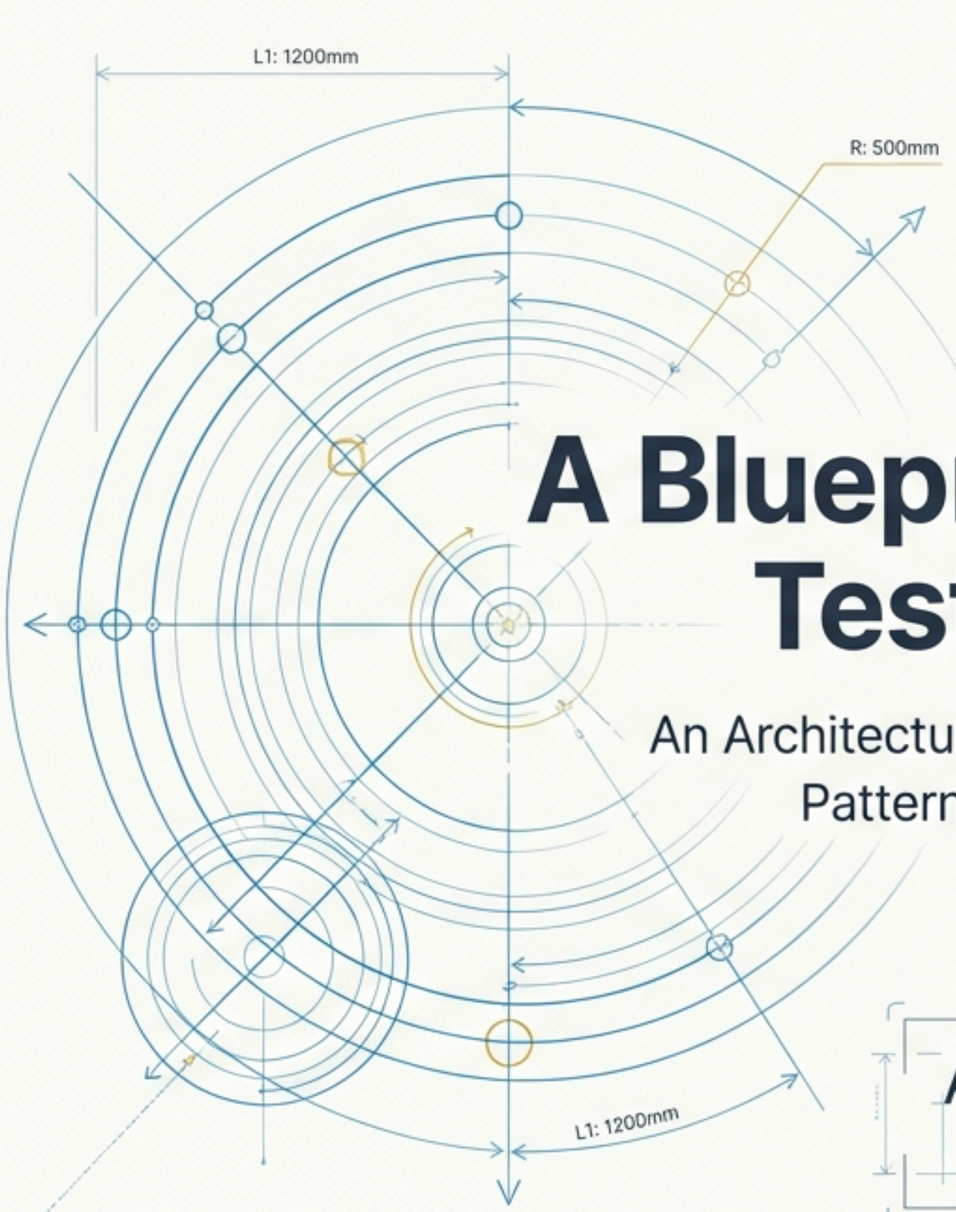
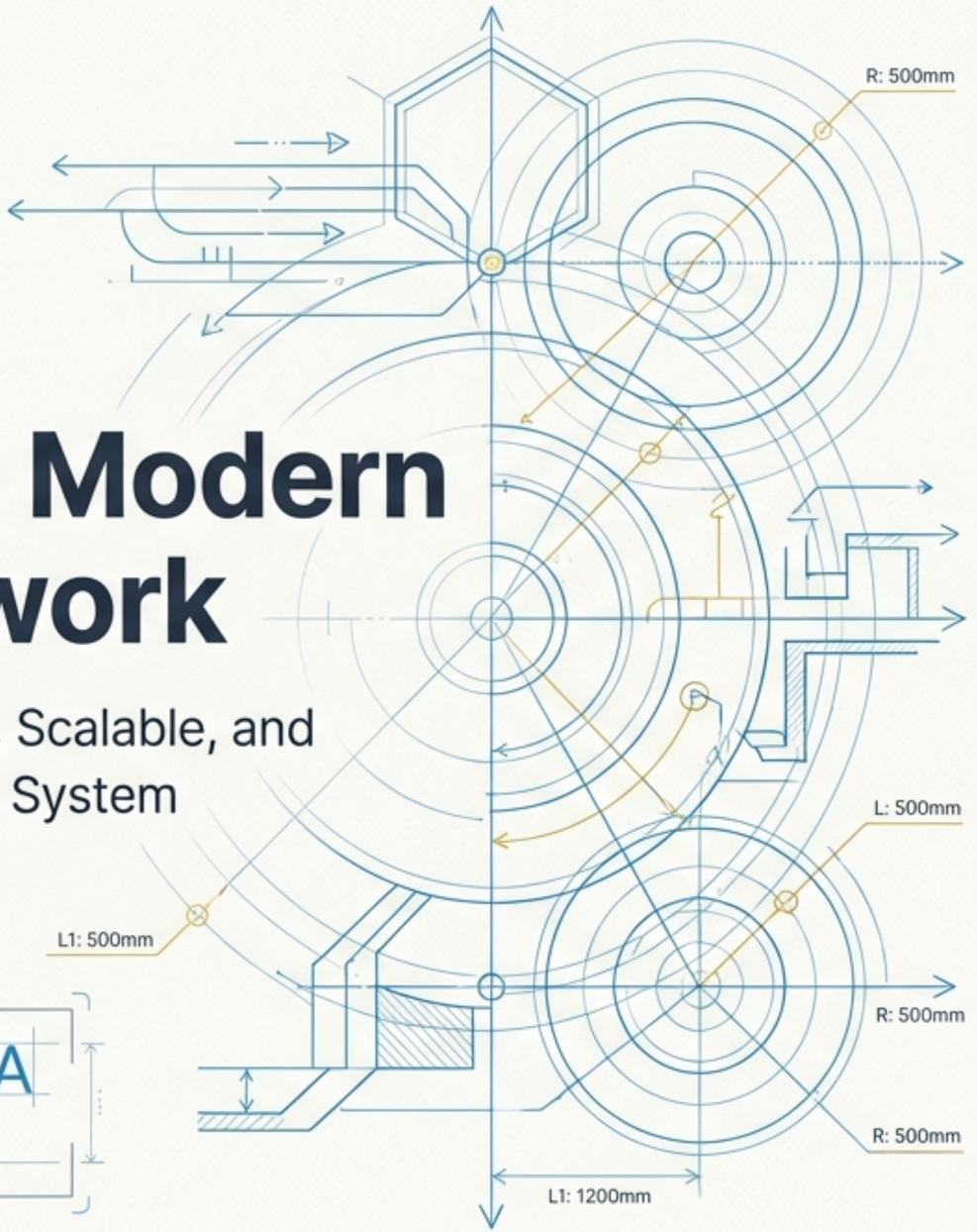


