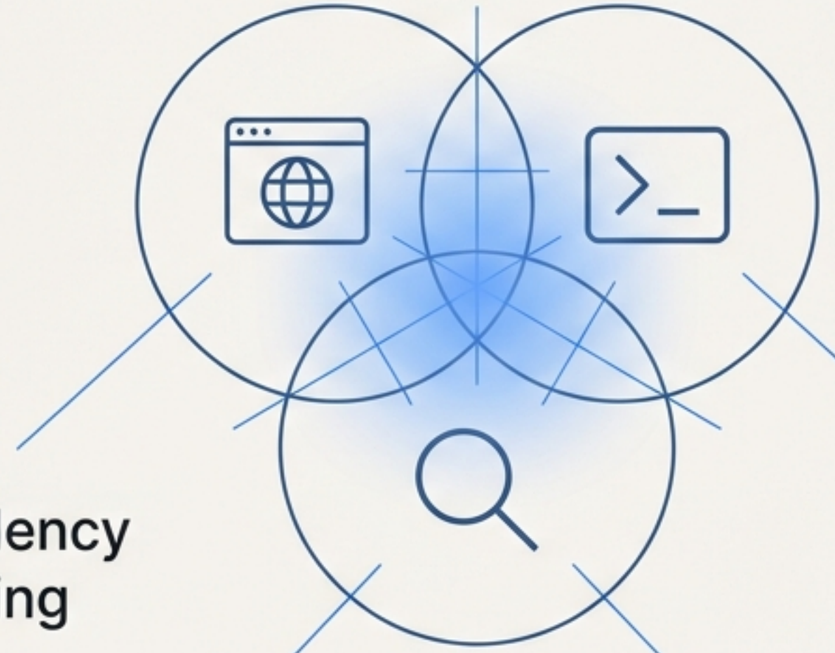
IFD Testing: One Workflow, Three Environments

A unified testing strategy for development, CI, and production.



Zero-dependency browser testing

CI/CD-ready pipelines

Live production QA capability

Surgical precision matching your code

A Single Test Suite for Every Stage of the Lifecycle

The Browser

Local development, debugging, and instant feedback with zero setup. Simply open `tests/index.html`.

The Command Line

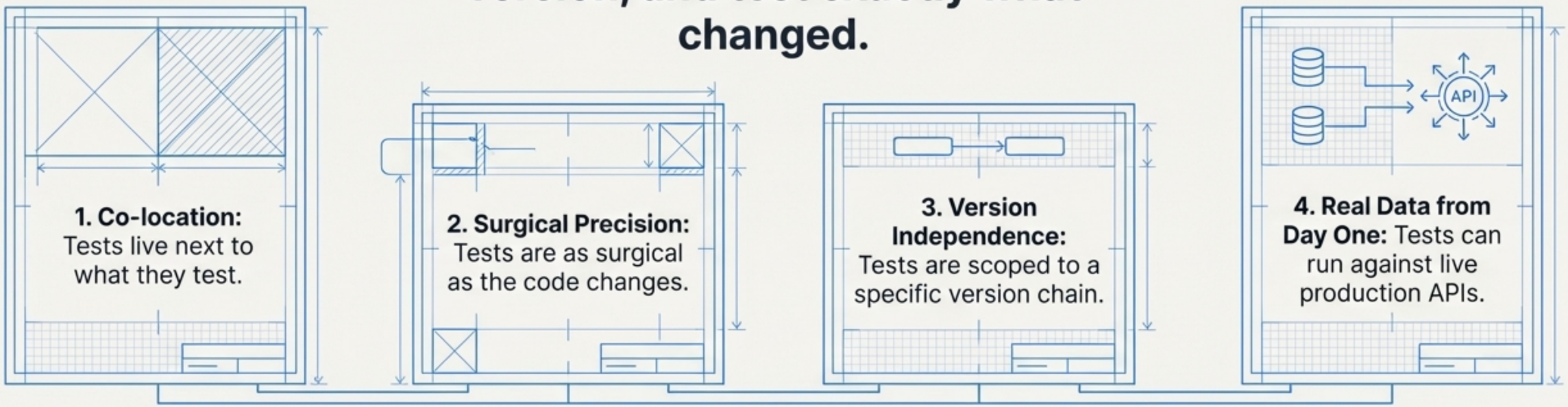
Automated CI pipelines and pre-commit hooks using Karma & ChromeHeadless. Blocks deployment on failure.

