ChatGPT

Sustainable Test-Driven Development (Python-Focused, Language-Agnostic)

Introduction

Test-driven development (TDD) is a renowned practice, but its name and application have evolved over time – some say even "corrupted" or misunderstood in negative ways. This guide presents a refined workflow for writing tests that drive development, focusing on Python for examples while being language-agnostic (the same approach has been used in JavaScript, .NET, Java, etc.). The goal is to make writing tests an integral, natural part of coding – **not** an afterthought or a burdensome mandate. As one expert famously put it, *"Code without tests is broken by design."* [1] In other words, every significant piece of code should come with automated tests to validate it. This workflow shows how to achieve that in a sustainable manner, yielding consistently high coverage (often 90-100%) and enabling fearless refactoring and maintenance.

Fast Feedback Loop and Keeping Tests Close to Code

A cornerstone of this approach is minimizing the "distance" (in time and effort) between writing code and validating it with tests. The sooner you get feedback on a code change, the better. Ideally, you should be able to run relevant tests in milliseconds and immediately know if your latest edit works [2]. Fast tests keep the developer in flow by avoiding costly context switches. Conversely, if tests take too long or require cumbersome steps (like manual UI clicks or lengthy integration setups), the developer's focus is broken. Studies on developer productivity note that *"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."* [3]

To achieve such fast feedback, tests should be as close to the code as possible. This means writing **automated unit or integration tests** that exercise the code directly (within the same process, without external dependencies when feasible). For example, if working in Python, one might use `pytest` to run tests on functions or classes directly, rather than verifying behavior through a web UI or an external environment. The further the test is from the code (imagine having to deploy and click through a browser UI to see if a function works), the more overhead and potential context-switching tax on the developer. Keeping tests near ensures they can be run frequently (many times per minute), enabling a truly rapid development cycle.

Key practices for fast feedback:

- Write tests at the lowest practical level that still gives confidence (often unit tests or lightweight integration tests) for the code you are changing. Avoid requiring a full application spin-up for each little code change.
- Ensure tests run in sub-second time (a good rule of thumb is under ~200ms each for unit tests [2]). This often entails using in-memory or fake data stores instead of real databases, and avoiding unnecessary network calls in tests.
- Integrate test running into your development workflow (many IDEs and editors can run tests on file save or with a single keystroke). The goal is that running tests becomes as natural as hitting "build" or refreshing a page.

- **Minimize context switches:** If running tests is quick and automatic, you don't need to distract yourself with other tasks while waiting. A slow test suite encourages multitasking (since you "might as well check email while it runs"), but a fast, focused test run keeps you engaged with the code at hand.

Using Tests to Drive Development (Always-Be-Passing Philosophy)

In this workflow, tests truly drive development – but not necessarily in the orthodox TDD way of *starting* with a failing test. Instead, the emphasis is on always having a working (passing) test suite as you incrementally build features. Development becomes a tight loop of writing a bit of code, writing or adjusting a test for that code, and ensuring tests pass before proceeding. This is done in very small steps, so at any given moment, the codebase is in a valid state with respect to tests.

How is this different from traditional TDD? Traditional TDD is often described by the *"red, green, refactor"* cycle – write a test first (red, failing), then write code to make it pass (green), then refactor. The twist here is a preference for keeping tests green at all times in the main code branch. In practice, that means you might sometimes write a quick initial implementation (or even a temporary stub) so that a new test can pass, and then iteratively evolve both code and tests. Alternatively, you might write a test *after* writing a small piece of logic, rather than before, to verify the behavior. Both approaches are fine as long as you don't wander too far without a test. The critical part is that you **use tests as the driver**: whenever you make a code change, you immediately run a test (or a small test suite) to verify that change. This rapid dance between code and test is what guides the development. In the IDE, you might literally have your code editor on the top pane and a test file on the bottom pane, constantly alternating between them.

