Flow Focused Testing Strategies: Working Towards a Pattern Language Alternative to the Test Pyramid

KARL EVARD, New York Times
REBECCA WIRFS-BROCK, Wirfs-Brock Associates

The Test Pyramid was initially proposed an ideal distribution for various types of software tests. It has been a fixture in software development for years. This paper evaluates the Test Pyramid as a Pattern to better understand how it compares to other alternative test shapes, and considers why the idea that distributing tests based on shapes may be fundamentally flawed. We then present an alternative way to think about test distribution written in the form of a pattern language centered around flows.

Categories and Subject Descriptors: **Software and its engineering~Software design engineering • Software and its engineering~Software design tradeoffs • Software and its engineering~Design patterns**

General Terms: Software Testing

Additional Key Words and Phrases: Software patterns, Flow Focused Testing, Test Types, Test Pyramid

ACM Reference Format:

Evard, K. and Wirfs-Brock, R. C. 2026. Flow Focused Testing Strategies: Working Towards a Pattern Language Alternative to the Test Pyramid. HILLSIDE Proceedings of 32nd Conference on Pattern Languages of Programs, People, and Practices (PLoP). (October 2025), 15 pages.

1. Introduction

The Test Pyramid, initially proposed by Mike Cohn in *Succeeding with Agile* [Cohn] and popularized by Martin Fowler [Fowler], proposes an ideal distribution for various types of software tests. It has been a fixture in software development for years. Recently, it has been challenged by authors who propose other "test shapes" such as the Testing Trophy [Dodds] and the Testing Honeycomb [Schaffer]. However, these shapes have been presented not as alternatives, but as objectively better replacements. Here we evaluate the Test Pyramid as a Pattern to better understand how it compares to these alternative shapes, consider why new test shapes may be arising, and explore the idea that distributing tests based on shapes may be fundamentally flawed. We then propose an alternative way to think about test distribution written in the form of a pattern language centered around flows.

2. Some Definitions

2.1 Test Types

Functional Testing focuses on verifying that the system behaves as expected. Functional Tests are commonly organized according to type, e.g.: Unit tests, System tests, Integration tests, End-to-end tests, or UI tests. While these test types are well known, they can have fuzzy boundaries and disputed definitions.

2.2 Test Shapes

The notion of a Test Shape, such as the Test Pyramid, is intended to model an ideal distribution of Test Types within a test suite. A particular "shape" corresponds to a different arrangement of Test Types. Specifically, the pyramid shape consists of a large base of unit tests, a modest number of System tests, and at the highest level of the pyramid, an even smaller number of UI tests. The Test Pyramid is the progenitor of all other Test Shapes. Other test shapes such Testing Trophy and Testing Honeycomb have been proposed as successors to Test Pyramid, while the Test Cupcake and Test Ice-cream Cone have been pointed out as Test Shapes to avoid [Pereira].

https://doi.org/10.64346/PLoP2025p15

2.3 Effective tests

We characterize a test as being effective if it has an overall net benefit because it saves time, money, or effort. For example, bugs in production can cost real money in lost revenue and take time and effort to fix. A test that prevents that from happening would be effective at its purpose. We automate tests for the same reason; an automated test run by a CD pipeline is much cheaper and more reliable than manually testing the same thing every deployment. However, whether a test is effective depends on the context. A test that is very expensive to maintain may be effective if the kind of bug it guards against would be disastrous and is likely to occur. Conversely, a fast and cheap test that detects a bug that is harmless or impossible to arise in production may not be effective enough to keep.

3. Background

3.1 The Test Pyramid as a pattern

Surprisingly for as much as the test pyramid is discussed it's rarely defined, and to our knowledge has not been written in pattern form. So perhaps the best way for us to weigh it against other testing alternatives would be to start by firmly defining it as a pattern. We'll use the definition from "The Test Pyramid" [Fowler] as it's the most cited and seminal work on the subject. We'll write this pattern using a simple Alexandrian style.

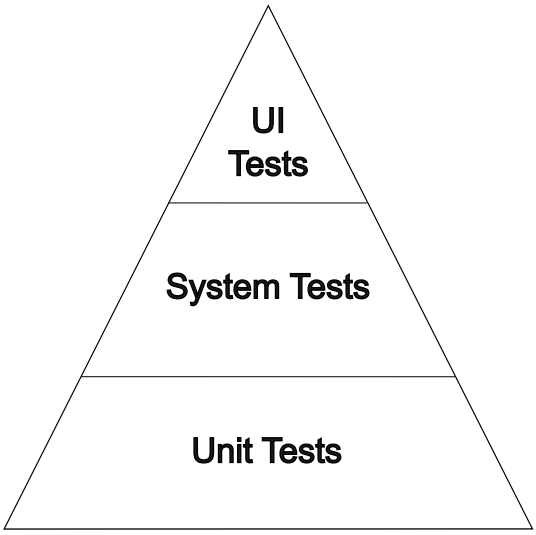

Fig. 1. The Test Pyramid as described in [Fowler]

3.2 The Test Pyramid

Tests can be broken down roughly into three different types: UI tests, Service tests, and unit tests. These different types have different attributes, meaning that tests at the lower-level support tests at the higher levels. Here support means that Unit tests are written first and in greater numbers. Service tests are fewer in number and call upon code that has been Unit Tested. UI tests are even fewer and call on code that has already been both Unit tested and Service tested.

UI tests are tests that manipulate an application via their User Interface. These tests offer the best model of realistic usage as they simulate usage of an application by a user. However, this comes at the expense of speed, reliability, and cost. UI testing frameworks require a fully functioning UI integrated with the application that can take much longer to execute than backend code. Small changes in a UI can cause tests to fail due to harmless changes such as the movement of a button. The frameworks that run these tests can be expensive and require dedicated machines.

Unit tests focus on testing the smallest meaningful units of code, making them fast to run and consistent in their behavior. They are fast to run as they only invoke local code, are easy to parallelize, and avoid external dependencies. They are consistent because intermittent test failures tend to come from changes and inconsistencies of external dependencies which are avoided in Unit Tests. However, since Unit Tests cover small units, many will be required to

cover an application. They also don't verify that units are integrated correctly, only that they work correctly in isolation.