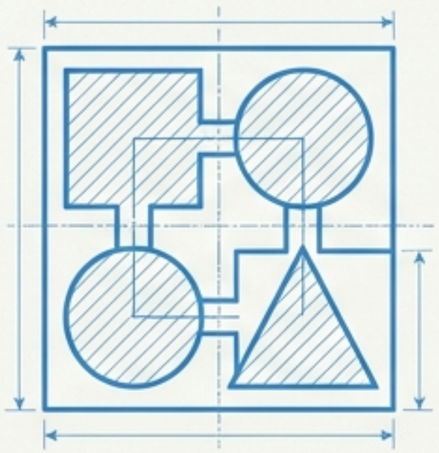
A Blueprint for a Modern Test Framework

An Architectural Tour of a Unified, Scalable, and Pattern-Driven Automation System

Automation AAA Framework

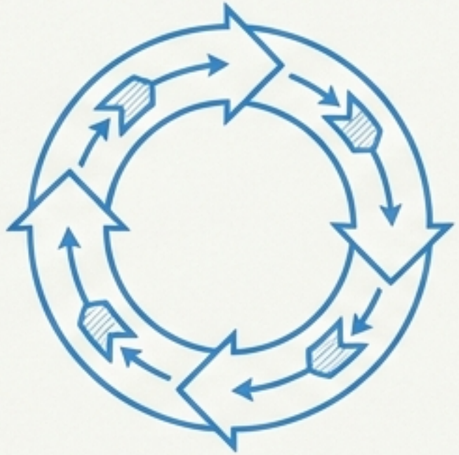


Engineering for Four Core Objectives



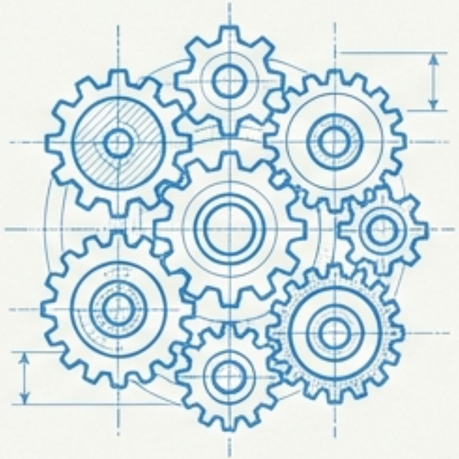
Unified Testing Infrastructure

Integrate different testing types (API, UI, SAB, Performance) in a single repository to create a single a source of truth.



Radical Code Reusability

Share common utilities, types, and test assistants across different testing domains to eliminate redundancy.



Designed for Maintainability

Organize code by domain and feature to minimize coupling and maximize cohesion, making the system easy to evolve.