Always-be-passing rule: One practical rule followed is that the version control's main branch (or any commit that goes to Continuous Integration) should not contain deliberately failing tests. You're not leaving "red" tests in the shared code base. If a test is written, it should reflect either existing behavior or the intended behavior such that you can update the code and test in tandem to keep things passing. In other words, you achieve the same final outcome (tests that validate new functionality), but you avoid breaking the build for extended periods. This has a benefit of keeping the CI pipeline green and deployable. It also means every commit is a working snapshot – you could release software at any point without worrying about incomplete features guarded by failing tests.

In practice, the workflow might look like:

1. **Start with a baseline working state** – perhaps the function returns a default or placeholder value that satisfies an initial trivial test.
2. **Write a test for a new requirement or behavior**. Initially this test might just confirm the current behavior (so it passes). This step is about setting up the structure.
3. **Implement the new behavior in code**. This likely causes the test to fail (since the expected outcome in the test can now be changed to the new expected result).
4. **Update the test's assertion** to the new expected result, making the test pass again.
5. Repeat for the next small piece of functionality.

This approach ensures that at each step, you have a passing test suite to verify your progress. It's a slight inversion of the typical TDD cycle (where the test fails first), but it achieves a similar effect with potentially less disruption to the CI pipeline. In essence, you're never far from a green state. The tests you create along the way act as an **evolving specification** of the code's behavior. In fact, the history of the test file can tell the story of how the function or module evolved over time. Each test captures a scenario or edge case that was considered. Over time, this leads to very high coverage of possible workflows, not by aiming for coverage per se, but simply as a byproduct of writing a test for every change or new behavior.

Why always keep tests passing? It enforces discipline to finish each piece of functionality before moving on. It also means you can integrate or commit at will, since you're not introducing known failures. Some developers fear that writing tests only after code (the "test-last" approach) could allow bad habits, but done in these small increments, it's effectively the same as test-first on a micro scale. Martin Fowler distinguishes between *self-testing code* and the act of TDD itself, noting that *you can also produce self-testing code by writing tests after writing code – although you can't consider your work to be done until you have the tests (and they pass). The important point is that you have the tests, not how you got to them.* [4] This underscores that the ultimate goal is a robust test suite accompanying the code; whether you wrote the test 3 minutes before or after the function is less important than the fact that you wrote it at all, and promptly.

Leaving a Trail of Automated Checks (No Manual Step Left Behind)

Another principle of this workflow is that **anytime you manually verify something, you immediately turn that into an automated test**. If a developer finds themselves running the application and clicking buttons or calling an API to see if their last code change works, that's a sign to pause and instead write a test to perform that verification. The idea is that nothing is verified only in a one-off, ephemeral way. Every check is committed to the test suite. This habit has two major benefits:

- **Preventing regression and drift**: If you only manually test a behavior once, there's no guarantee someone will remember to test it again in the future. By automating it, you ensure that the behavior will be continuously checked in every future run of the test suite. This addresses the issue that often *"the next time, [the developer is] not always going to check everything"* – you don't rely on memory or manual QA for that scenario ever again.
- **Forcing testability and good design**: If a feature is hard to test programmatically, that friction often indicates a design issue (perhaps too much logic in the UI, or insufficient separation of concerns). Committing to turning all checks into tests motivates you to create testing hooks, dependency injection points, or better modularization so that everything *can* be tested easily. In fact, building a rich test **framework** or helper library for your project is time well spent. The test framework (utilities for creating test data, resetting state, simulating external systems, etc.) can be as important as the product code itself. It's the scaffolding that allows developers to work efficiently and correctly. Many times, **senior developers or engineering leaders** take the lead in developing these testing tools and patterns, because they see the long-term benefit. This is a healthy inversion of the anti-pattern where junior devs write tests and senior devs only write code – instead, everyone, including the most experienced engineers, contributes to making the test suite comprehensive and easy to work with.

