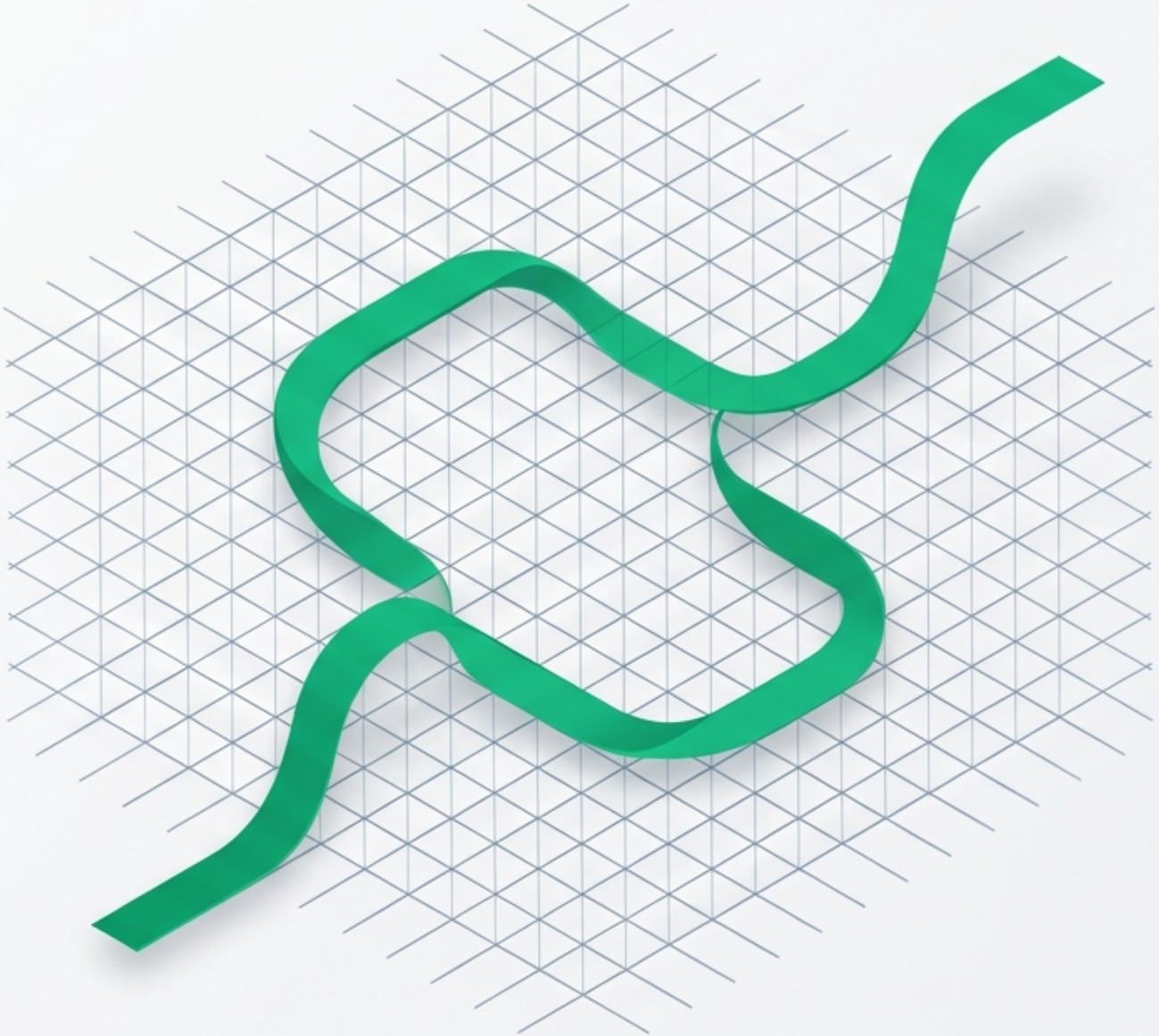
Code with Confidence

A Sustainable Workflow for Test-Driven Development



"Code without tests is broken by design." – Jacob Kaplan-Moss

The Fear of a Changing Codebase

Untested code creates a culture of fear that cripples velocity and quality.

Brittle Systems

Small changes can cause unexpected, cascading failures in distant parts of the application.