Production QA

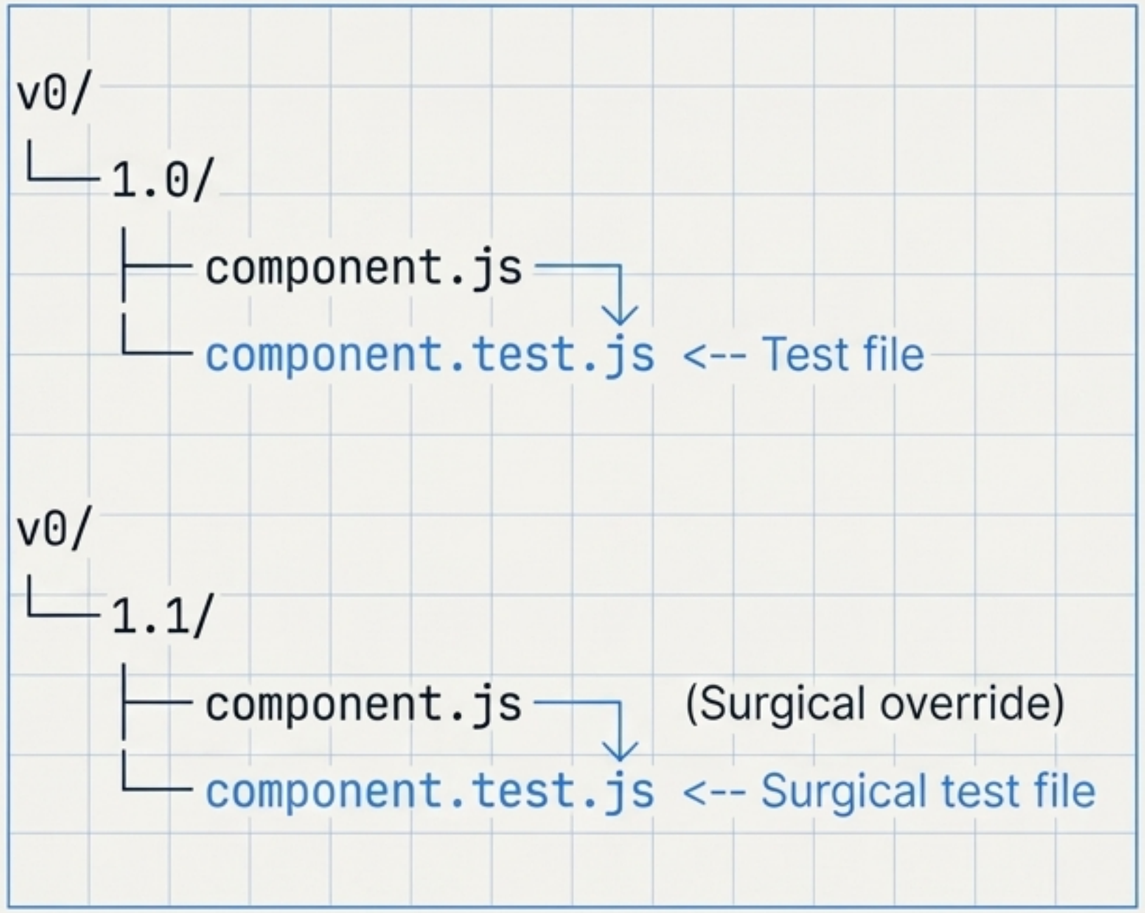
Run the *exact same* tests against a live deployment to debug user-reported issues in their native environment.

The Philosophy: Testing is an Extension of Development

Testing in IFD follows the same core principles as development. Tests live with the code they test, evolve with each version, and test exactly what changed.