Service tests are a middle ground between UI tests and Unit tests. They cover more than unit tests but less than UI tests, balancing realism with reliability. They can be used to validate that units are working together correctly without depending on the UI. Since the Service tests invoke significant portions of the application's workflows they require a deep knowledge of the application and how upstream dependencies use them. This leads to complex test setup and tests being sensitive to workflow changes.

Given these challenges, how should we go about choosing what types of tests to write and when?

Favor writing fast and consistent Unit tests that provide the base for all other testing. Then create Service tests to validate that the Units are working as intended. Finally, develop UI tests to verify that the application works in realistic usage scenarios.

As consequence, applications will have a large number of small targeted tests. These serve as a foundation that "supports" the correctness of the application by ensuring that individual components work as intended. This can become a liability, however when refactoring, as changes to existing code are highly likely to require many test changes.

3.3 Reflections on the Test Pyramid

The popularity of the Test Pyramid has caused it to be repeated and slightly altered. The main difference in our Test Pyramid Pattern description is that we use different test type names. This difference is because we use the original terms that Fowler used while more recent versions use terms that are in vogue now. For example, Integration test has likely replaced Service test because the term service now commonly refers to individual applications. End-to-end test has likely replaced UI test as backend services use endpoints and that is not generally referred to as a UI. That said, no Test Pyramid variant we've seen is meaningfully different for our purposes as they all imply the same key heuristic: "smaller tests are better."

The original Test Pyramid article didn't explicitly state this, but it's ultimately what many developers take away from it in practice. This results in a unit tests first approach, where test development is primarily focused on writing small self-contained tests instead of creating a balanced mix of tests that provide good system coverage.

In practice, we've also seen developers combining this unit tests first practice with the practice of mocking all downstream calls and writing production code with methods that are as small as possible. This results in writing many small unit tests that don't adequately cover actual execution paths as the code being tested does very little and all downstream calls are mocked.

3.4 How does the Test Pyramid compare to other Test Shapes?

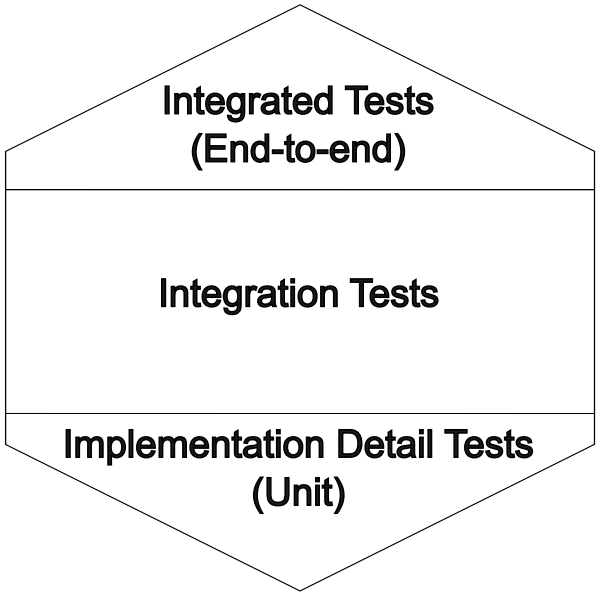

Fig. 2. The Testing Honeycomb as described in [Schaffer] with more common Test Type terms included

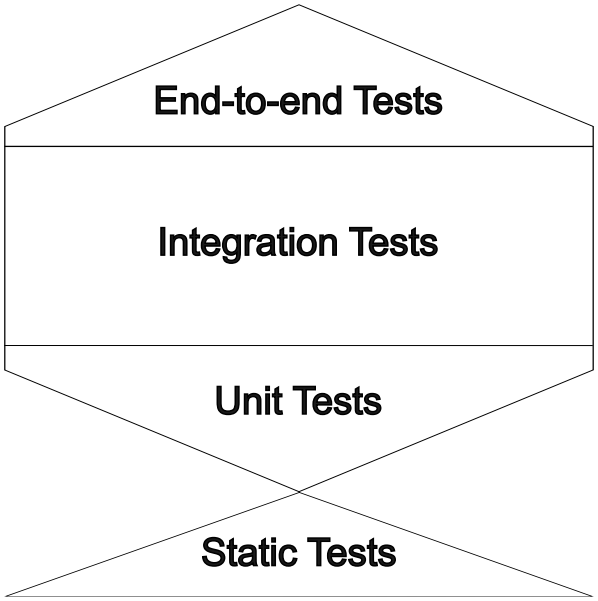

Fig. 3. The Testing Trophy as described in [Dodds]

The Testing HoneyComb is a direct counter to the "smaller tests are better" of the Test Pyramid. Unit tests may be consistent in the sense that they are deterministic and have the same result when rerun. However, often they are not resilient in the face of refactoring. This is especially true if their focus is too narrow and they do not exercise actual production usage as well as system tests do. This one reason why "smaller tests are better" can cause problems. The Testing Honeycomb attempts to solve these problems by trading a foundation of unit tests for a foundation of Integration (AKA Service) tests.

The Testing Trophy is nearly identical to the Testing Honeycomb, but with an additional base of static tests below the unit tests. These static tests are meant to be run by static code analysis tools to detect and prevent bugs related to types and naming conventions.

What's interesting in this comparison is how much various test shapes hinge on the distinctions between test types. As stated earlier, the distinctions between test types have been fuzzy. Depending on how you define the test types or implement your tests, a test could be argued to be a member of several different types. For example, if a test evokes a method that calls another method, is that a unit test or an integration test? Is a test that calls an endpoint in a test environment an integration test because the system under test is integrating with other systems, a system test because it's testing the whole system, or an end-to-end test because it's testing the whole system?

3.5 What makes a Test Type?

If alternate test shapes reverse some of the Test Pyramid's recommendations around test type ratios, what does that tell us about how valuable these distinctions are? Test Types do have meaningful differences but they are not themselves a valuable metric; instead, their attributes are. For example, in the Test Pyramid UI tests and system tests are considered to be slow and unreliable, but if this were not the case, we should consider different distributions.