Built for Scalability

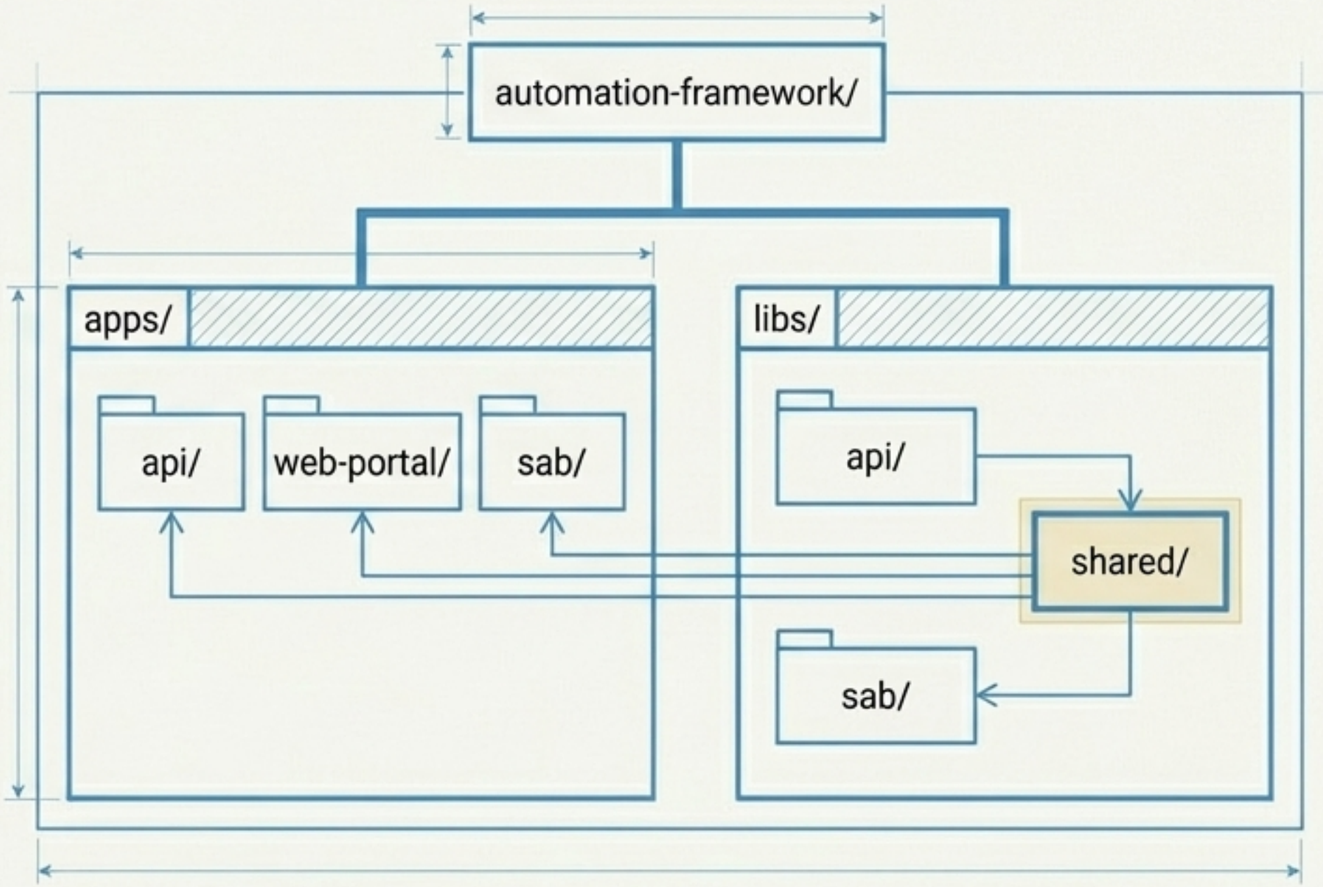
Natively support parallel test execution, dynamic test user management, and flexible configuration to handle growing complexity.

The Foundation: A Unified Monorepo Managed by Nx

The entire framework is housed in a single monorepo to ensure consistency and simplify dependency management.

Nx (Nrwl Extensions) is used to orchestrate the monorepo, enabling:

- **Smart Dependency Management:** Automatic dependency graph computation and affected project detection.
- **Seamless Code Sharing:** Type-safe imports across projects using TypeScript path mapping.
- **Efficient Task Orchestration:** Parallel execution of linting, testing, and building tasks.



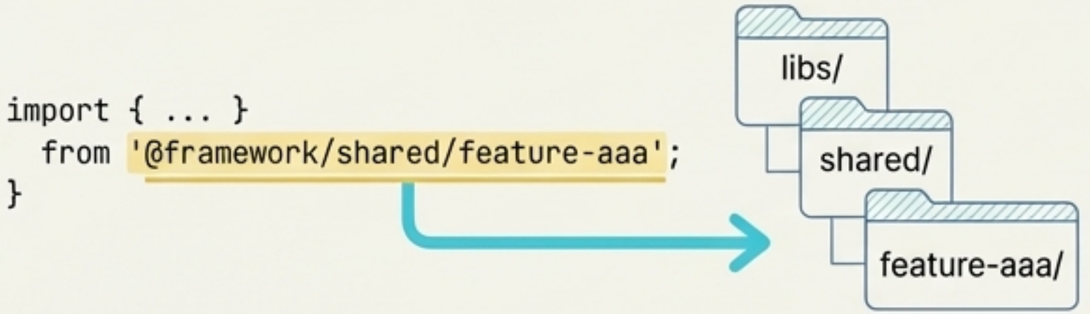
Orchestrating Services with Precision

1. Centralized Configuration

A base `playwright.base.config.ts` provides global settings, with each application (`api`, `web-portal`, etc.) able to override and extend it. This creates a clear hierarchy.

2. Shared Libraries

Domain-specific and cross-domain libraries are universally accessible via TypeScript path mapping (e.g., `@framework/shared/feature-aaa`), ensuring type-safe code reuse.



3. Unified Test User & Reporting

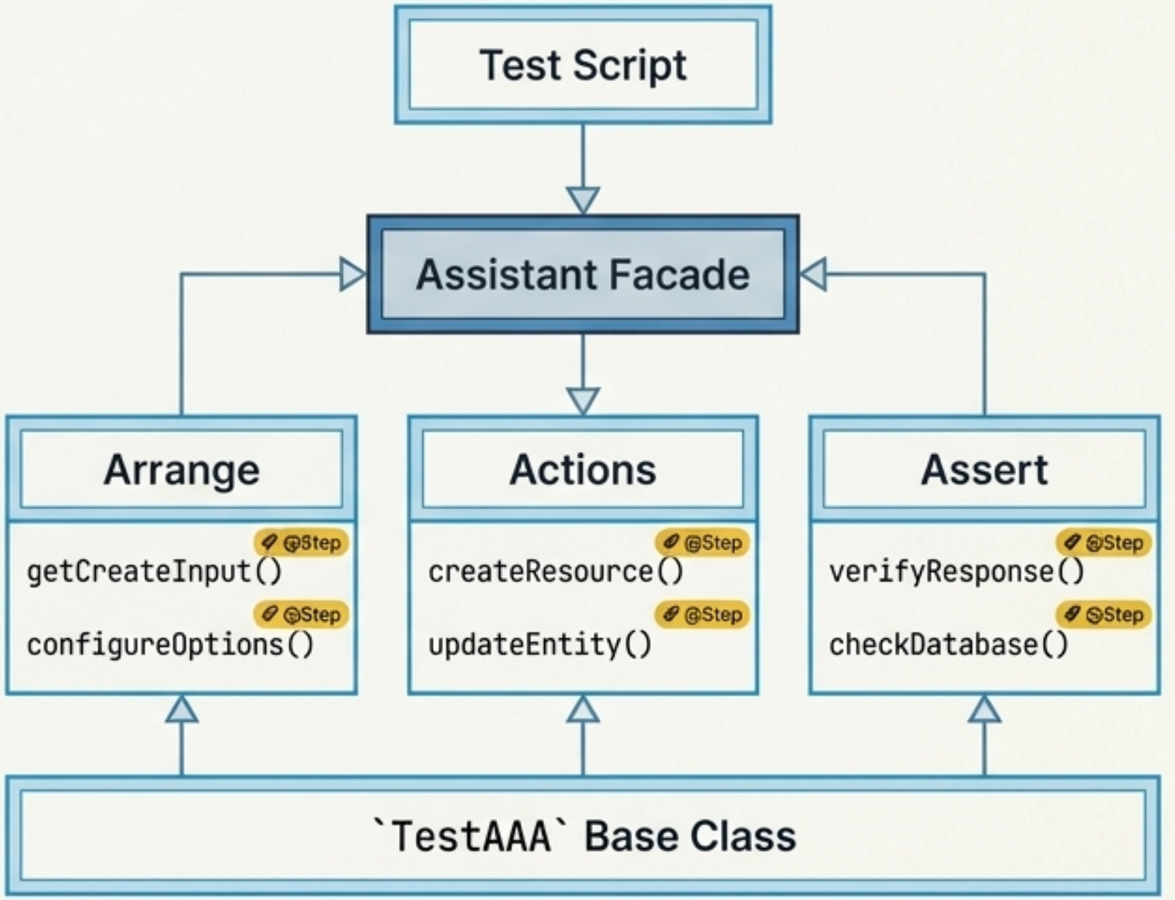
A centralized Test User Repository with a database-backed locking mechanism manages account lifecycles, while Report Portal integration provides consolidated test result tracking across all services.

