Chapter 3. Coding and Testing for Observability

Historically, testing has been something that referred to a pre-production or pre-release activity. Some companies employed—and continue to employ—dedicated teams of testers or QA engineers to perform manual or automated tests for the software built by development teams. Once a piece of software passed QA, it was handed over to the operations team to run (in the case of services) or shipped as a product release (in the case of desktop software or games).

This model is slowly but surely being phased out (at least as far as *services* go). Development teams are now responsible for testing as well as operating the services they author. This new model is incredibly powerful. It truly allows development teams to think about the scope, goal, trade-offs, and payoffs of the entire spectrum of testing in a manner that's realistic as well as sustainable. To craft a holistic strategy for understanding how services function and to gain confidence in their correctness before issues surface in production, it becomes salient to be able to pick and choose the right subset of testing techniques given the availability, reliability, and correctness requirements of the service.

Software developers are acclimatized to the status quo of upholding production as sacrosanct and not to be fiddled around with, even if that means they always verify in environments that are, at best, a pale imitation of the genuine article (production). Verifying in environments kept as identical to production as possible is akin to a dress rehearsal; while there are some benefits to this, it's not quite the same as performing in front of a full house.

Pre-production testing is something ingrained in software engineers from the very beginning of their careers. The idea of experimenting with live traffic is either seen as the preserve of operations engineers or is something that's met with alarm. Pushing some amount of regression testing to post-production monitoring requires not just a change in mindset and a certain appetite for risk, but more importantly an overhaul in system design, along with a solid investment in good release engineering practices and tooling.

In other words, it involves not just architecting for failure, but, in essence, coding and testing for failure when the default was coding (and testing) for success.

Coding for Failure

Coding for failure entails acknowledging that systems will fail, being able to debug such failures is of paramount importance, and enshrining debuggability into the system from the ground up. It boils down to three things:

- Understanding the operational semantics of the application
- Understanding the operational characteristics of the dependencies
- Writing code that's debuggable

Operational Semantics of the Application

Focusing on the operational semantics of an application requires developers and SREs to consider:

- How a service is deployed and with what tooling
- Whether the service is binding to port 0 or to a standard port
- How an application handles signals
- How process starts on a given host
- How it registers with service discovery
- How it discovers upstreams
- How the service is drained off connections when it's about to exit
- How graceful (or not) the restarts are
- How configuration—both static and dynamic—is fed to the process
- The concurrency model of the application (multithreaded, purely single threaded and event driven, actor based, or a hybrid model)
- The way the reverse proxy in front of the application handles connections (pre-forked, versus threaded, versus process based)

Many organizations see these questions as something that's best abstracted away from developers with the help of either platforms or standardized tooling. Personally, I believe having at least a baseline understanding of these concepts can greatly help software engineers.

Operational Characteristics of the Dependencies

We build on top of increasingly leaky abstractions with failure modes that are not well understood. Here are some examples of such characteristics I've had to be conversant with in the last several years:

- The default read consistency mode of the Consul client library (the default is usually "strongly consistent," which isn't something you necessarily want for service discovery)
- The caching guarantees offered by an RPC client or the default TTLs
- The threading model of the official Confluent Python Kafka client and the ramifications of using it in a single-threaded Python server
- The default connection pool size setting for pgbouncer, how connections are reused (the default is LIFO), and whether that default is the best option for the given Postgres installation topology

Understanding such dependencies better has sometimes meant changing only a single line of configuration somewhere or overriding the default provided by a library, but the reliability gains from changes have been immense.

Debuggable Code

Writing debuggable code involves being able to ask questions in the future, which in turn involves the following:

- Having an understanding of the instrumentation format of choice (be it metrics or logs or exception trackers or traces or a combination of these) and its pros and cons
- Being able to pick the best instrumentation format given the requirements of the given service and the operational quirks of the dependencies

This can be daunting, and coding accordingly (and more crucially, being able to *test* accordingly) entails being able to appreciate these challenges and to address them head-on at the time of writing code.

Testing for Failure

It’s important to understand that testing is a best-effort verification of the correctness of a system as well as a best-effort simulation of failure modes. It’s impossible to predict every possible way in which a service might fail and write a regression test case for it, as E. W. Dijkstra has pointed out:

> *The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.*

Unit tests only ever test the behavior of a system against a specified set of inputs. Furthermore, tests are conducted in very controlled (often heavily mocked) environments. While the few who do fuzz their code benefit from having their code tested against a set of randomly generated input, fuzzing can comprehensively test against the set of inputs to only *one* service. End-to-end testing might allow for some degree of holistic testing of the system, but complex systems fail in complex ways, and there is no testing under the sun that enables one to predict every last vector that could contribute toward a failure.

This isn’t to suggest that testing is useless. If nothing else, testing enables one to write better, maintainable code. More importantly, research has shown that something as simple as “testing error handling code could have prevented 58% of catastrophic failures” in many distributed systems. The renaissance of tooling aimed to understand the behavior of our services in production does not obviate the need for pre-production testing.

However, it’s becoming increasingly clear that a sole reliance on pre-production testing is largely ineffective in surfacing even the known-unknowns of a system. Testing for failure, as such, involves acknowledging that certain types of failures can only be surfaced in the production environment.