Here the change in shapes like the Testing Honeycomb make sense. The speed of running UI tests may not have changed much since the Test Pyramid, but System tests have. The newer test shapes aim to fix recommendations from the Test Pyramid that may no longer be relevant by making small adjustments to its recommendations to account for changes in the characteristics of different Test Types that have happened over time due to technology changes.

However, we feel that people should be talking about these tradeoffs more directly and focus on how to adjust their testing strategy to address them. A test's type is meaningful, but is there a better way to determine the scope of the next test we write?

3.6 What makes a comprehensive Testing Strategy?

A comprehensive Testing Strategy guides the writing of tests in different contexts, ideally taking into consideration the various complex forces at play. As the Test Pyramid is rather simplistic only considering Test Types, it lacks a discussion of alternatives and tradeoffs. By ignoring other considerations, it implies they don't matter. By not mentioning any alternatives, it implies they don't exist. And by not discussing tradeoffs, it implies there aren't any.

If it's so flawed why has the Test Pyramid endured? To answer this, we need to consider the time when it was conceived. It was at a time when large monolithic applications with integrated UIs were common. UI tests were possible, but brittle. Fully testing such applications was a lengthy process that was typically done on developers' machines. This also was a time before tools such as Docker which make a local database and other environment set up an automated part of local testing. Additionally, being monolithic meant that such applications were typically owned by a large team or multiple teams.

Such an environment would likely result in a test distribution that resembles the Test Pyramid. On average it would be a successful methodology and thus would be pointed to as a best practice even if its definition lacked nuance. In fact, its lack of nuance may have helped. The Test Pyramid can be communicated via a single graphic image making it an ideal inclusion for documentation, books, and articles.

So why are other Test Shapes challenging the Test Pyramid now? Simply put, the development and deployment environment has radically changed [Caske]. Monoliths are often broken down into microservices. UIs are broken out into their own services. Testing tools make local testing faster and easier. And CI/CD pipelines move automated testing off of single developer machines and into the cloud, making test speed less important.

In short, over the past two decades the environment that gave rise to the Test Pyramid shifted to one full of microservices better served by the Testing Honeycomb. However, since the Test Pyramid did not come with any guidelines on when to use it and it can still be applicable to some applications, a one-size-fits-all suggestion to use it has not gone away. This has resulted in Test Pyramid traditionalists and Testing Honeycomb revolutionaries talking past each other. Both sides argue their approach is the single universal Testing Strategy and rarely discuss how to balance the real forces at play instead of only considering Test Types.

4. Flow Focused Testing: Working towards a Pattern Language for effective Test Suites

Test Shapes create two problems we wish to solve. First, Test Shapes are too simplistic; they only offer guidance on the distribution of Test Types. A more comprehensive Testing Strategy should address additional forces that affect the health of a test suite. Balancing these forces will add significant complexity to the testing strategy, implying that the eventual testing approach is better represented as pattern language and not just a single pattern.

Secondly, Test Types are a poor foundation for judging the effectiveness of a particular test or the health of a test suite. That said, the recent recommendation of preferring Testing Honeycomb over the Test Pyramid suggests that some aspects of Test Types are relevant. Specifically, what should our new guidelines be for deciding how much each test should cover?

A focus on code coverage—the percentage of code that is triggered by tests—might be an alternative. But upon closer examination, code coverage is also inadequate: A line of code being covered only confirms that it was run by a test at least once, not that the code that was tested actually works. Working code that operates as expected is at the heart of what really matters. We'd like a test suite that checks that our application operates correctly in all the different ways it can actually be used in production. So instead of code coverage, we propose a focus on holistically testing execution flows through the application.

What we will present is an approach that primarily is based on experience gained from 10+ years of maintaining a large backend subscription system that hosts over 10 million subscribers. In the beginning, the team Karl was on used the traditional Test Pyramid to guide their testing. Over time, they ran into many of the issues already discussed and began to experiment with other approaches. They then successfully used those approaches to build and test other systems. While we believe there is significant value in the approach we present here, we also recognize that this is just a start and needs subsequent refinement to complete. Additionally, we speculate that developing an effective test suite may be a problem with multiple pragmatic solutions and therefore the patterns and approach we propose here is likely only one of several viable alternatives.

4.1 Defining Flows

We will use the term flow to mean a superset of terms such as User Flow (the flow of actions a user performs on a system) and Work Flow (the flow of actions performed in a system automatically). Regardless of what kind of flow is being tested, the value gained in testing a particular flow is that it as accurately as possible represents how a system operates in production. The more a test matches a flow's execution in production, the more likely it is to break on a meaningful defect. Any part of a flow that is not covered by a test is a place bugs can hide.

For example, if a flow involves state changes, it is possible for a write step to produce a state that is not compatible with a subsequent read step. If the flow is only tested in pieces, the bug will only be discovered if the test for the write step accounts for every nuance of the data written, e.g., a field is written as "ACTIVE" but the read step is case sensitive and looking for "Active". Such issues are easy to miss when writing isolated tests and even easier to miss when refactoring. Conversely, if the flow is tested holistically, the read step invoked by the test will be reading the data written by the write step in the same test and accurately fail on the incompatibly written data.

4.2 How Flows can support effective tests

Previously we defined an effective test as, a test that is a net benefit because it saves time, money, or effort that would be lost without it. This can be broken down into two main considerations. First, a test should be preventing issues that are actually possible and probable. Conversely, a test that does not test possible scenarios will not be effective in testing an application. This can be generally summed up as *realism* and becomes one of our primary forces we will describe in more detail and consider in our patterns.

Since a flow represents an expected path through the code, it focuses on *realism* by definition. Therefore, a focus on Flows when testing means focusing on realism—a core requirement for effective tests. To be clear, this does not mean that every test created this way will be effective, only that it puts the concern of realism front and center in the minds of test writers.

Secondly, our definition of an effective test focuses on various types of costs incurred by writing and maintaining a test. That is, a test can be more trouble than it is worth and incur more costs than its existence protects. This comes up as various cost forces—consistency, resiliency, test suite speed, change speed, price—that must be balanced against realism.