The Core Logic: A Structured Arrange-Act-Assert (AAA) Pattern

Every test assistant across API, SAB, and Web Portal domains implements a consistent AAA pattern, providing three distinct interfaces for test composition.

Core Architectural Components

- **`TestAAA` Base Class**
  The foundation for all `Arrange`, `Actions`, and `Assert` classes. It provides shared access to API services, utility managers, and builders.
- **`@Step` Decorator**
  A custom decorator that automatically wraps methods in Playwright's `test.step()` for context-aware logging and reporting. This ensures every action is tracked.
- **Assistant Facade**
  A unified interface that bundles the `Arrange`, `Actions`, and `Assert` components into a single, easy-to-use assistant object.



The AAA Pattern in Action: An API Test Example

THE FLOW

1. ARRANGE

The test prepares all necessary data. Methods like `getCreateInputPayload()` and `getExpectedResult()` are used.

2. ACTIONS

The test executes the core operation. Methods like `createResource()` call the underlying API client.

3. ASSERT

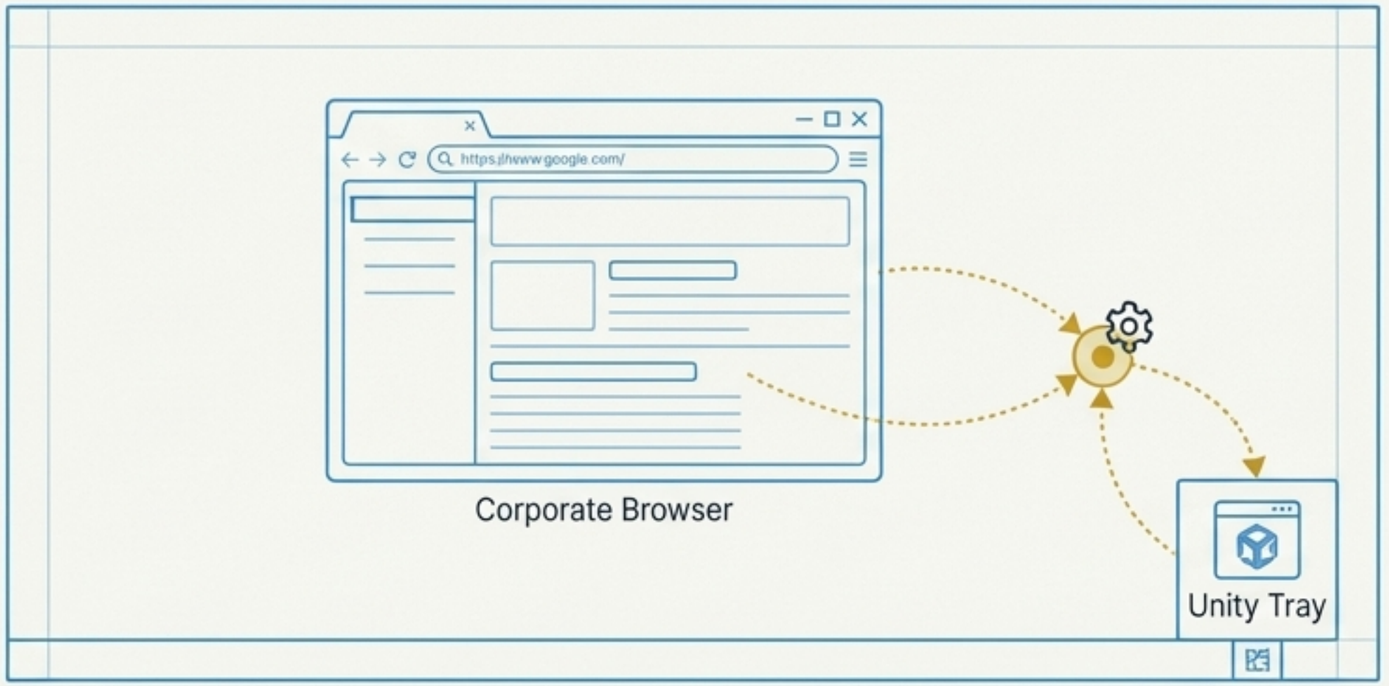
The test validates the outcome. Methods like `resourceCreatedOK()` compare the actual result against the expected result.