Over time, following this "leave no manual check behind" rule leads to extremely high coverage of behaviors. You'll notice that whenever you make a change in the code, *some* test will fail (if not, that's a hint you may have untested logic). In fact, a mantra here is: **if you make a code change and no existing test fails, you either made a no-op change or you don't have a test covering that scenario.** This approach naturally encourages writing tests for all code paths, because if something isn't tested, it feels like an uncomfortable blind spot. It aligns with the view that *"no programming episode is complete without working code and the tests to keep it working."* [5] In practice, consistently following this workflow yields code coverage numbers in the upper 90% or even 100% on critical modules – not by chasing coverage metrics, but as a side effect of testing everything that gets built.

Importantly, the focus is not only on line coverage but on **scenario coverage** – ensuring that different combinations of inputs, edge cases, and usage patterns are covered by tests. For instance, you might have multiple tests for a single function, each exploring a different branch or edge condition. Together they document the intended behavior thoroughly. This depth of testing brings a high degree of confidence in the system.

Prefer Real Code Over Mocks (Integration Where Possible)

A notable aspect of the described testing style is a bias against excessive use of mocks and stubs. Whenever feasible, tests are run on the real code units working together, not isolated with heavy mocking. The rationale is that tests should validate actual functionality and catch integration issues, not just verify that a method was called. Over-reliance on mocks can lead to tests that pass but the system still breaks in reality because the interactions between components weren't truly exercised. As testing expert Kent C. Dodds advises, *"stop mocking so much stuff... most of the time you can avoid mocking and you'll be better for it."* [6]

This doesn't mean *never* use mocks – but use them judiciously, only when interacting with something truly external or impractical in tests (like a payment gateway or a network call in a unit test). For the core logic of your application, strive to run it as a whole within your test harness. For example, if you're testing a Python class that normally would call a couple of internal helper classes, prefer to instantiate the real helpers in tests rather than mocking them out, unless they do something like hitting a real database. This way you catch issues in the interactions and ensure the pieces work together properly. This approach aligns with the notion of writing more **integration tests** (tests that cover multiple units together in a realistic way) for higher confidence. It also dovetails with the goal of fast tests: design your code such that even when using real components, you can run them quickly (e.g., use an in-memory SQLite database for tests instead of a production database server, so your test is both real and fast).

By keeping most tests close to real usage, you also make refactoring easier – since your tests aren't tightly coupled to implementation details or mocks. If you change how two classes collaborate (but not the external behavior), an overly mocked test suite might break (because it was expecting certain calls), whereas an integration-style test will continue to pass as long as the end observable behavior is correct. Well-designed tests *"should very rarely have to change when you refactor code."* [7] [6] The focus stays on *what* the code should do, not *how* it does it, which is a more robust way to test.

Continuous Refactoring with Confidence

One of the biggest payoffs of having a robust, high-coverage test suite is the freedom to refactor mercilessly. Codebases with low test coverage tend to accrete technical debt because developers are afraid to change working code – they don't know if a small change will break something else, so they apply band-aid fixes or avoid improvements altogether. In contrast, a codebase with comprehensive tests gives developers the confidence to clean up and optimize code aggressively. Every time you run the tests, you have a "bug detector" watching for any unintended side effects of your changes [8] [9].

When following this testing-first (or testing-alongside) workflow, any time you consider refactoring, you can do so with the safety net of tests. If you inadvertently break something, one of the many tests will alert you immediately. Martin Fowler describes this scenario: with self-testing code, *"should you make a mistake (or rather 'when I make a mistake') the bug detector will go off and you can quickly recover and* *continue. 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."* [10] In other words, high test coverage enables a positive feedback loop: you refactor boldly, which keeps the code cleaner, which in turn makes it easier to add features or more tests, and so on.