Focusing on Flows does not directly address the cost of a test's existence. However, in practice these concerns get addressed only when costs become an obvious burden. Only then do developers address them out of necessity. This approach can be seen as a way to avoid premature optimization—tests may start out covering a full flow but get their scope narrowed only as problems arise. A number of our patterns focus on how to make tests less expensive with minimal impact on realism**.**

5. Forces

Previously, we discussed how Test Types don't account for all the forces that influence effective tests. Before we recommend actionable patterns, let's describe forces that influence them.

5.1 Realism

Realism is how closely a test emulates production usage. Users don't run individual methods or query a database; instead, they fill out fields, click buttons, and read the displayed text. However, tests that exercise such functionality may be so slow or brittle that the value of their accuracy does not outweigh the value of their realism. In other words, a less realistic test may be more effective in the long run than the most realistic test possible. Because of this, realism vs. the other forces is the core trade off our patterns help balance.

5.2 Consistency

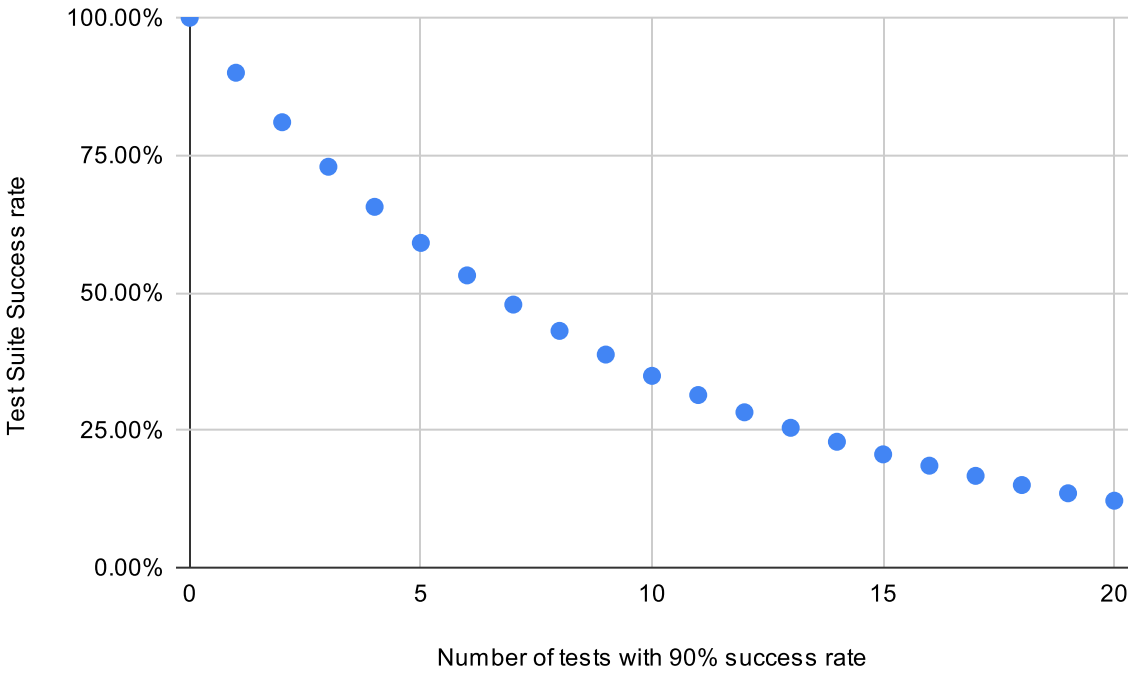

Fig. 4. Test suite success rate by Number of tests with 90% success rate

Consistency is an aspect of reliability that measures how likely a test suite or an individual test succeeds without changes. Any test that can pass, but does not always pass, would not be consistent.

The relation between the consistency of individual tests and a test suite as a whole can be deceptive. Since a single test failure will cause an entire test suite to fail, the success rate of a test suite is the product of the success rate of each individual test. For example, if a test suite consists of 7 tests that are each 90% likely to pass, the test suite as a whole only has a 47.83% chance of passing (see Figure 4). As more tests are added, the success rate of the suite can only go down. So, in a test suite of 1000 tests, just 7 tests with a success rate of 90% or less is enough to cause the suite as a whole to fail more than half the time. This highlights how important the consistency of each test is. Just a few slightly inconsistent tests can make a test suite unusable.

5.3 Resiliency

Resiliency is a form of reliability in the face of changes. Ideally, a test should only break if a code change breaks the functionality the test is checking. However, tests lacking resiliency can fail on changes to their environment that don't meaningfully change functionality or how they interact with downstream systems. In both situations those tests are too tightly coupled with something they shouldn't be. If tests are not resilient to refactoring, they are too tightly coupled to internal details that don't affect functionality. When tests are dependent on details of external systems that do not really impact the system being tested, they too can lack resiliency.

5.4 Test suite speed

Test suite speed is a measure of how long it takes to run an entire test suite. This is typically the sum of how long it takes to run all the tests individually, but can be shortened if tests can be run in parallel. Speed is also impacted by startup steps like building, application startup, or initializing a database.

The time to run a test suite is also affected by automation. Manual steps artificially increase the run time of a test suite as it will pause the suite until a manual step is completed. Conversely, running a test suite for a pull request in a CI/CD pipeline instead of on a developer's local machine can feel faster as it frees a developer from the need to wait for a test run to completion before switching to another task.

5.5 Change speed

An additional speed consideration is the speed of making changes. This is how long it takes to change, add, or understand a test. This is largely dependent on an application's custom test framework, how tests are written, and how they are organized. By test framework we mean any code written to make writing tests easier. A custom test framework can assist by providing helpers, generating production like data for tests, or emulating downstream services for use during testing. A clear and concise custom test framework makes tests clearer and more readable, and consequently easier to understand and edit. Such frameworks also make new tests easier to write as they can reuse or extend existing setup logic.

5.6 Price

By price we generally mean a monetary cost that does not get accounted for by other forces. Some examples are the price of software licenses, infrastructure, and outsourcing QA tasks. For example, the easiest way to increase test suite speed is to run test suites on better hardware, but the price of licensing and infrastructure may be so high that the change is a net negative. Additionally outsourcing testing to a manual QA team could be a continuous cost to running regression tests that could be replaced with a one-time cost of building an automated test suite.