THE CODE

```
test('Create LoginPolicy', async ({ adminUser }) => {
   // 1. Acquire the assistant
   const adminAPITA = await adminUser.useAPIAssistant
     <LoginPolicyAPI, TALoginPolicyAPI>(LoginPolicyAPI);

   // 2. Arrange: Prepare test data
   const createInput = await adminAPITA.arrange.
     getCreateInputPayload({ isDenyAction: false });

   // 3. Actions: Execute API operation
   const actualResult = await adminAPITA.actions.
     createLoginPolicy(createInput);

   // 4. Assert: Validate results
   const expectedResult = await adminAPITA.arrange.get
     ExpectedResultForCreatePolicyOK(createInput);
   await adminAPITA.assert.loginPolicyCreatedOK(actualResult,
     expectedResult);
 });
```

The Crown Jewel: Tackling Complex Desktop Automation

The framework's most sophisticated architecture is designed for Secure Access Browser (SAB) testing—a native desktop application that combines a system tray component (Unity Tray) and a sandboxed browser engine (Corporate Browser).

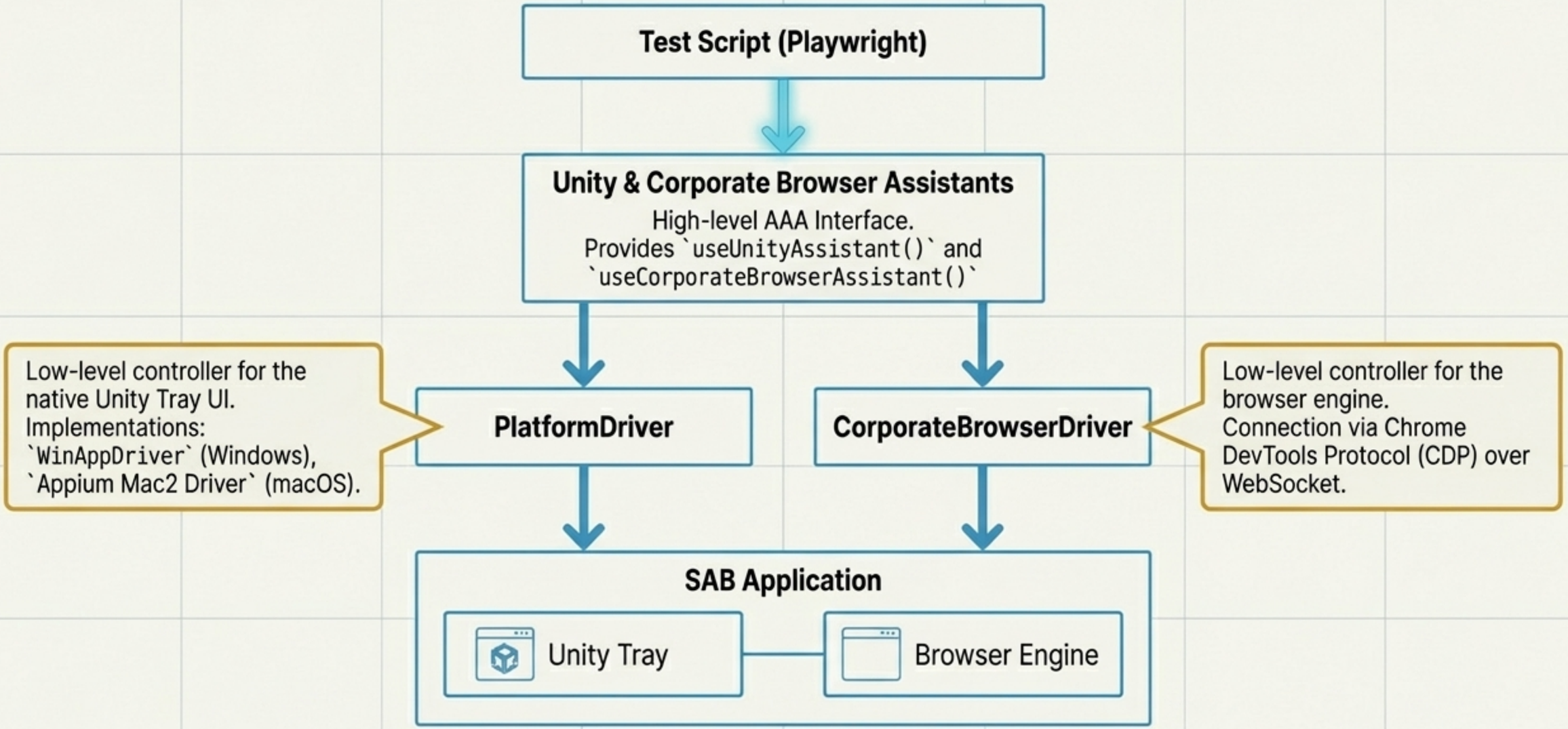


The Challenge

Automating SAB requires orchestrating multiple, distinct components simultaneously:

- Controlling a native UI element in the system tray.
- Driving a browser engine via a remote debugging protocol.
- Managing system-level interactions like keyboard shortcuts and window focus.
- Ensuring thread-safe parallel execution on isolated machines.