Technical Debt Accretes

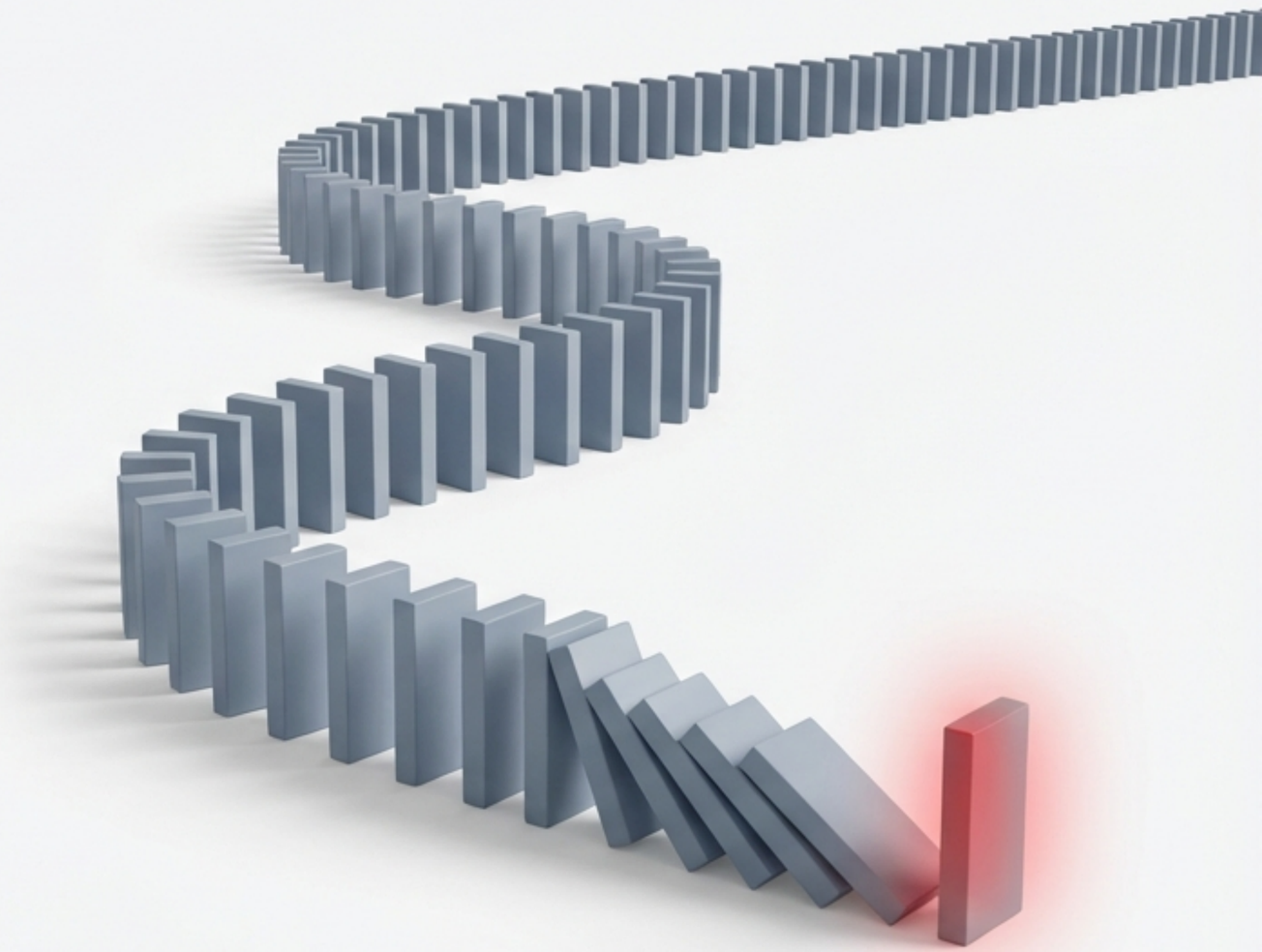
Developers avoid cleaning up code because they're afraid to break what's currently “working,” applying band-aid fixes instead of addressing root causes.

Slow, Manual Verification

Progress is gated by slow, repetitive manual checks, which are unreliable and drain developer energy.