5.7 Code Boundaries

A code ownership boundary exists wherever code owned by one team interacts with code owned by another team. If a test crosses an ownership boundary, a change made by one team may break tests owned by another. In such cases, even the fastest test suite can get bogged down with frequent failures that are not easily fixed by the team that is being blocked.

With a microservice architecture, testing can be complicated by downstream dependencies. For example, if Team A has tests that depend on a system owned by Team and Team B introduces a change, Team A may have tests fail that they can't fix without Team B. Single systems managed by multiple teams can have similar problems if different teams maintain modules which have dependencies. In both cases the largest test type that does not cross a team boundary may be a unit test.

Another aspect to consider is the stability of interface between boundaries. Stable interface between boundaries are common interface points that are rarely expected to change. Tests that depend on stable interface code boundaries are less likely to break due to refactoring, making them more resilient.

Typically, the most stable interface code boundaries are upheld by a contract like REST endpoints. This makes them less likely to cause false positive test failures due to refactoring.

5.8 Internal and external mocks

Mocks can be critical for testing complex code. Mocks come in two general types—internal mocks and external mocks. Internal mocks act as test doubles that mimic behaviors of other parts of the application being tested. External mocks mimic expected behavior of external services such as downstream dependencies or imported libraries. As mocks do not replicate production code, but only imitate it, they should be used with care. Generally, mocks are preferred when using actual production code would be too expensive or is unavailable.

6. Patterns

Considering the forces stated above, we now present patterns in a pattern language that is based around Flow Focused Testing. We present these patterns as short pattern gists grouped into three categories: Flow Patterns are about flows and realism; Speed Patterns consider test suite speed and change speed; and Reliability Patterns address consistency and resiliency.

A number of these patterns are not completely new. They are either adopted wholesale or are variants of other known patterns that can be found in works such as *xUnit Test Patterns* [Meszaros] and *Growing Object-Oriented Software, Guided by Tests* [Freeman]. They are included here because they have been found to be essential to effective Flow Focused testing.

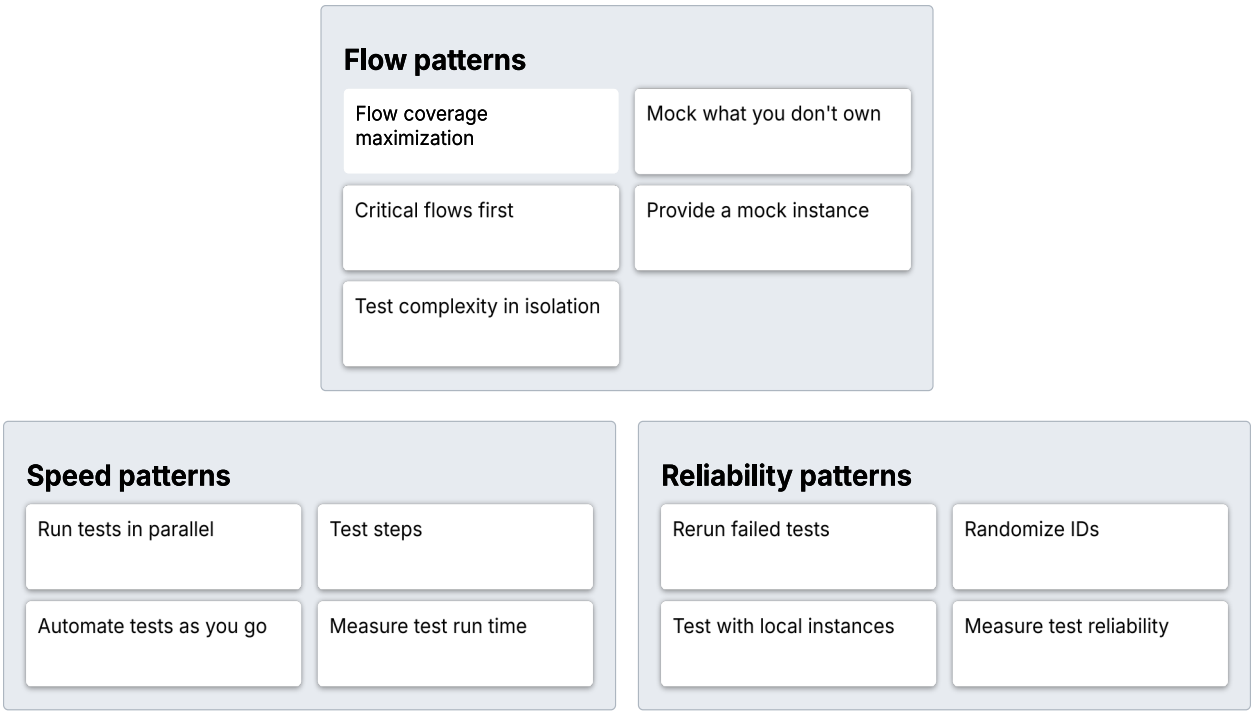

Fig. 5. The proposed Flow Focused testing patterns grouped by their categories

6.1 Pattern: Flow coverage maximization

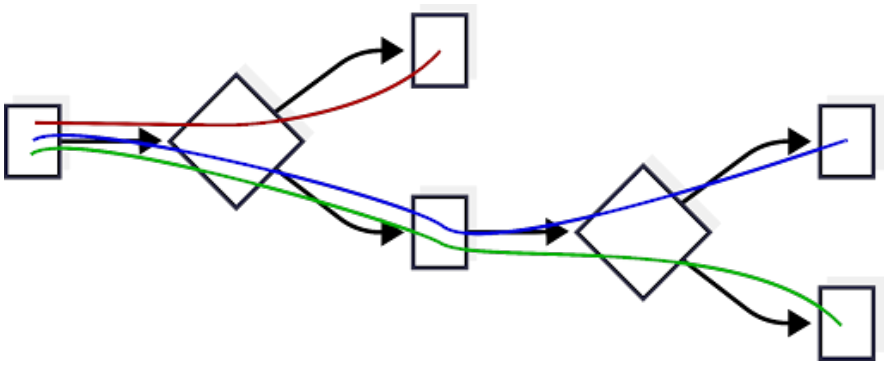

Figure 6: A hypothetical activity diagram showing three distinct but overlapping flows

Adequately testing a system requires testing its flows using a collection of overlapping tests of various sizes. The purpose of a test is to effectively validate usage of an application to ensure its flows work as intended. Such realism requires execution similar to that which is found in production. However, realism has a cost. Downstream integrations may be unreliable harming consistency, or change frequently harming resiliency.