SAB Component Architecture: A Multi-Layered Solution



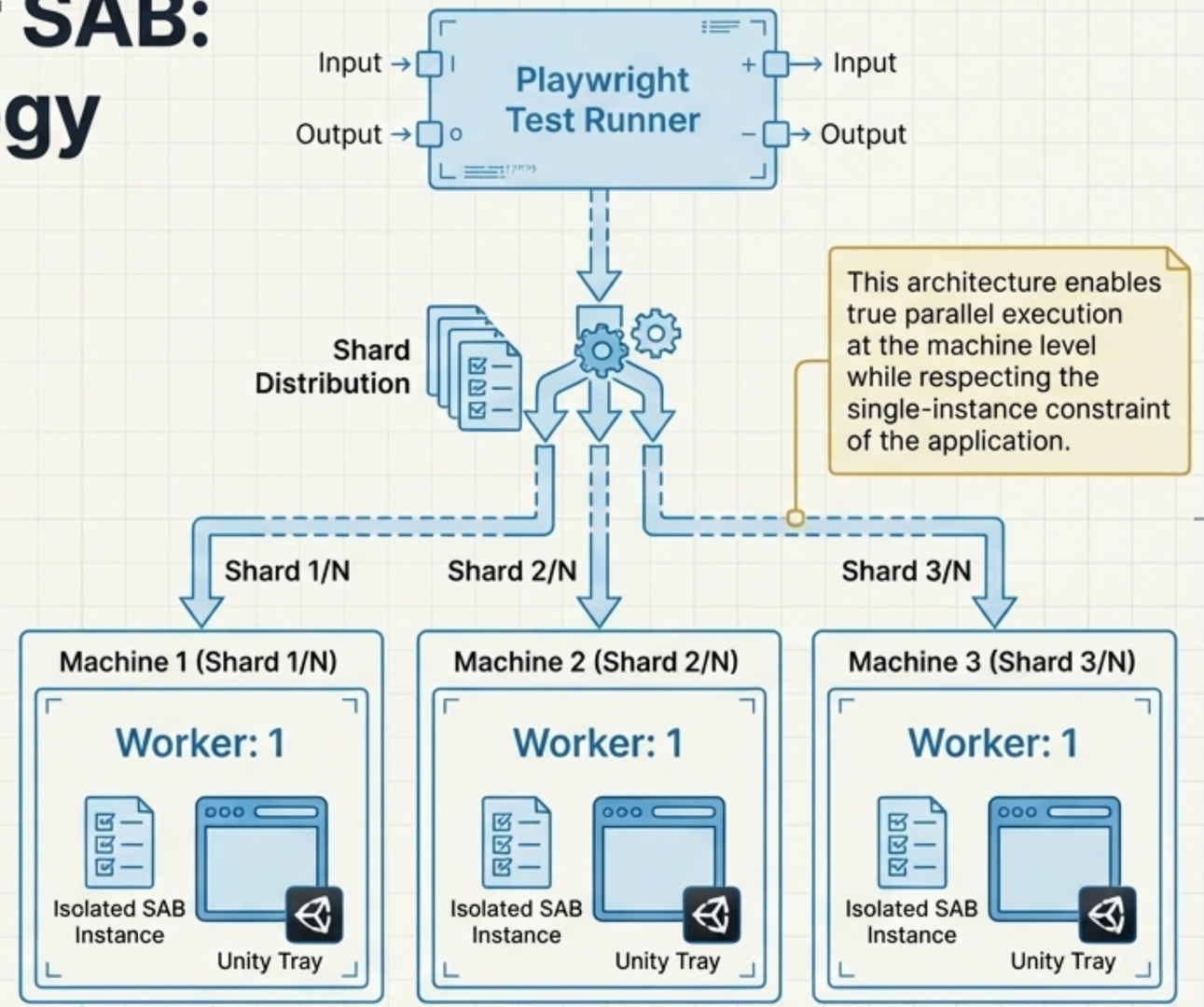
Achieving Parallelism for SAB: A Sharding-Based Strategy

The Constraint

The SAB application cannot run multiple instances on the same machine. This prevents standard multi-worker parallelization within a single OS.

The Solution: Test Sharding

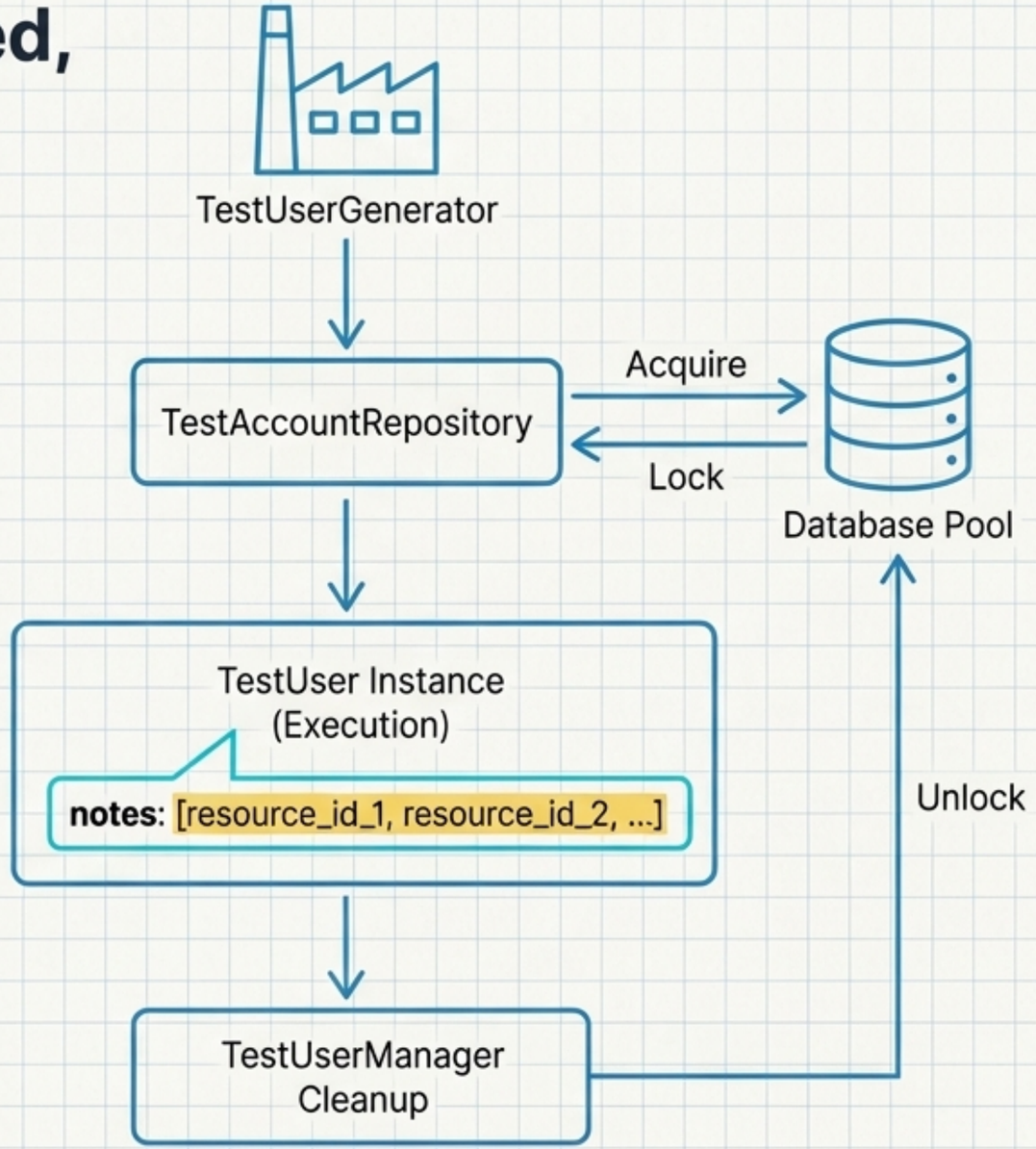
The framework leverages Playwright's `--shard=X/Y` feature to distribute the test suite across multiple machines. Each machine runs only a single worker (`workers: 1`), ensuring resource isolation.



