



Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Dec 31, 2017 · 39 min read

Testing Microservices, the sane way

There's no dearth of information or best-practices or books about how best to test software. This post, however, focuses solely on testing backend *services* and not desktop software or safety critical systems or GUI tools or frontend applications and what have you.

Distributed systems means different things to different people.

For the purpose of this post, by “distributed system” I mean a system composed of many moving parts, each with different guarantees and failure modes which, together, work in unison to provide a business feature. This certainly isn't close to the canonical definition of a “distributed system”, but these are the kind of systems I have most experience working with and I'd argue these are the kind of systems the *vast majority* of us build and operate. Albeit we're talking about a distributed system, the terminology du jour is “microservices architecture”.

The mainspring of microservices

The ability to develop, deploy and scale different business functionality independently is one of the most touted benefits of adopting a microservices architecture.

While the jury's still out on if these benefits really check out or not, microservices are very much in vogue, to the point where this architecture has now become the default of even many startups.

Yet when it comes to *testing* (or worse, when *developing*) microservices, most organizations seem to be quite attached to an antediluvian model of testing all components *in unison*. Elaborate testing pipeline infrastructures are considered mandatory to enable this form of end-to-end testing where the test suite of *every* service is executed to confirm there aren't any regressions or breaking changes being introduced.

**Cindy Sridharan**

@copyconstruct

Yep! The whole point of microservices is to enable teams to develop, deploy and scale independently.

Yet when it comes to testing, we insist on testing *everything* together by spinning up *identical*

Full Stack in a Box—A Cautionary Tale

I often see or hear about many companies trying to replicate the *entire* service topology on developer laptops locally. I've had first hand experience with this fallacy at a previous company I worked at where we tried to spin up our entire stack in a Vagrant box. The Vagrant repo itself was called something along the lines of “full-stack in a box”, and the idea, as you might imagine, was that a simple `vagrant up` should enable *any* engineer in the company (even frontend and mobile developers) to be able to spin up the stack in its *entirety* on their laptops.

Properly speaking, this really wasn't even a fully-blown Amazon-esque microservices architecture comprising of thousands of services. We had *two* services on the backend—a gevent based API server and some asynchronous Python background workers which had a tangle of gnarly native dependencies including the C++ boost library which, if memory serves me right, was compiled from scratch every time a new Vagrant box got spun up.

I remember spending my entire first week at this company trying to successfully spin up the VM locally only to run into a plethora of errors. Finally, at around 4:00pm on the Friday of my first week, I'd successfully managed to get the Vagrant setup working and had all the tests passing locally. More than anything else I remember feeling incredibly exhausted after this rigmarole and took it upon myself to document the issues I ran into to so that the next engineer we hired would have it easier.

Except, the next engineer ran into their own fair share of issues with the Vagrant setup that I personally couldn't replicate locally on my laptop. In fact, the entire setup on my laptop was so *brittle* that I feared

even so much as upgrading a Python library, since running `pip install` wrecked my Vagrant setup and would lead to tests failing locally.

It also turned out that none of the other engineers on any of the web or mobile teams had much success with the Vagrant setup either and troubleshooting Vagrant issues became a frequent source of support requests to the team I was on. While one could argue that we should've spent more engineering cycles to get the Vagrant setup fixed once and for all so that it "just worked", to our defense this was at a startup and engineering cycles at startups are *invariably* thin on the ground.

Fred Hébert left an incredibly thoughtful review of this post and an observation he made struck me as being absolutely spot-on that I'm sharing it here:

... asking to boot a cloud on a dev machine is equivalent to becoming multi-substrate, supporting more than one cloud provider, but one of them is the worst you've ever seen (a single laptop)

Even with modern Operational best practices like infrastructure-as-code and immutable infrastructure, trying to replicate a cloud environment locally doesn't offer benefits commensurate with the effort required to get it off the ground and subsequently maintain it.

Building and Testing microservices, monolith style

What I've since come to realize after speaking with friends is that this is a problem that plagues not just startups but also a vast number of much larger organizations. Over the course of the last few years I've heard enough anecdotal evidence about the inherent fragility of this setup and the maintenance costs of the machinery required to support it that I now believe trying to spin up the full stack on developer laptops is fundamentally the *wrong mindset* to begin with, be it at startups or at bigger companies.

In fact, doing so is tantamount to building a distributed monolith.



Kelsey Hightower
@kelseyhightower

2020 prediction: Monolithic applications will be back in style after people discover the drawbacks of distributed monolithic applications

As the noted blogger Tyler Treat says :

lmao at this microservices discussion on HN. "Devs should be able to run entire env locally. Anything else is just a sign of bad tooling." Ok yeah please tell me how well it works running two dozen services with different dbs and dependencies on your macbook. But yeah I'm sure docker compose has you covered.

This is all too common. People start building microservices with a monolith mindset and it always ends up a shitshow. "I need to run everything on my machine with this particular configuration of services to test this one change." jfc.