The Result

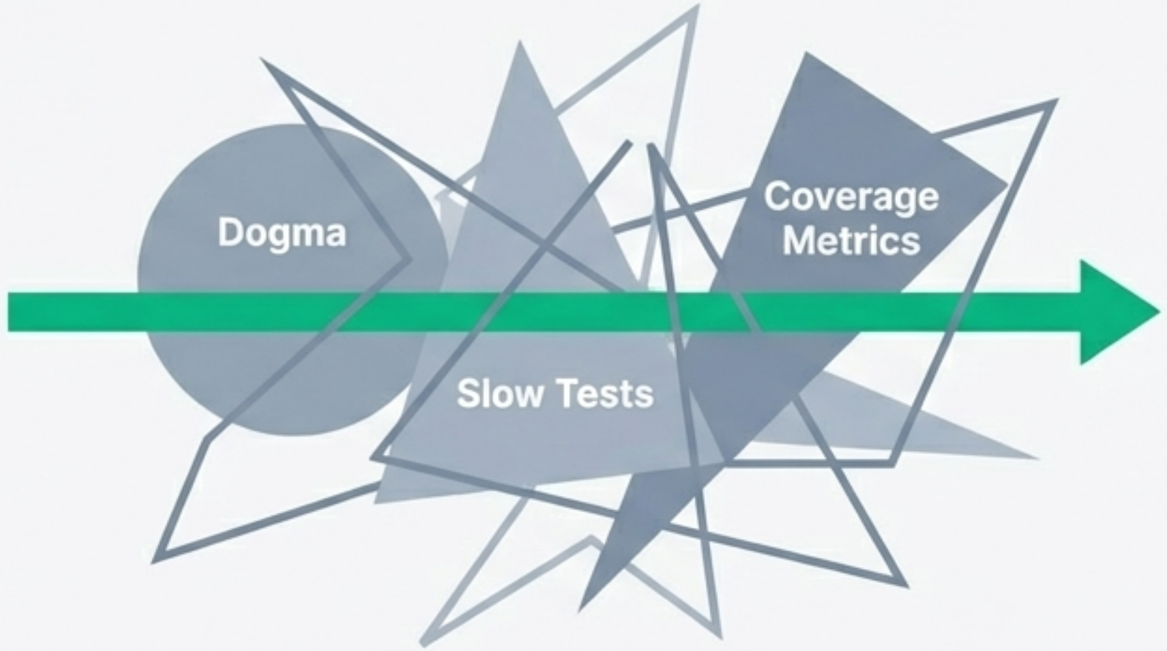
A downward spiral where the codebase becomes more fragile and development slows down over time. Martin Fowler notes these “old codebases [become] terrifying places.”



Why ‘TDD’ Can Feel Like a Burden

The principles of TDD are powerful, but its name has been “corrupted” by rigid, impractical applications that miss the point.

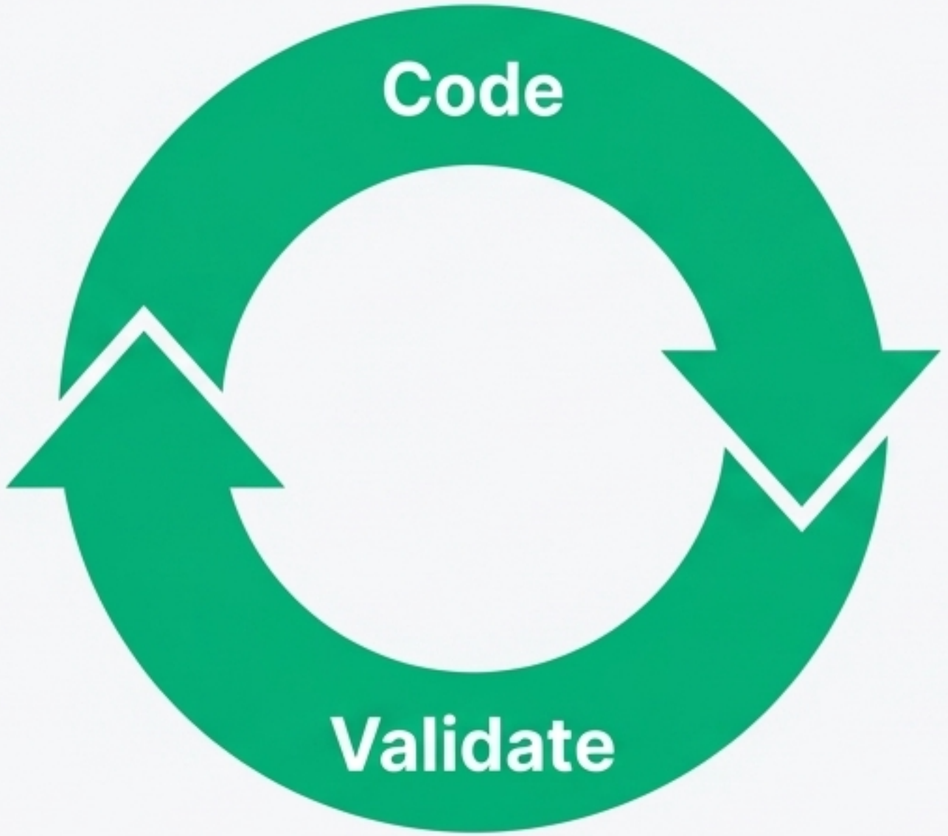
- **Dogmatic ‘Red-Green-Refactor’**
  Forcing a failing test for every single line of code can feel unnatural and disruptive to the creative flow.
- **Slow and Cumbersome Test Suites**
  When tests are slow or require complex setup, running them becomes a context-switching tax, breaking a developer's focus.
- **Testing as an Afterthought**
  Mandates to “add tests” after the fact often lead to low-value, brittle tests written just to satisfy a metric.



The Consequence: Testing is seen as a chore, not a tool. This presentation offers a sustainable alternative that integrates testing into the natural rhythm of coding.

The Foundation: A Fast Feedback Loop

Minimise the distance between writing code and validating it. The faster the feedback, the less context switching is required, and the more a developer can stay in a state of flow.