Principle 1: Tests Live With The Code They Test



Why Co-location is Non-Negotiable

- Tests naturally evolve with their source code.
- Ownership is clear and unambiguous.
- Enables true version independence (v0.1.1 changes don’t break v0.1.0 tests).
- Ensures code coverage maps correctly without cross-version confusion.

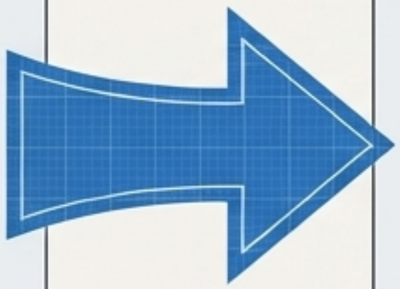
Principle 2: Surgical Tests for Surgical Changes

> **“The test file is as surgical as the source file.”**

`v0/1.1/component.js` in JetBrains Mono

```
// Only contains overridden methods
({
  methodB() { /* new logic */ },

  methodC() { /* new logic */ }
})
```



`v0/1.1/component.test.js` in JetBrains Mono

```
QUnit.test('v0.1.1 methodB override', assert => {
  // ... test logic for B
});

QUnit.test('v0.1.1 methodC override', assert => {
  // ... test logic for C
});
```

Naming Conventions

`component.js` → `component.test.js` (Unit Test)
`page.html` → `page.html.test.js` (Integration Test)

The System in Practice: Writing the Tests

The Base Test

`v0/1.0/component.test.js`

The base version's test file is comprehensive, testing the component's full public API.

```
QUnit.module('v0.1.0 component');

QUnit.test('methodA works correctly', assert => {
  // ...
});

QUnit.test('methodB works correctly', assert => {
  // ...
});

QUnit.test('methodC works correctly', assert => {
  // ...
});
```

The Surgical Test

`v0/1.1/component.test.js`

The minor version overrides `methodC`. Its test file only needs to validate the new behaviour of `methodC`.

```
QUnit.module('v0.1.1 component');

QUnit.test('methodC override has new behaviour',
assert => {
  // Test only the change
});
```

The global test runner (`tests/index.html`) discovers all `*.test.js` files and loads the full dependency chain (base + patches) before running any tests.

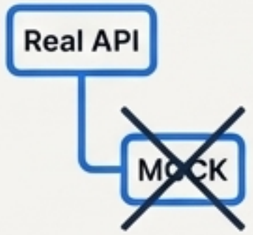
The Strategic Advantage: Running Tests in Production

Your QA team can reproduce issues in the exact environment where they occur, with zero local setup. Simply visit `https://your-app.com/tests/index.html`.

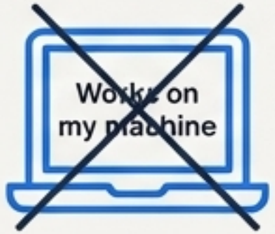
Why This Transforms QA & Debugging

Debug with Precision: Isolate browser-specific issues reported by users.

Verify Deployments: Instantly confirm a new deployment didn't introduce regressions.



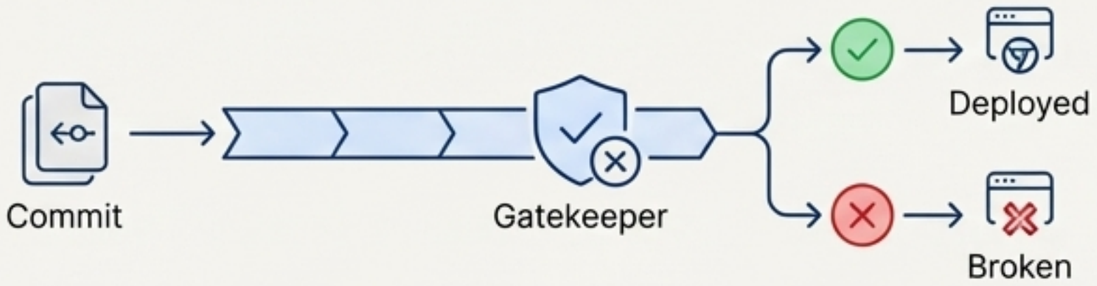
Test Against Reality: Validate against production APIs and data, not mocks.