Another advantage of having tests for virtually everything is that you can pinpoint the "blast radius" of a change. Suppose you change a core function that is used in 10 different contexts; ideally, you have tests for all those contexts, and many of them might fail after your change. Rather than being annoyed by broken tests, this is incredibly valuable information – it shows you exactly which parts of the system are affected. You might even decide a planned change is too risky if it causes widespread breakage, and choose an alternative approach. Or, if the breakage is intentional and desired, you now have a to-do list of test expectations to update to align with the new correct behavior. In effect, the tests give you immediate impact analysis for any significant code modification.

Fix problems at the root, not with hacks: Because the tests make it clear what the symptoms of a bug or design flaw are across the system, a developer can confidently fix an issue in the proper place in the code hierarchy. There's less temptation to just patch around a bug in a single spot (which might be easier but not clean), because you can verify that fixing it at its origin will solve all the higher-level failures. You can run the full test suite and see that the bug is truly gone everywhere. This approach fosters better architecture, since you aren't piling up workarounds – you're solving issues where they should be solved, guided by the tests.

In summary, strong test coverage turns fear into freedom. Refactoring and continuous improvement of the codebase become routine, not something to dread. Teams with this culture treat any new bug as not only a flaw in the code but also a flaw in the tests (because the bug escaped in the first place). They then improve both the code and the tests. In fact, a common rule is to **always write a test for a bug that was found** (before fixing it), which we'll discuss next.

Writing Bug Reproduction and Regression Tests

Whenever a bug is reported or a defect is discovered, the first response in this methodology is: **write a test that reproduces the bug.** This is sometimes called a "bug test" or a characterization test. The idea is to capture the incorrect behavior in an automated test case, which initially will **fail** (since the bug is present). Writing the test first forces you to think about how to trigger the issue in a controlled way. It also ensures you don't "fix and forget" – by the time you fix the bug, you'll have a test that proves the fix and will prevent that bug from reappearing in the future (hence becoming a *regression test* once it passes with the fix in place).

Fowler describes this practice: *"The usual reaction of a team using self-testing code is to first write a test that exposes the bug, and only then to try to fix it. Often writing this test may really be a series of tests that gradually narrow the scope until you reach a unit test that triggers the bug. ... [This ensures] once the bug is fixed it stays fixed."* [11] Let's unpack that: - **Replicate the bug at a high level:** Start by writing a test at the highest level where the bug is observed (e.g. calling a public API or simulating a user action that fails). This test should capture the current faulty behavior (for instance, it might assert that the bug condition is present, or it might just fail until the bug is fixed). - **Narrow down with additional tests if needed:** Sometimes, the cause of a bug is not obvious from the top-level failure. In such cases, write more tests for the lower-level components involved in the bug. For example, if a web API returns the wrong result due to a bug deep in the logic, you might write a test for the middle layer and one directly for the function that has the flaw. Each test moves closer to the root cause, confirming which layer the defect originates from. - By doing this, you might end up with a *chain of tests* for one issue – from an end-to-end test, to an integration test, down to a specific unit test. These tests document how the bug manifests at different layers.

Once you have identified the root cause, you can decide the best place to fix it. Perhaps fixing it in a deeper layer is ideal, even if that means changing the behavior for several components. When you apply the fix, some of the higher-level bug tests will start **failing** (because the bug condition they were asserting is no longer happening – a good thing!). At that point, you update those tests to assert the correct behavior instead of the bug. Those tests thus graduate from “bug tests” (capturing a known incorrect behavior) to **regression tests** (verifying the bug is gone and stays gone). Any additional tests that were written on the side (e.g., testing intermediate layers) can either be kept as documentation or removed, depending on whether they still provide unique value. Often they are kept, as they might be useful for catching similar issues or as documentation of how the components should behave.