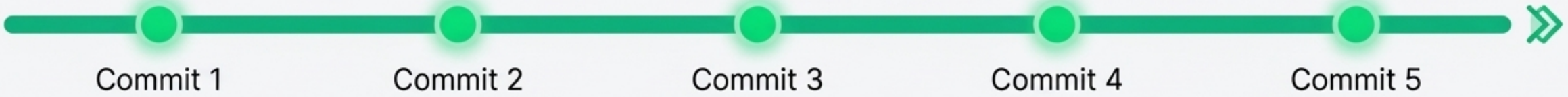
Key Practices:

- → **Speed is Paramount:** Aim for tests that run in milliseconds (e.g., under ~200ms per unit test).
- → **Keep Tests Close to Code:** Write automated unit or lightweight integration tests that exercise code directly, avoiding slow, external dependencies.
- → **Integrate into Your Workflow:** Use IDE plugins or keystrokes to run relevant tests instantly, making it as natural as saving a file.

"The faster your feedback loop, the less need there is for context switching — and the faster you'll be able to ship features and bug fixes."
— PythonSpeed

Evolving TDD: The 'Always-Be-Passing' Philosophy

Core Idea: Instead of starting with a failing test in the main branch, the goal is to keep the test suite green at all times through small, incremental steps.



How it Works in Practice

1. Start with a baseline working state and a passing test.
2. Write a new test structure for the next behaviour, initially confirming the current state (it passes).
3. Implement the new behaviour in the code.
4. Update the test's assertion to reflect the new, correct behaviour, making it pass again.

The Benefit

The main branch is always in a valid, deployable state. Every commit is a working snapshot. This isn't “test-last”; it's a rapid, micro-scale dance between coding and testing.

> “The important point is that you have the tests, not how you got to them.”
> — Martin Fowler

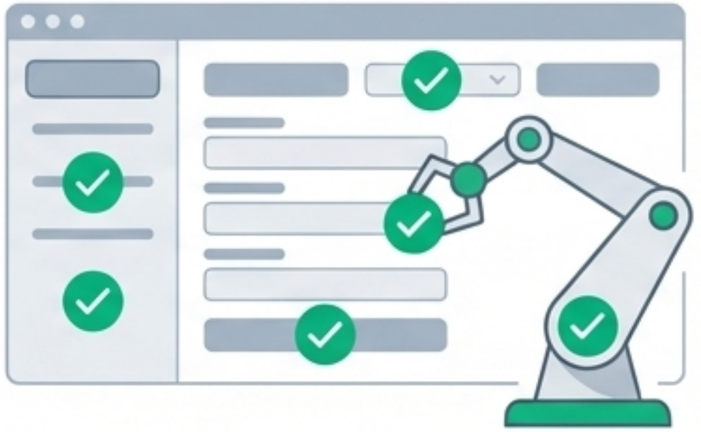
Leave a Trail of Automated Checks

The Rule

If you manually verify something (click a button, call an API), immediately turn that action into an automated test. Nothing is verified only once.

Manual Verification

Automated Knowledge



Why This is Transformative:

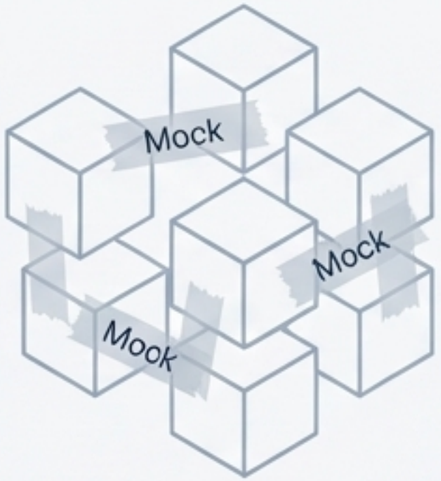
- **Prevents Regression and Drift**: It **immunises the codebase** against future changes breaking this behaviour. You don't rely on human memory.
- **Forces Testable Design**: Difficulty in writing a test is a **signal of a design flaw**. It pushes you to build more modular, maintainable code.

Mantra: If you make a code change and no existing test fails, you either made a no-op change or you lack coverage for that scenario.

Prefer Real Code Over Mocks

Core Principle: Use mocks and stubs judiciously, primarily for truly external dependencies. For your application's internal logic, test real components working together.