Eliminate 'Works on My Machine': The production environment becomes the single source of truth.

Powering Automation and Real-Time Feedback

The Pipeline Guardian



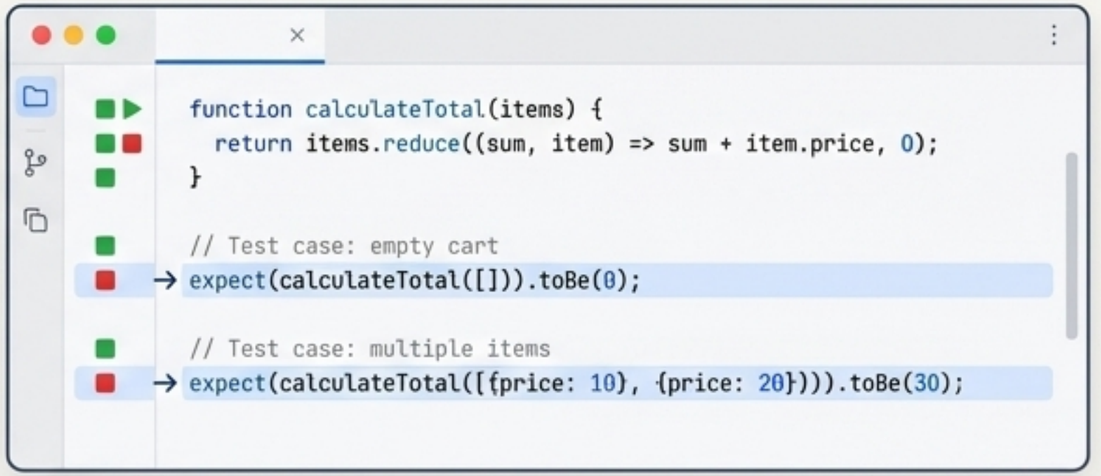
Tooling: Karma + ChromeHeadless

Trigger: `npm test`

Acts as a gatekeeper in CI pipelines and pre-commit hooks, preventing broken code from being deployed.

```
"scripts": {
  "test": "karma start --single-run"
}
```

The TDD Accelerator



Tooling: Wallaby.js

Trigger: Runs automatically on code changes.

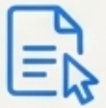
Provides instant, in-line feedback on test success/failure and code coverage directly within the IDE.

Taming the Unseen Enemy: Test State Leakage

The Problem

Tests can unknowingly pollute the global state, causing cascading failures. Common culprits include:

Mocked global functions (fetch, console.log)



Event listeners on document

Lingering timers (setTimeout)

Global variables (window.apiClient)

Critical: Prototype modifications.

The Solution

A rigorous `afterEach` cleanup routine is mandatory.