The Test User Lifecycle: Centralized, Scalable, and State-Aware

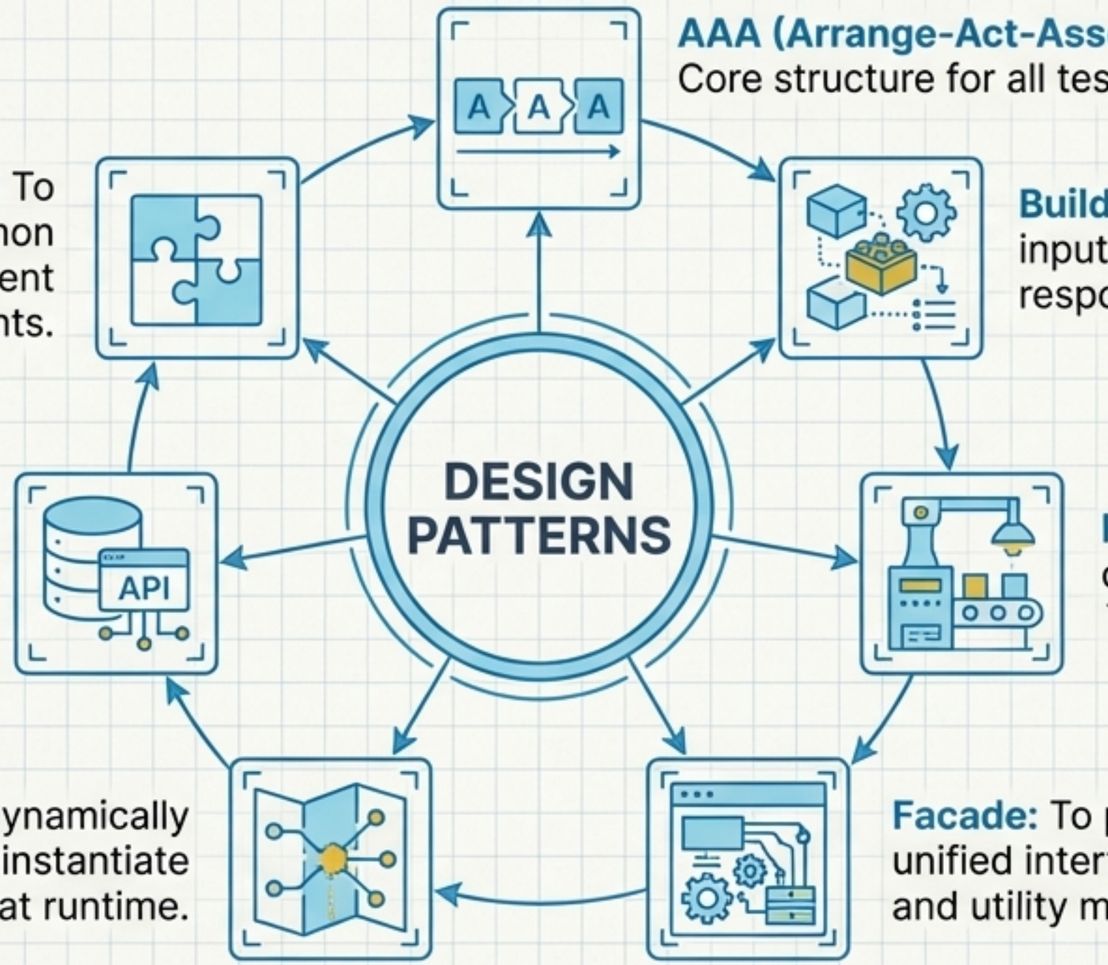
The Process

1. **Generation:** The `TestUserGenerator` acts as a factory to create diverse user personas (Admin, Random, etc.).
2. **Acquisition & Locking:** The `TestAccountRepository` acquires a user from a database pool, locking it to the specific test runner's hostname to prevent collisions during parallel SAB runs.
3. **Execution:** The `TestUser` instance serves as the central context object, acquiring assistants (`useAPIAssistant`) and tracking resources created during the test via its `notes` property.
4. **Cleanup:** The `TestUserManager` orchestrates the teardown, automatically deleting created resources and unlocking the user account in the database.



The Engineering Pattern Palette

The framework's architecture is built upon a collection of proven design patterns to ensure consistency, reusability, and maintainability.



AAA (Arrange-Act-Assert): Core structure for all test logic.

Builder: For creating complex input payloads and expected response objects.

Factory: For generating different types of `TestUser` instances.