Flows can overlap and require multiple runs to account for edge cases. This can be seen in the accompanying figure where a hypothetical activity diagram of an application shows flows through it represented by colored lines. Covering every possible flow would require a large number of tests that take time to write and execute, harming change speed and test suit speed, respectively. These speed concerns can also be addressed by running more tests in parallel or on faster machines, but this raises the price of each test run.

Some of these issues can be solved by breaking up tests so that individual flows are covered in parts across multiple tests. This results in smaller faster tests, but these smaller tests will lack realism as they will likely depend on mocks or fake test data, and can't account for how one end of a flow might impact the other.

How do we pick the size and scope of tests?

Favor tests that maximize the amount of the flow that is exercised. Tests should cover less than a full flow in cases where testing the full flow detrimentally impacts consistency, resiliency, test suit speed, change speed, or price. These situations and how to address them are covered in other Flow Focused Testing Strategy patterns. For example, a few tests with partially overlapping flows are not likely to impact test suit speed. However, if the number of overlapping tests grows to a point where it does, it can be handled by addressing TEST COMPLEXITY IN ISOLATION. In cases where a test should cover less than the full flow, tests should be aware and take into consideration code boundaries such as code ownership boundaries or stable interface boundaries.

6.2 Pattern: Critical flows first

Even a small code base can have a large number of possible flows. Completely testing all possible flows may be impractical or even impossible. So, a subset of the possible flows must be chosen for testing. Each flow has multiple factors that should be considered: its length, how much it overlaps with existing tests, how frequently it's used in production, etc.

When testing an application, how do we choose what flows to test?

When selecting what flows to test, favor flows that are the most critical in production and are currently underrepresented by the test suite. The most critical flows are those that would cause the most issues if they failed in production. Underrepresented flows are those that have the fewest parts already covered by existing tests.

Other tests might be tempting to consider simply because they are "low hanging fruit." That is, they are easy to construct. But following this tactic can result in the creation of niche tests that are ineffective.

6.3 Pattern: Test complexity in isolation

Certain code paths may be difficult to test either because parts of a particular path contain a large number of branching sub-paths or have code that relies upon conditions that are hard to trigger or information that is difficult to obtain. Testing a code path with a large number of branching paths can require many tests to account for each branch. And testing a sub-path which relies upon environmental conditions that are difficult to setup or resources that are difficult to acquire can also be challenging.

How do we test code paths that contain difficult to test subsections?

Complex paths are good places to lean into smaller tests like Unit tests and internal mocking to fill in testing gaps. This is a corollary to FLOW COVERAGE MAXIMIZATION and CRITICAL FLOWS FIRST. Testing of difficult sub-sections should be done after larger sections of these flows are already well covered. To put that another way–use additional, smaller Unit tests to fill in testing gaps in complex flows.

In general, internal mocks should be avoided as they can bypass large portions of the flows being tested. But in the case of a complex flow, judicious use of internal mocks can be invaluable for simulating situations that are impractical to trigger naturally, such as networking errors. Those portions that are being mocked should have already been covered by FLOW COVERAGE MAXIMIZATION.

6.4 Pattern: Mock what you don't own

Mocks are a powerful tool that can be used to replace parts of flows simplifying set up and speeding up test execution. However, since they replace the actual code that will be used in production, they hurt the realism of the flow they are being used to test.

When and how should mocks be used?

Internal mocks should generally be avoided. They can be critical to testing complexity in isolation but they hurt realism by skipping parts of the flows being tested.

External mocks should be preferred when testing code which has dependencies that cross code boundaries. Calling across a distribution boundary is realistic but it can harm test suite speed and consistency if the dependencies change. Even worse, a break in a dependency that is not owned by the same team that owns the test can cause development and/or deployments to be blocked. This is more likely to happen with tests as they're typically connected to non-production environments that are less stable. However, a database or other service may be a dependency so closely coupled to the system being tested that it should be considered part of it. In these cases, local or test instances of the dependency are preferable to using an external mock. A general strategy is to mock out systems that are not part of the application being tested.

6.5 Pattern: Provide a mock instance

External mocks can be a critical aspect of tests that depend on external dependencies. However, the team that needs the external mock is rarely the same team that is best qualified to create and maintain it. A poor implementation of an external mock can reduce change speed, resiliency, and realism. Additionally, external mocks are likely to be used by more than one test suite as systems tend to have multiple clients.

How can we best manage the creation and maintenance of external mocks?

Ideally, teams that create systems or libraries should PROVIDE A MOCK INSTANCE for clients to use in testing. Such a mock can provide an accurate representation of the dependency. If that dependency needs to change, the mock can be updated before new changes are put into production.

An alternative solution, particularly useful for libraries, is to provide testing modes or hooks. These hooks can modify how a dependency works to trigger specific responses or force specific states or transitions.

At a minimum, teams that create systems should provide clear documentation that is detailed enough so that creation of an external mock does not require guesswork.

6.6 Pattern: Run tests in parallel

As applications grow, the number of tests that support them tend to grow as well. This can be the biggest threat to test suite speed.

How can we speed up a large test suite?

The most direct way to speed up a test suite is to RUN TESTS IN PARALLEL. Modern hardware supports parallel execution well and applications are likely to have their operations bound by wait intensive IO operations.

However, parallel execution opens the door to race conditions. Consequently, running tests in parallel can improve test suite speed by sacrificing consistency. In practice, however, such issues can typically be addressed with a bit of debugging and use of other patterns such as TEST STEPS, TESTING WITH LOCAL INSTANCES, RANDOMIZING IDs, MOCKING WHAT YOU DON'T OWN, and RERUNNING FAILED TESTS.