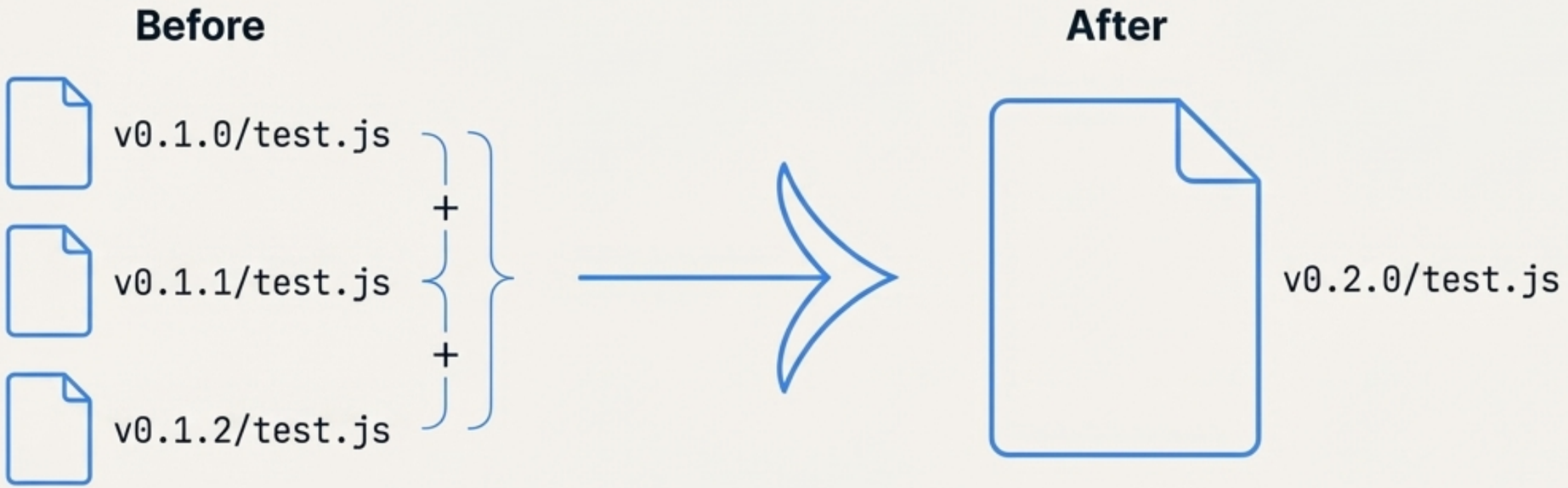
```
// Example using a helper
const { trackAndRestore } = testUtils;

QUnit.module('My Component', {
  afterEach() {
    // Cleans up all tracked changes
    trackAndRestore.restoreAll();
  }
});

QUnit.test('it does something', assert => {
  // Track a prototype change
  trackAndRestore.replace(
    String.prototype, 'trim', () => 'mocked!'
  );
  // ... test logic
});
```

The Circle of Life: Test Consolidation

When minor versions are merged into a new major version, their corresponding tests are merged alongside them.



The Process

1. Copy tests along with their source files.
2. Merge the base tests and all surgical patch tests into a new, complete test file.
3. Update `test-paths.js` to point to the new consolidated version.

The Payoff

The full, consolidated test suite passes without modification, and code coverage is unified.

IFD Testing: Quick Reference

Aspect	IFD Testing Approach
Test Location	Co-located with source
Surgical Override	Surgical test (only changed methods)
Base Component	Full test (all methods)
Unit Test Naming	`source.test.js`
Integration Test Naming	`page.html.test.js`
Loading Strategy	Full chain: base + patches in order
Framework	QUnit (no build step)
Browser Runner	`tests/index.html` (works in production!)
CI Runner	Karma + ChromeHeadless
IDE Integration	Wallaby.js
Isolation	`afterEach` cleanup (including prototypes)

Critical Anti-Patterns to Avoid

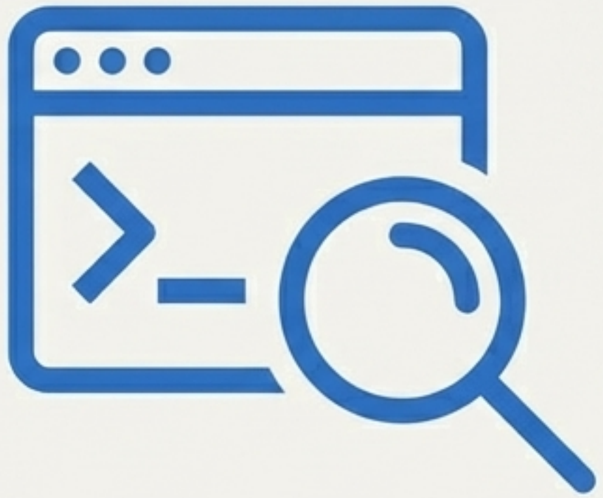
Adopting the system is easy. Mastering it means avoiding these common pitfalls.

Scoping & Structure

- Don't create full test files for surgical overrides.
- Don't put tests in a central folder separate from source code.
- Don't share tests across major versions.

Isolation & State Management

- Don't forget to restore mocked globals in `afterEach`.
- Don't forget prototype modifications are global; they *must* be restored.
- Don't add event listeners without tracking them for cleanup.



This is IFD Testing.

Zero-dependency browser testing, CI-ready pipelines, production QA capability, and surgical precision matching your code changes.