This approach of writing bug tests has several advantages: - It **decouples the act of understanding the problem from fixing the problem**. By writing a test, you are forced to clarify what the expected vs. actual behavior is, before diving into the code change. This often saves time because you might realize the issue is different than it appeared, or identify related edge cases in the process of writing the test. - It **improves your test suite**. If a bug happened in an area that lacked a test, now you’ve added one (or more) in that area – preventing future similar bugs. The team can also reflect on why the bug wasn’t caught and add additional tests if needed for similar scenarios (adopting the attitude that any bug implies something was missing in the tests [12]). - It gives a safety net while fixing: you can run the new test and see it fail (red), then apply the fix and watch it pass (green), confident that the test is actually exercising the right code path. This is much safer than attempting a fix and manually testing the app in an ad-hoc way to see if it’s resolved, which might miss nuances.

In practice, an example might be: - Bug: “Users get an error when saving a profile with a certain character in their name.” - Write an automated test (perhaps at the API level or even a unit test on the validation function) that simulates that input and asserts that an error occurs (this test fails, confirming the bug is reproducible in code). - If needed, write deeper tests, e.g., directly call the validation utility with the problematic input to confirm exactly where it breaks. - Fix the code (e.g., adjust the validation to allow that character). - Now run the tests: the bug test at the API level should now fail if it was asserting an error (because no error occurs now). Flip the expectation – the test should assert **no** error occurs with that input. It now passes. Lower-level tests might not need flipping if they were just detecting an exception; they will simply pass now if the exception is gone. - Verify all tests pass, and keep those tests in the suite forever. The bug will not sneak back undetected.

This discipline of writing regression tests for every bug ensures that your test suite only grows stronger with time. Each bug adds one more case the suite checks, acting like an immunization record for the codebase.

Building a Robust Test Infrastructure (Role of Senior Devs)

To enable all of the above practices, investing in a solid test infrastructure is key. By *test infrastructure*, we mean the supporting code and setup that makes writing and running tests easy. This can include: - Utility functions or classes for setting up test data (e.g., factories or builders that create objects with default valid data, so tests can quickly get a valid object and then tweak specifics relevant to the test). - Helpers for common assertions (for example, a custom assertion to compare complex data structures, or to simulate a user login and return an authenticated session in integration tests). - In a Python context, fixtures in `pytest` that manage setup/teardown (like creating a temporary database, or patching configuration). - Continuous integration configuration that runs tests in parallel or automates test execution on each commit/pull request.

Often, the most experienced engineers spearhead these efforts. It might be a tech lead writing a flexible **fake server** that mimics third-party APIs so that tests can run against it. Or a senior developer might introduce contract tests and shared stubs across microservices so that each service can test against a common expected interface of its peers. This is time well spent: a powerful test framework multiplies everyone's productivity. It makes doing "the right thing" (writing a test for every change) the path of least resistance, because the friction is low.

On the other hand, if the team neglects the test infrastructure, writing tests can feel slow or cumbersome – and whenever something feels like a tax, people will try to skip it. That's when testing becomes inconsistent. Thus, part of this philosophy is **making testing the natural way to work**, almost the only way to work. The speaker of the original content even noted that he does the right thing (testing thoroughly) not due to external pressure or mandates, but *"because it is the only way that I can work."* When the environment is set up such that writing code and tests together is the easiest and most efficient workflow, developers will happily do it and quality will soar as a result.

Senior developers and engineering leaders play a crucial role in this cultural and tooling shift. They can: - Lead by example by writing tests for their own code and not treating testing as someone else's responsibility. - Pair with other developers to instill good testing habits, showing how to break down tasks into small code-and-test increments. - Review code with an eye for missing tests: if a pull request has code changes with no accompanying tests, call that out and request additional tests (or add them). - Continuously improve test tooling – for instance, if a certain kind of component is hard to test, maybe write a testing wrapper or improve the design so it's easier next time. - Keep the test suite fast. If tests start slowing down or flaky, allocate time to fix that (e.g., by optimizing queries, using in-memory fakes, or splitting tests into parallel jobs). A slow or unreliable test suite is a major deterrent to the whole practice.

