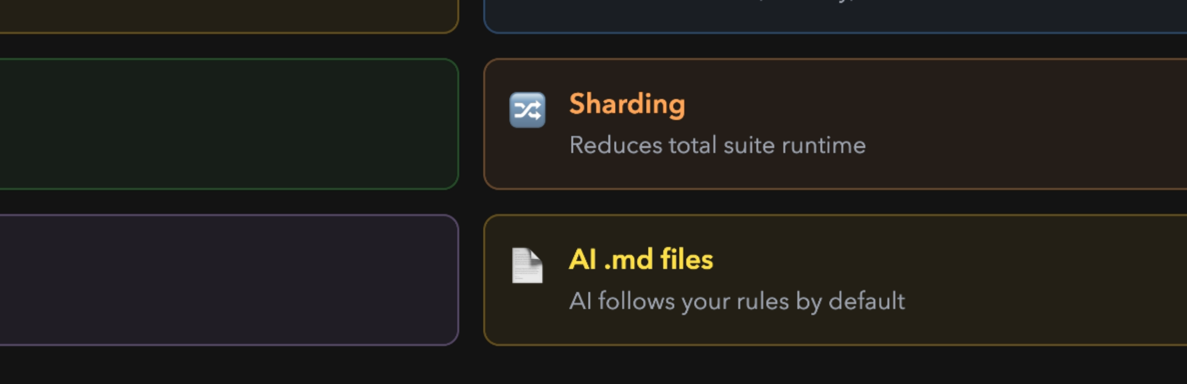
### monorepo/

Shared packages across apps. AI has access to the development source code for test generation.

`workspace structure AI must respect`

# The Full Picture





**Folder structure**
AI knows where to put new tests

**Hooks**
AI uses lifecycle hooks correctly

**Config**
AI generates config-aware tests

**Global setup & teardown**
AI uses clean setup and teardown

**Storage state**
AI skips writing login boilerplate

**Design patterns**
AI follows Builder, Factory, Facade

**Page Objects**
AI uses existing classes

**Sharding**
Reduces total suite runtime

**Fixtures**
AI reuses declared fixtures

**AI .md files**
AI follows your rules by default

**Build the foundation. Let AI multiply it.**

# Key Takeaways

### Foundation first

Invest in structure, patterns, and conventions before scaling test count

### Isolation always

Storage state, fixtures, and clean setup eliminate 80% of flakiness

### AI needs context

agent.md and consistent patterns turn AI into a multiplier

WETEST CONFERENCE 2026 · ATHENS, GREECE 🇬🇷

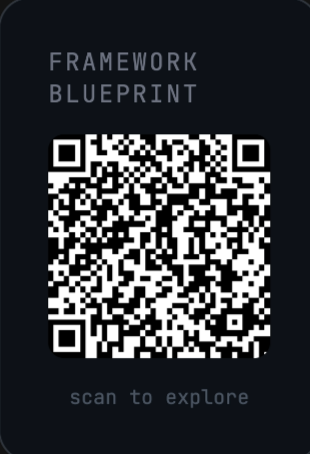
# Σας ευχαριστώ πολύ!

Building a Scalable Test Automation Framework in the Age of AI


## Lars de Bruijn

- Github: @Lars-db
- LinkedIn: /in/lars-de-bruijn-/
- Company website: techchamps.io
- Personal website: arcanaops.io


FRAMEWORK
BLUEPRINT



scan to explore


## Marco Maes

- Github: @marcomaes
- LinkedIn: /in/marcomaes/
- Company website: polteq.com
- Personal website: thetestboss.com


TypeScript · Playwright · AI

Questions? → find us after the talk

Credits for the Toolshop (https://practicesoftwaretesting.com) → Roy de Kleijn

FRAMEWORK BLUEPRINT

github.com/Lars-db/WeTest-Framework-Blueprint