If anyone so much as sneezes my service become untestable. Good luck with that. Also, massive integration tests spanning numerous services are an anti-pattern, but it still seems like an uphill battle convincing people. Moving to microservices also means using the right tools and techniques. Stop applying the old ones in a new context.

As an industry, we're beholden to test methodologies invented in an era vastly different to the current one we're in. People still seem to be enamored with ideas such as full test coverage (so much so that at certain companies a merge is blocked if a patch or a new feature branch ends up *fractionally* decreasing the test coverage of the codebase), test-driven development and complete end-to-end testing at the system level.

To this end they wind up investing excessive engineering resources to build out complex CI pipelines and intricate local development environments to meet these goals. Soon enough, sustaining such an elaborate setup ends up requiring a team in its own right to build, maintain, troubleshoot and evolve the infrastructure. Bigger companies

can afford this level of sophistication, but for the rest of us treating testing as what it really is—a best effort verification of the system—and making smart bets and tradeoffs given our needs appears to be the best way forward.

The Spectrum of Testing

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 whose sole responsibility was 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 and what have you).

This model is slowly but surely being phased out (at least as far as *services* go, from what I can tell ensconced in my San Francisco startup bubble). Development teams are now responsible for testing as well as operating the services they author. This new model is something I find *incredibly* powerful since it truly allows development teams to think about the scope, goal, tradeoffs and payoffs of the *entire spectrum* of testing in a manner that's realistic as well as sustainable. In order to be able to craft a holistic strategy for understanding how our services function and gain confidence in their correctness, 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.



Charity Majors

@mipsytipsy

Replying to @mipsytipsy and 2 others
Testing don't end when shit deployed.



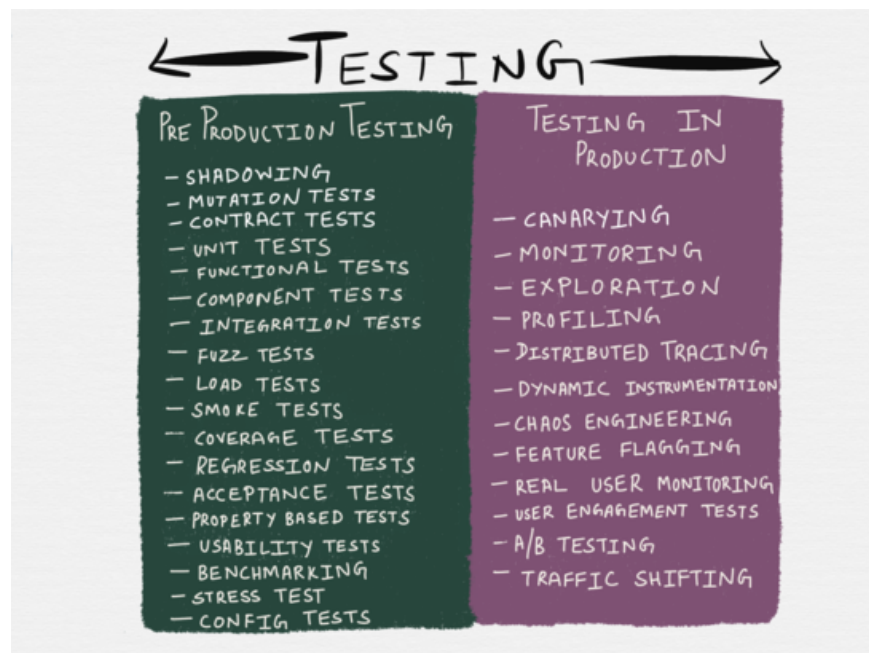
Baron Schwartz

@xaprb

Replying to @mipsytipsy and 2 others

So many amens. Testing pre-deploy is partially prep for testing in production. To paraphrase some of what you said: pre-deploy testing may teach, rehearse & reinforce biased mental models of the system, and

By and large, “testing” can be used to encompass a variety of activities, including many practices that traditionally used to fall under the umbrella of “release engineering” or Operations or QA. Several of the techniques listed in the illustration below are not strictly seen as forms of testing in some circles—for instance, the official definition of chaos engineering classifies it as a form of *experimentation*—and I also do not aim to pretend this is a comprehensive list—security testing (vulnerability testing, penetration testing, threat modeling and so forth) isn’t featured anywhere at all, for a start—but it encompasses some of the most common forms of testing seen in the wild all the same.



I must insist here that while the illustration above presents the testing taxonomy as a binary, the reality isn’t quite as neatly delineable as depicted here. For instance, profiling falls under the “testing in

production” column, but it can very well be done during development time in which case it becomes a form of pre-production testing. Shadowing, similarly, is a technique where a small fraction of production traffic is replayed against a small number of test instances, and depending on how you view it, might qualify as “testing in production” (since it involves testing the service with production traffic) or as a form of pre-production testing (since end users aren’t being affected by it).