High Coverage as a Side Effect, Not an Obsession

With the workflow described, one positive outcome is very high code coverage. It's not uncommon to reach 90%+ coverage consistently, and even approach 100% in many areas, **without explicitly aiming for it**. The reason is simply that you wrote tests for essentially everything that was built or fixed. Coverage, in this sense, is a useful metric to watch as an **indicator** (it can reveal untested code if something is drastically low), but the real focus is on covering important behaviors and scenarios. One could theoretically game coverage numbers (writing trivial tests that execute lines without truly asserting useful outcomes), but that's not the intent here. In fact, high coverage **with meaningful tests** is the goal – and that comes naturally from the practice of always writing a test for each change/feature and for each bug fix.

It's worth noting that some industry experts caution against treating 100% coverage as a requirement if it leads to writing tests that don't provide value [13] [7]. In our approach, however, the high coverage emerges from valuable tests (since every test corresponds to a real feature or a real bug that was addressed). You rarely find yourself writing a test solely to tick a coverage box – you write it because you needed it during development or debugging. Thus, the diminishing returns concern is minimized.

Additionally, high coverage across the codebase changes the development experience qualitatively: - **New team members or contributors can modify code with confidence.** They can rely on the safety net of tests to catch mistakes, which lowers the risk when ramping up on a large codebase. The tests also serve as documentation – new devs can read through tests to understand how different parts of the system are supposed to behave and interact. - **Refactoring and major upgrades become more feasible.** For example, if you want to swap out a library or update a framework version, the extensive test suite will immediately surface any incompatibilities or behavior changes. It's like having a comprehensive checklist to validate the system after any big change. - **Fewer bugs escape to production.** It sounds obvious, but with so many automated checks, most issues are caught long before deployment. The tests run on each commit (in CI) act as a gate. Many teams practicing this approach report a very low rate of production incidents relative to the amount of change being deployed. It's not that bugs never happen, but they are often caught at the development or code review stage (because failing tests block the merge).

In essence, this methodology turns testing into an integral part of development (often called "self-testing code" culture), as opposed to an optional phase that happens after coding. As Fowler noted, having self-testing code fundamentally changes the dynamics of software development by removing fear and increasing agility. Teams practicing it find that *"old codebases [are no longer] terrifying places"* and instead become flexible and resilient to change [14] .

Leveraging AI (LLMs) to Enhance Testing

An exciting development in recent times is the use of AI (Large Language Models like GPT-4 and others) to assist with coding – including writing tests. This workflow, which already emphasizes writing plenty of tests, pairs very naturally with AI assistance. For example, developers can use AI-based tools to generate initial test cases for a given piece of code, or even to update expected outcomes when code changes.

Here are some ways AI can be incorporated: - **Generating boilerplate tests:** Given a function or class, an AI can suggest a suite of unit tests covering common cases. Developers can then refine these tests to fit the exact business logic. This can speed up the process of achieving high coverage, especially for straightforward code where writing tests might otherwise be seen as repetitive. One developer reported using GPT-4 to take a new code module from 0 to 100% test coverage in minutes by auto-generating tests and then adjusting as needed [15] . - **Explaining code via tests:** Sometimes when an AI produces code (say via Copilot or ChatGPT), you might not fully trust or understand it. By asking the AI to also produce tests for that code, you can run the tests to see examples of how the code is intended to work. The speaker mentioned using tests to *"review the code via the tests"* – essentially validating the AI's output and clarifying its behavior. This is a clever way to use AI as both coder and tester, with the human in the loop to verify correctness. - **Maintaining tests:** If a code change breaks many tests (which is expected in this approach for significant changes), an AI assistant can help update those tests. For instance, if you change a function's output format, you might have a dozen tests with hard-coded expected values. AI can quickly propagate the new expected values or adjust assertions across those tests, saving time while you focus on whether the new behavior is correct. Of course, each change should still be reviewed by the developer to ensure the tests aren't blindly modified to "make them pass" incorrectly – the AI is a helper, not an infallible oracle.