Brittle Tests with Mocks



Robust Tests with Real Components

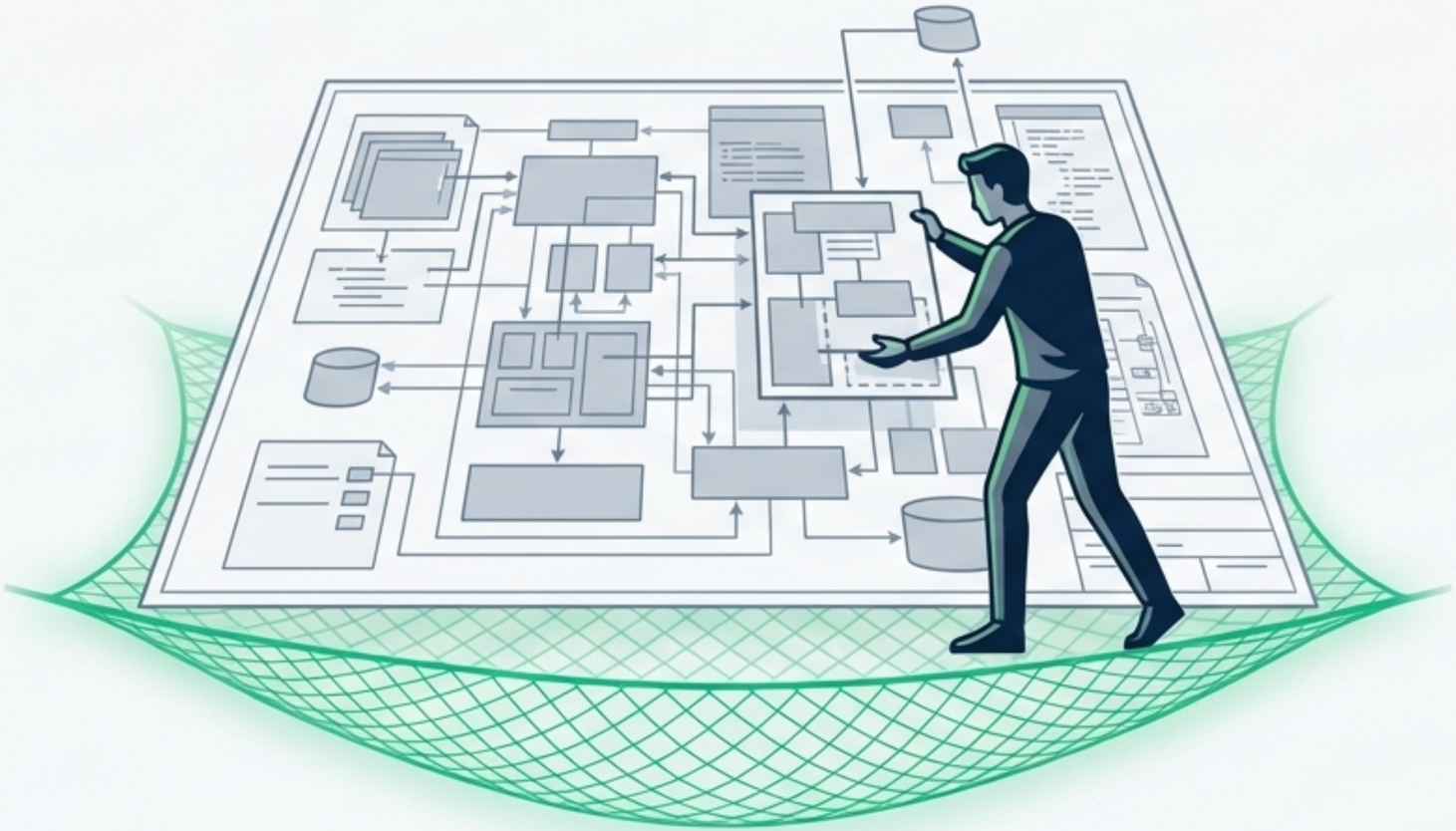
The Rationale

- **Tests for Reality:** Validates that components actually integrate correctly, not just that a method was called. This catches issues that isolated unit tests miss.
- **Enables Refactoring:** Tests are coupled to behaviour, not implementation. You can refactor internal collaboration, and as long as the final outcome is the same, the test will still pass.

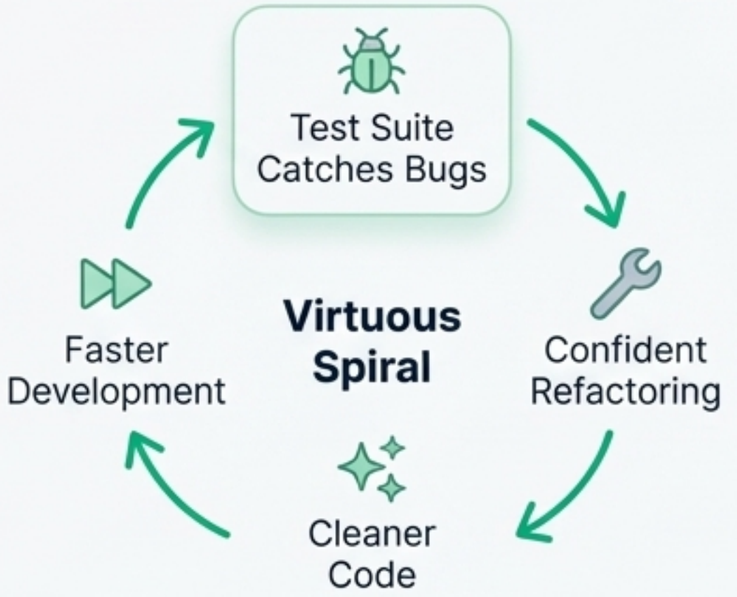
> “Stop mocking so much stuff... most of the time you can avoid mocking and you’ll be better for it.” — Kent C. Dodds

The Payoff: Refactor with Confidence

A comprehensive, high-coverage test suite is a safety net that transforms refactoring from a high-risk activity into a routine practice.