6.7 Pattern: Test steps

Writing the setup and teardown for tests can be a repetitive and time-consuming process that negatively impacts change speed and resiliency. Such code can also be long winded and convoluted making it difficult to read. Many tests may need slightly different versions of this code, leading to a kind of change blindness where a critically different part of a test's setup is easy to miss.

How can we reduce the pain of repetitive test code?

Turn custom testing framework code into reusable step functions. The term step has its origins in Behavior-driven development (BDD). A Step Definition in BDD is a code implementation that is used in place of a domain language phrase [Smart]. The general organizational idea of steps can be used without a full BDD domain language approach by simply organizing common test code into easy to use, reusable, and well named helper functions.

Ideally every step condenses a large common chunk of test code into a single, easy to read, line of code that invokes a step. Additionally, steps could be composed of other steps. For example, in a commerce site an “Add item to cart” step and “Checkout” step could be combined into a "Purchase Item” step.

Steps can also help to bring DRY principles to tests. This reduces change speed as small application changes can be accounted for with isolated step changes. Concise code is also critical to avoid change blindness. For example. when looking at invocations of a step with a large number of parameters, it can be difficult to notice if a single parameter has changed between two similar tests.

6.8 Pattern: Automate tests as you go

Tests suites can leverage a wide array of tools to ease test development and execution. However, tools can require time consuming configuration and troubleshooting for local usage that can hurt resiliency, test suite speed, and change speed.

How can we minimize the negative impacts of tools used in testing?

Removing manual steps should not be confined to just the tests themselves. Tasks such as installing tools, initializing them, and correcting common issues they can run into are steps that can and should be automated. That said, automation opportunities are hard to predict and are best addressed as they arise.

6.9 Pattern: Measure test run time

A continuous stream of development changes can slowly eat away at test suite speed as the corresponding test suite grows. This can be mitigated by refactoring tests, but it can be difficult to identify what tests are eating up the most execution time. Especially if the time consumption is inconsistent.

How do we know where to focus our efforts when refactoring tests for speed?

By recording the run time of individual tests, we can identify what tests take the longest to run. Additionally, if such tracking is built into a CI/CD pipeline and automatically saved as a metric, tracking test run time across multiple runs can become trivial and help to identify specifically what change impacted a test's runtime.

6.10 Pattern: Rerun failed tests

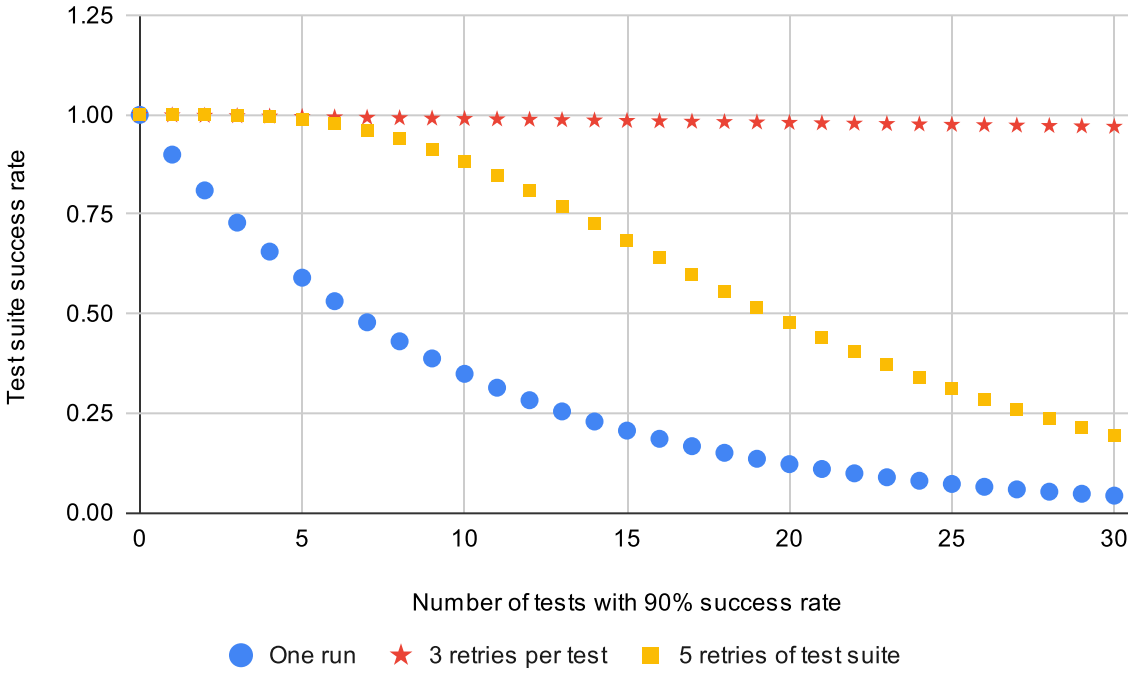

Fig. 7. Test suite success rate by on number of tests with 90% success rate with different mitigation approaches

Ideally every test will be fully deterministic; if the code is working the test will pass 100% of the time. In reality however, some tests will have intractable problems that cause them to have a less than perfect success rate and therefore compromise test suite consistency.

How can we make a test suite consistent while it contains inconsistent tests?

The overall consistency of a test suite can be improved by simply rerunning only the tests that failed. This comes at the expense of test suite speed as additional reruns take time, especially when a real error is present. However, by only running the tests that failed the additional runtime is kept to a minimum.

There needs to be some limit to how many times failed tests are rerun in order to prevent a test suite with legitimate failures from being rerun indefinitely. If the maximum number of retries is too high, a failing test suite will take a significant amount of time to complete. If that number is too low, inconsistent tests can still cause the test suite to fail.

Retrying failed tests improves the consistency of the test suite significantly and drastically more than if the entire test suite were to be rerun as shown in Figure 7.

6.11 Pattern: Randomize IDs

Tests that depend on resources like local files or databases can be greatly sped up if they do not each need their own set up. However, sharing such resources opens the opportunity for collisions between tests that would work if run independently. This can be an issue with running tests in parallel, testing with local instances, or when rerunning failed tests.

How can we reduce collisions when we share resources between tests?