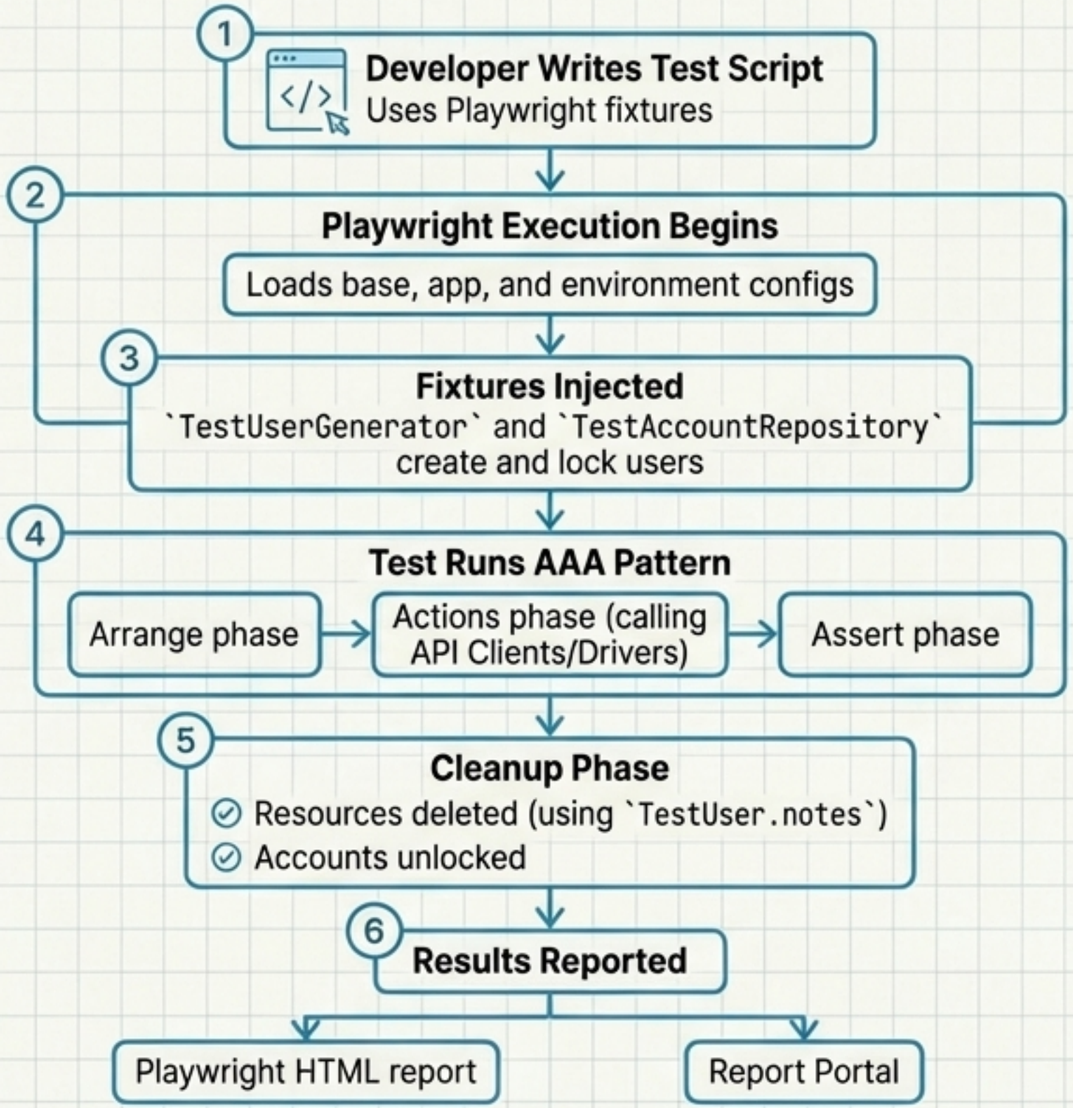
Facade: To provide a simplified, unified interface for assistants and utility managers.

Mapper: To dynamically resolve and instantiate assistants at runtime.

Repository: To abstract and manage the test account data source.

Mixin (SAB Domain): To compose and share common UI actions across different page assistants.

The Complete Test Execution Flow



This flow demonstrates a clear separation of concerns, from test intent to execution logic and resource management.

Core Technologies & Stack

Core Technologies

Nx: v17.1.3
(Monorepo management)

Playwright: v1.52.0
(End-to-end testing framework)

TypeScript: v5.1.6
(Type-safe development)

Node.js: v22.0.0
(Runtime environment)

Key Domain Dependencies

- **API Domain**
  - graphql
  - graphql-request
  - axios
- **SAB Domain**
  - appium
  - webdriverio
  - robotjs
- **Shared Utilities**
  - @aws-sdk/client-ec2
  - amazon-cognito-identity-js
  - @faker-js/faker
  - mysql2

Path Mapping Example

```
"paths": {
  "@framework/api/type-api":
   ["libs/api/type-api/src/
 index.ts"],
  "@framework/shared/feature
 -aaa": ["libs/shared/featur
 e-aaa/src/index.ts"],
  "@framework/shared/util-
 core": ["libs/shared/util-
 core/src/index.ts"]
}
```

Production-Ready: CI/CD Integration with Jenkins

Pipeline Structure

The framework integrates with multiple Jenkins pipelines for automated, scheduled, and platform-specific test execution.

Main Pipeline

Daily scheduled run executing API and Web Portal tests.

Platform-Specific Pipelines

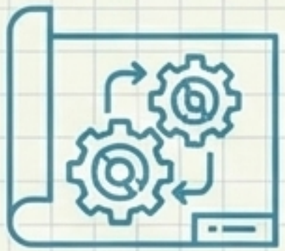
- `Jenkinsfile.macos`: Runs SAB tests on a dedicated macOS agent.
- `Jenkinsfile.windows`: Runs SAB tests on a dedicated Windows agent.

Pipeline Snippet

```
stage('API Tests') {
    steps {
        container('playwright') {
            withCredentials(...) {
                sh '''
                    cd /app
                    export PLAYWRIGHT_HTML_REPORT=out/api_html
                    npx playwright test --project="api testing"
                '''
            }
        }
    }
}
post {
    always {
        publishHTML target: [
            reportName: 'Playwright result',
            reportDir: 'out',
            reportFiles: 'api_html/index.html',
            keepAll: true
        ]
    }
}
```

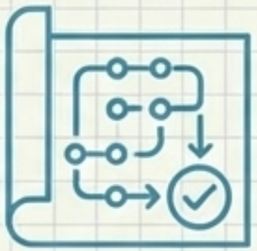
The Blueprint Realized: A Cohesive & Scalable Infrastructure

By integrating a monorepo architecture with a consistent, pattern-driven design, the framework successfully delivers a sophisticated and maintainable testing infrastructure.



Unified Architecture

Nx and shared libraries provide a single, cohesive system for all test types.



Pattern-Driven Consistency

The AAA pattern and other design principles ensure code is predictable, reusable, and easy to maintain.

Advanced Orchestration

Centralized configuration, reporting, and user management simplify complex test scenarios.

Engineered for Scale

The architecture for SAB testing demonstrates the framework's capability to solve complex, parallel execution challenges.