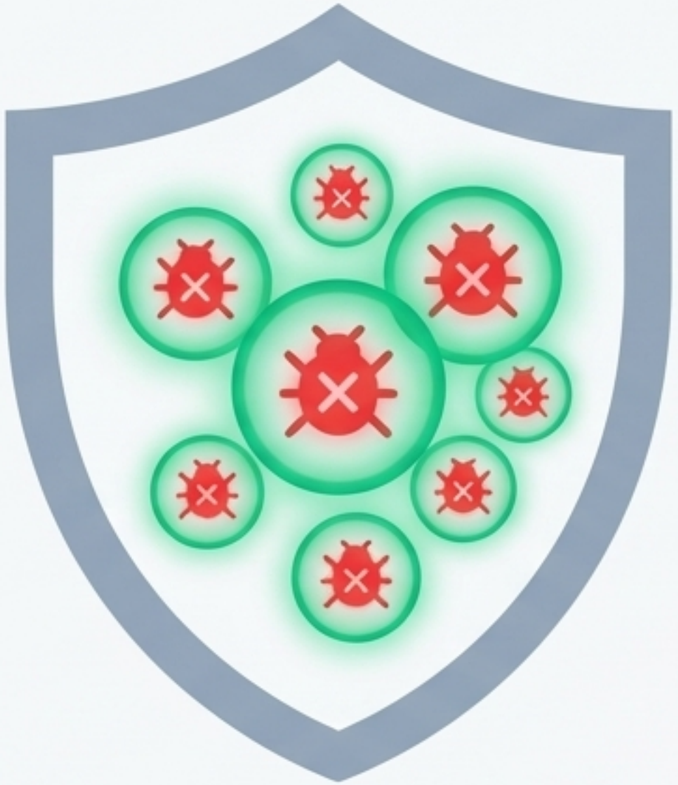


Comprehensive Test Suite



"With that safety net, you can spend time keeping the code in good shape, and end up in a virtuous spiral where you get steadily faster at adding new features." — Martin Fowler

Every Bug Immunises the Codebase



****The Process****

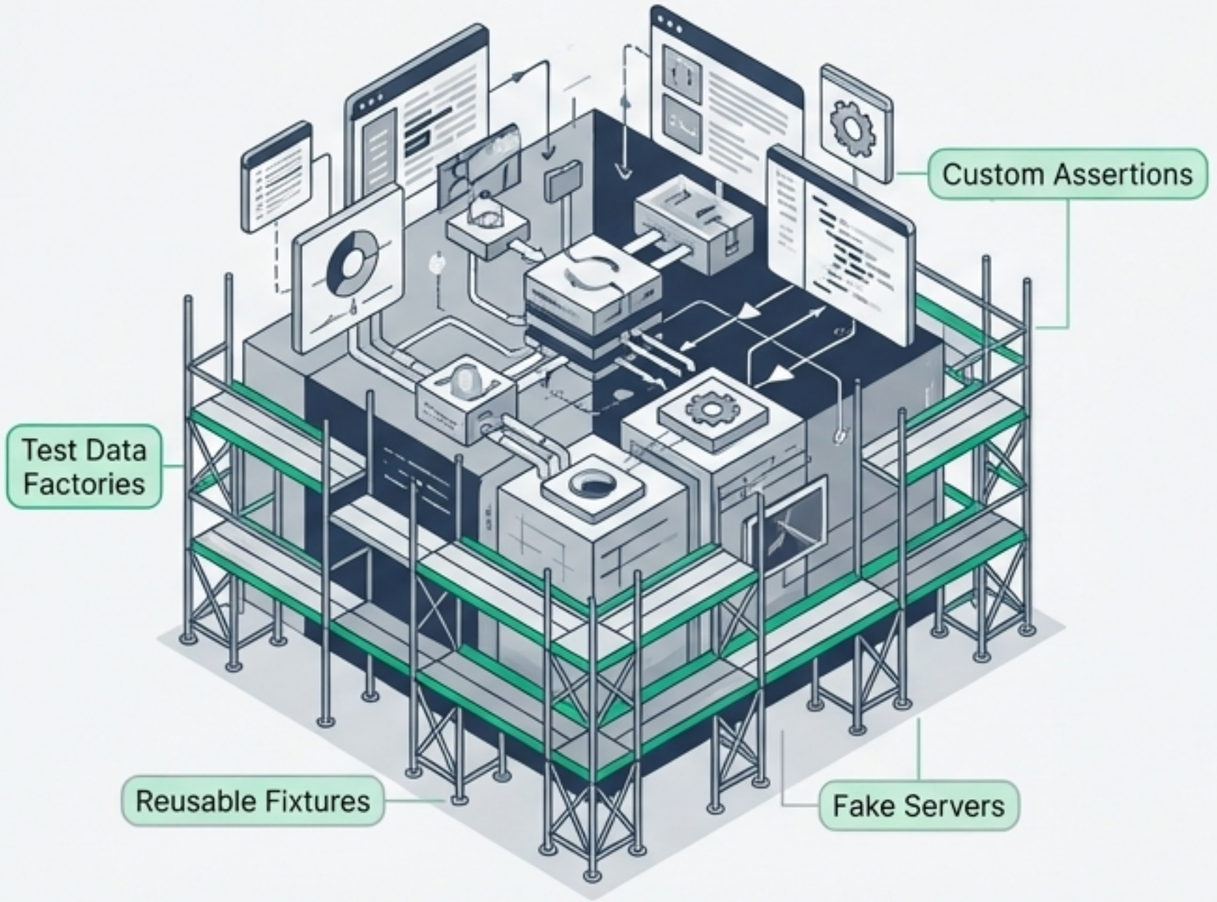
When a bug is discovered, the first step is always to write a test that reproduces it.