The combination of a robust testing culture with AI tools can lead to a hyper-productive cycle: developers spend more time designing and thinking about the logic, while letting the AI suggest tedious parts of tests or code, then the developers verify and refine. The result can be even more tests written (AI doesn't get tired of writing tests!) and an even stronger safety net. We should note, however, that AI-generated tests are only as good as the prompts and the AI's training – they might miss edge cases or sometimes assert the wrong thing. Therefore, human oversight remains vital. The AI can handle quantity, but the team ensures quality.

Finally, AI can also assist in maintaining **code coverage metrics and identifying gaps**. Some tools use AI to analyze your codebase and find areas that lack tests, suggesting what kinds of tests to write. This can be seen as a smart way to continuously improve the test suite, guided by an analysis of risk and complexity.

Conclusion

Test-driven development, in the purest sense, is about using tests to guide and validate every step of coding. The approach outlined here adheres to that spirit, even if it doesn't always start with a failing test for each micro-step. By insisting on writing tests for every change and bug, keeping the feedback loop instantaneous, and treating the test code as a first-class citizen, you create a development environment where doing the right thing is the path of least resistance. The result is **sustainable high-quality software**: code that is well-designed (because it's easy to test and refactor), a test suite that acts as a comprehensive "bug detector" [9], and a team confidence level that allows rapid innovation without sacrificing stability.

This method is language-agnostic – it has been practiced in Python, JavaScript, .NET, Java and more – because it deals with fundamental principles of software engineering. Regardless of stack, fast feedback, continuous testing, and thorough coverage yield the same benefits. It may take some initial investment to set up the necessary test infrastructure and to get the team used to the rhythm, but once it clicks, many developers find they don't want to work any other way. It's a psychologically satisfying workflow: you build something, you get immediate confirmation it works (from your tests), and you move forward with confidence. When something breaks, your tests pinpoint the issue within seconds, and you fix it knowing you won't miss anything.

In an era of complex systems and rapid delivery, this approach provides a firm foundation of quality. It empowers engineering teams – especially senior developers and leaders – to maintain velocity **and** assurance. The mandate isn't "write tests because you must," but rather *"write tests because it makes coding more fun, reliable, and fast in the long run."* By adopting these practices, teams can achieve both high productivity and high reliability, delivering value to users with fewer hiccups and a happier development process for the engineers.

Sources:

- Martin Fowler, *Self-Testing Code* – on the importance of a testing culture and its benefits for refactoring and bug fixing [16] [10] [11].
- Jacob Kaplan-Moss – quote on why code without tests is inherently flawed [1].
- Kent C. Dodds, *"Write tests, not too many, mostly integration"* – advice on favoring real interactions over excessive mocking in tests [6].
- PythonSpeed, *"Fast Feedback Loop"* – notes on how quick test feedback reduces context switching and improves development speed [3].
- Mrinal Maheshwari, *"How to Write Unit Test Cases"* – characteristics of good unit tests being fast and focused [2].

---

[1] Software Engineering Great Quotes - by Maximiliano Contieri
https://maxicontieri.substack.com/p/software-engineering-great-quotes-3af63cea6782

[2] How to Write Unit Test Cases for Any App | Medium
https://blog.mrinalmaheshwari.com/how-to-write-unit-test-cases-for-any-application-28b140cf08f8?gi=fba99283af56

[3] Stuck with slow tests? Speed up your feedback loop
https://pythonspeed.com/articles/slow-tests-fast-feedback/

[4] [5] [8] [9] [10] [11] [12] [14] [16] Self Testing Code
https://martinfowler.com/bliki/SelfTestingCode.html

[6] [7] [13] Write tests. Not too many. Mostly integration.
https://kentcdodds.com/blog/write-tests

[15] I Used GPT-4 to Raise My Test Coverage to 100% in Minutes
https://levelup.gitconnected.com/i-used-gpt-4-to-raise-my-test-coverage-to-100-in-minutes-ae83b5a9d320