When IDs created by a test can conflict, randomize them. Randomized IDs offer a way to automatically create IDs that can't conflict.

6.12 Pattern: Test with local instances

It is common for applications to depend on other external systems in their production environment. Testing against such dependencies provides realism but risks consistency if those dependencies are having issues and risks change speed and test suite speed if the dependency is difficult to set up for testing.

How can we address external dependencies when testing?

If an external dependency is part of your application and you can run an instance of it locally (for example a database), then run an instance of it locally. If an external dependency is provided, you should also run that locally. While this may not be as realistic as testing your application against a production or staging instance, it's more realistic than creating mock responses.

6.13 Pattern: Measure test reliability

As applications grow, small changes can reduce resiliency. This can happen so slowly that problems are noticed well after changes have been introduced. Additionally, slightly inconsistent tests are inherently hard to track as their failures are rare and easily forgotten.

How do we know where to focus our efforts when refactoring tests for consistency and resiliency?

By keeping a record of the successes and failures of individual tests, we can identify what tests are most likely to fail. Additionally, if such tracking is built into a CI/CD pipeline and automatically saved as a metric, tracking test run failures across multiple runs can become trivial. This helps to identify specifically when a test's reliability has changed. Some tests can fail due to conflicts with each other. Such failures can occur in clusters that are much easier to notice and diagnose with a rich history of test runs to analyze.

7. Consequences of Flow Based Testing

The following are some consequences of flow focused testing that Karl and his team have experienced after using it in their applications.

7.1 Flows are hard to see

While flows represent real paths through the code used in production, such paths are not easy to see from directly reading code. Organization is also complicated as flows can partly cover various components. This means that simple naming conventions such as the JUnit standard of adding "test" to the name of the class being tested doesn't work.

Together this means that how to identify what flows are covered by an existing test and what flows are not covered lacks a straightforward solution. Care must be taken to use meaningful test naming conventions and maintain order.

In practice, we've found this to be a minor annoyance. When looking to see if an existing flow is covered, a developer can usually do a simple text search for TEST STEPS used by the flow in question. To organize, tests can be grouped by the feature they test. Identifying gaps is the most challenging problem we've found, but that can be accomplished by listing critical flows and then searching to see if they are already accounted for.

It's worth noting that these problems could be addressed in the future with better tools. Code coverage tools could be enhanced to track what lines are called by what test to highlight what flows are covered. This would provide similar insights to flow coverage that we currently get from code coverage tools.

7.2 What flows are critical is nebulous

The CRITICAL FLOWS FIRST pattern discusses the importance of identifying critical flows and ensuring they are well tested. What it does not address is the challenge and finer points of how to select what flows are critical. Additionally, what is critical can change over time as new features are added and usage of existing features changes.

In our experience this is only a minor annoyance. We have found critical flows to be fairly obvious and easily uncovered by simply creating a test plan. However, this does imply that developers should extend their skill set to include creating test plans and understanding production usage.

This is also a possible place where tool enhancements could improve things. Tracing and observability libraries could be enhanced to track what lines or methods are used together in production to evaluate what flows are used the most in production. This could give concrete data on what flows are used most and how usage is changing.

7.3 Longer tests can be harder to manage

The idea that larger tests are more costly is a foundational principle of the Test Pyramid. In practice we've found this to be the most persistent challenge with flow based testing. Tests that cover more will take more time to run and can require more set up. This is why a large number of our flow focused testing patterns address these problems.

Our largest application that uses this approach has 1,800 integration tests and runs in roughly 10 minutes. We've accomplished this by running our tests in parallel thus trading consistency to gain test suite speed. Use of some of the patterns here have been critical to keeping consistency, resiliency, test suite speed, and change speed under control and ensure that our test suite is effective. A commitment to focusing on flows requires an ongoing focus on managing the forces that cost developer time and attention.

8. Conclusion

In the past, the Test Pyramid has been a convenient way to recommend a Test Strategy. However, recently proposed alternatives indicate that it may be time to look at more fundamental principles than simply the Test Types we should select when testing. Focusing on flows that contribute business value and understanding the forces behind effective Test Suites offers an approach that is more complex but more comprehensive.

8.1 Future work

What we've offered here is only a start. We've highlighted a gap in testing best practices and presented a summary of a solution we've practiced in the form of a new pattern language for testing strategies based on flows. However, this summary has gaps we intend to explore in future papers. Our approach was primarily developed by testing a backend monolith. This means we have limited experience using this approach for microservices, frontend systems, and native apps. Additionally, we don't yet have good suggestions for measuring flow coverage and identifying existing gaps in coverage. As what we've presented are summary patterns or gists, we also plan to expand on them with more comprehensive pattern descriptions and examples. We also plan refine by incorporating feedback from others developers who are experienced in testing complex system.

REFERENCES

[Cohn] Mike Cohn. *Succeeding With Agile: Software Development Using Scrum*. Addison-Wesley Professional, 2009

[Craske] Antoine Craske, The Traditional Test Automation Pyramid, Pitfalls and Anti-patterns. Retrieved Jan. 11, 2026 from: https://laredoute.io/2020/04/24/the-traditional-test-pyramid-pitfalls-and-anti-patterns/

[Dodds] Kent Dodds. June 2021. The Testing Trophy and Testing Classifications. Retrieved January 11, 2026 from: https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications

[Fowler] Martin Fowler. May 2012, Test Pyramid. Retrieved January 11, 2026 from: https://martinfowler.com/bliki/TestPyramid.html

[Freeman] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009

[Meszaros] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007

[Pereira] Fabio Pereira. June 2014, Introducing the Software Testing Cupcake (Anti-Pattern) Retrieved January 19, 2026 from: https://www.thoughtworks.com/insights/blog/introducing-software-testing-cupcake-anti-pattern

[Schaffer] André Schaffer. January 2018, Testing of Microservices. Retrieved January 11, 2026 from: https://engineering.atspotify.com/2018/01/testing-of-microservices

[Smart] John Ferguson Smart and Jan Molak. *BDD in Action Second Edition: Behavior-Driven Development for the whole software lifecycle*. 2nd Edition. Manning, 2023