1. ****Write the Bug Test****: Create an automated test that fails because the bug is present. This clarifies the exact conditions of the problem.
2. ****Fix the Code****: Implement the fix.
3. ****Watch the Test Pass****: The bug test now passes, confirming the fix is effective.
4. ****Keep the Test Forever****: The 'bug test' graduates to become a permanent 'regression test,' ensuring this specific bug never reappears.

****The Benefit****: The test suite grows stronger with every defect found, systematically eliminating entire classes of future errors. Each bug fix adds another layer to the system's "immune record."

Test Infrastructure is a First-Class Citizen

The ease of writing good tests depends on a robust foundation of tools and helpers. Building this infrastructure is a key responsibility of senior engineers and tech leads.



Examples of High-Value Infrastructure:

- Test data factories and builders.
- Custom assertion helpers for complex data structures.
- Reusable `pytest` fixtures for managing state.
- Fake servers that mimic third-party APIs.

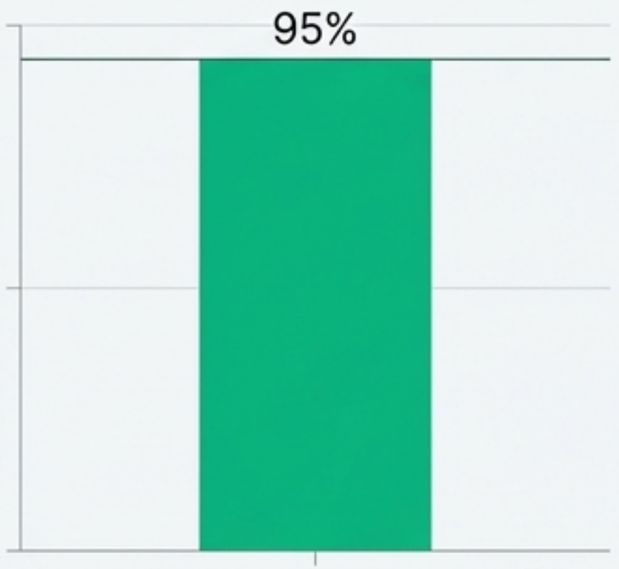
The Goal: Make doing the right thing (writing a test) the path of least resistance. When testing is easy and fast, it becomes the natural way to work. As one practitioner noted, it's done not from mandate, but "because it is the only way that I can work."

High Coverage is a Side Effect, Not a Goal

Teams following this workflow consistently achieve **90-100%** code coverage **without** explicitly aiming for it.

Why it Happens Naturally:

- Every new feature or behaviour gets a test.
- Every bug fix adds a regression test.
- The 'leave no manual check behind' rule fills in the gaps.