Testing in production has a certain stigma and negative connotations linked to cowboy programming, insufficient or absent unit and integration testing, as well as a certain recklessness or lack of care for the end-user experience.

When done poorly or haphazardly, “testing in production” does, in fact, very much live up to this reputation. Testing in production is by no means a substitute for pre-production testing, nor is it, by any stretch, easy. Being able to successfully and safely test in production requires a significant amount of diligence and rigor, a firm understanding of best practices, as well as systems designed from the ground up to lend themselves well to this form of testing.

In order to be able to craft a holistic and safe process to effectively test services in production, it becomes salient to not treat “testing in production” as a broad umbrella term to refer to a ragbag of tools and techniques.

“Testing in production” encompasses the entire gamut of techniques across three distinct phases: deploy, release, and post-release (Figure 3-1).

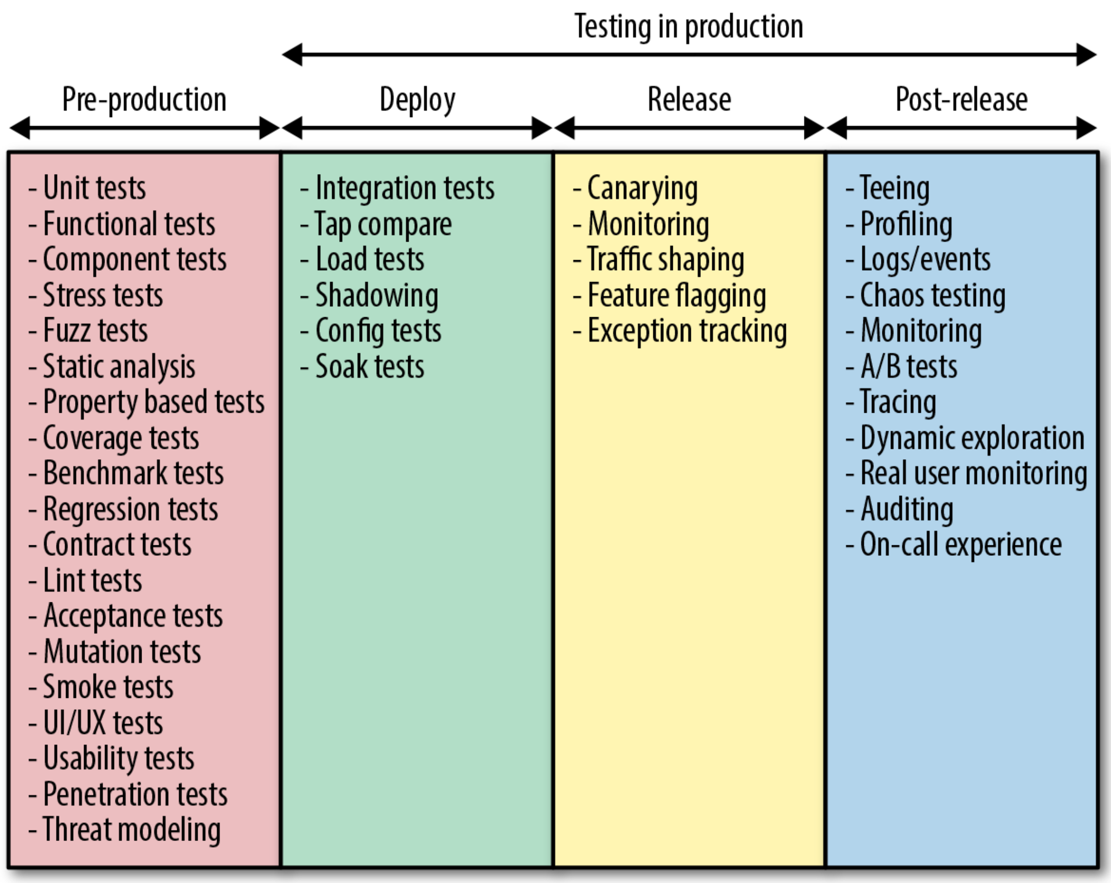

Figure 3-1. The three phases of testing in production

The scope of this report doesn’t permit me to delve deeper into the topic of testing in production, but for those interested in learning more, I’ve written a detailed post about the topic.

The overarching fact about testing in production is that it’s impossible to do so without measuring how the system under test is performing in production. Being able to test in production requires that testing be halted if the need arises. This in turn means that one can test in production only if one has the following:

- A quick feedback loop about the behavior of the system under test
- The ability to be on the lookout for changes to key performance indicators of the system

For an HTTP service, this could mean attributes like error rate and latencies of key endpoints. For a user-facing service, this could additionally mean a change in user engagement. Put differently, testing in production essentially means proactively “monitoring” the test that’s happening in production.

Conclusion

Testing in production might seem daunting, way above the pay grade of most engineering organizations. The goal of testing in production isn’t to eliminate all manner of system failures. It’s also not something one gets for free. While not easy or risk-free, undertaken meticulously, testing in production can greatly build confidence in the reliability of complex distributed systems.