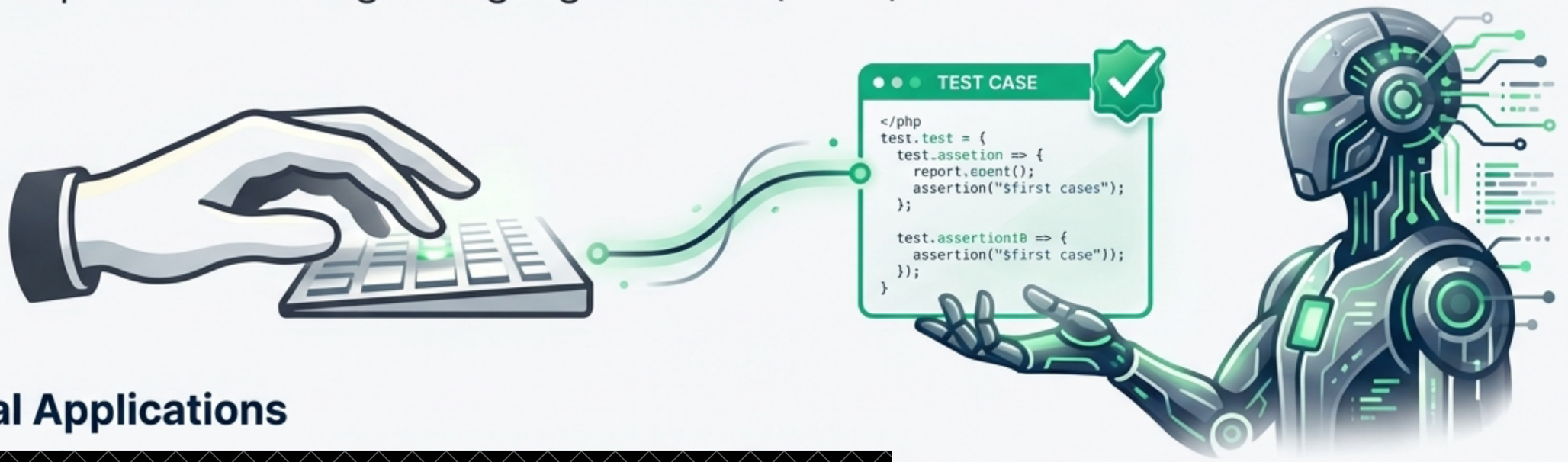
The result of a disciplined process, not the target.

The Qualitative Difference:

- **Confident Onboarding**: New developers can make changes safely, guided by the test suite.
- **Fearless Upgrades**: Major changes (like swapping a library) become verifiable and manageable.
- **A Reliable Quality Gate**: Most bugs are caught by CI long before they reach production.

Amplifying the Workflow with AI Assistants

The discipline of writing many small, focused tests pairs perfectly with the capabilities of Large Language Models (LLMs).



Practical Applications

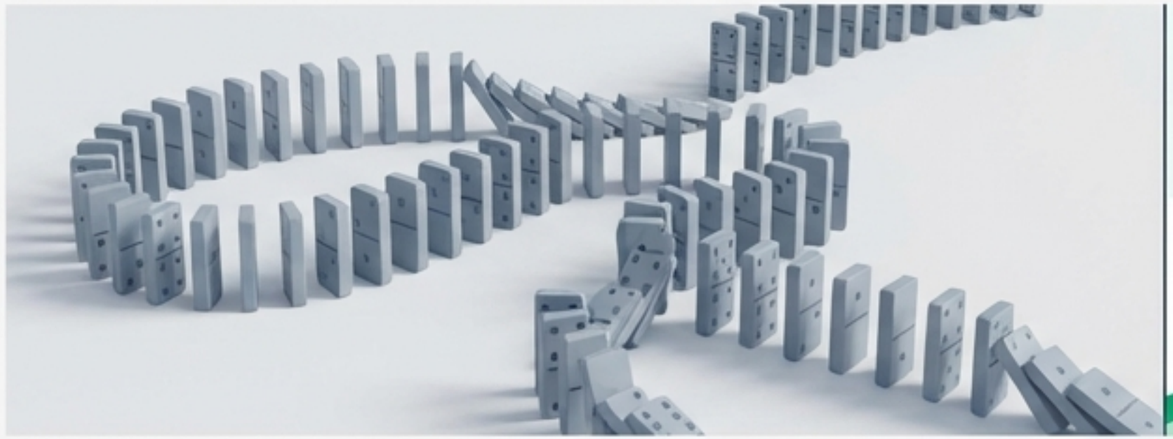
Synergy: [redacted].

- **Generate Boilerplate:** Ask an AI to generate initial test cases for a new function, which the developer then refines. One developer took a module from 0 to 100% coverage in minutes this way.
- **"Review Code via the Tests":** When an AI generates code, ask it to also generate the tests. This provides a clear specification and validation of the AI's output.
- **Automate Test Maintenance:** Use AI to help update assertions across many tests after a significant, intentional refactoring.

The Synergy: AI handles the quantity and repetitive tasks, while the developer ensures the quality and correctness of the tests.

From Fear to Flow

Fear of Change



Freedom to Innovate

Fear of Change becomes **Freedom to Innovate**.
Slow, Manual Checks become **Instant, Automated Confidence**.
Brittle Code becomes **Resilient, Self-Testing Systems**.

The Core Benefit

It creates a psychologically satisfying rhythm where you build, get immediate confirmation it works, and move forward. The comprehensive test suite acts as your trusted “bug detector.”

The mandate isn’t “write tests because you must.”
It’s “write tests because it makes coding more fun, reliable, and fast in the long run.”