Programming languages offer varying degrees of support for testing applications in production as well. If you’re an Erlangista, you’re probably familiar with Fred Hébert’s manual on using the Erlang VM’s primitives to debug production systems while they are running. Languages like Go come with built in support for obtaining the heap, mutex, CPU and goroutine profiles of any running Go process (testing in production) or when running unit tests (this would then qualify as pre-production testing).

Testing in Production as a substitute to Pre-Production testing?

I’ve previously written in great detail about “post-production testing” from primarily an Observability standpoint. Monitoring is a form of post-production testing, as is alerting, exploration and dynamic instrumentation. It might not even be stretching it to call techniques like featuring flagging and gating as forms of testing in production. User interaction or user experience measurement—often performed using techniques like A/B testing and real user monitoring—constitute as testing in production as well.

There has been some discussion in certain circles about how such forms of testing could possibly replace pre-production testing. Sarah Mei had a thought provoking discussion about this a while ago. There’s a lot to unpack here and while I don’t agree with all of the points Sarah makes, there’s a hell lot I do see eye to eye with. Sarah goes on to state that:

Conventional wisdom says you need a comprehensive set of regression tests to go green before you release code. You want to know that your changes didn’t break something elsewhere in the app. But there are ways to tell other than a regression suite. Especially with the rise of more sophisticated monitoring and better understanding of error rates (vs uptime) on the operational side.

With sufficiently advanced monitoring & enough scale, it's a realistic strategy to write code, push it to prod, & watch the error rates. If something in another part of the app breaks, it'll be apparent very quickly in your error rates. You can either fix or roll back. You're basically letting your monitoring system play the role that a regression suite & continuous integration play on other teams.

A lot of people construed this to mean that pre-production testing wasn't required at all, which isn't how I interpreted this. Reading between the lines, I suspect a lot of the uproar could be attributed to the fact that a lot of software developers (and professional software testers) have a hard time coming to terms with the fact that manual or automated pre-production testing *alone* might perhaps not be sufficient, to the point where it might sometimes be *entirely insufficient*.

The book **Lessons Learned in Software Testing** has a chapter called *Automated Testing*, where the authors claim that automated regression tests find only a *minority* of the bugs.

Informal surveys reveal that the percentage of bugs found by automated tests are surprisingly low. Projects with significant, well-designed automation efforts report that regression tests find about 15 percent of the total bugs reported (Marick, online).

Regression-test automators typically find more bugs during test development than when they execute tests later. However, if you take your regression tests and find opportunities to reuse them in different environments (e.g. a different hardware platform or with different software drivers), your tests are more likely to find problems. In effect, they are actually no longer regression tests, since they are testing configurations that haven't been tested before. Testers report that these kinds of automated tests can generate yields closer to 30 to 80 percent.

This book is certainly a bit dated and I haven't been able to find any recent study about the efficacy of regression tests, but so much of what we've been conditioned to believe to be good software engineering practices and disciplines is built upon the primacy of automated testing that any form of doubt cast on its efficacy sounds sacrilegious to many. If there's anything I've learned in the last few years of witnessing how

services fail, it's that pre-production testing is a *best effort verification* of a *small subset* of the guarantees of a system and often can prove to be grossly insufficient for long running systems with protean traffic patterns.

To continue with Sarah's thread:

*This strategy assumes a lot of things, starting with an operational sophistication that most dev teams don't have. There's more, as well. It assumes the ability to segment users, show each dev's change-in-progress to a different segment, & allocate error rates per segment. It assumes sufficient scale that variations in a segment's error rates will be statistically significant. **And perhaps most unusually, it assumes a product organization that's comfortable experimenting on live traffic.** Then again, if they already do A/B testing for product changes on live traffic, this is just extending that idea to dev changes. But if you can make it work, getting live feedback on changes you **just** made is amazing.*

The emphasis is mine, and this, in my opinion, is the biggest stumbling block to gaining more confidence about the systems we build. By and large, the biggest impediment to adopting a more comprehensive approach to testing is the required shift in mindset. Pre-production testing is something ingrained in software engineers from the very beginning of their careers whereas the idea of experimenting with live traffic is either seen as the preserve of Operations engineers or is something that's met with alarm and/or FUD.

We're acclimatized to the status quo of upholding production as sacrosanct and not to be fiddled around with, even if that means we only ever verify in environments which 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 it's indubitable that there are *some* benefits to this, it's not quite the same as performing in front of a full-house.

In fact, many of the responses to Sarah's points corroborate this. As Sarah responds,

So this thread (above) took off, & my mentions are full of well-actually. The most interesting one is "but your users will see more errors!" There are

*so many subtle misconceptions in that statement. Let's see if we can unpack it a bit. Shifting responsibility for regression catching away from tests, toward production monitoring, likely means users will generate more errors. As a result, you **cannot do this** without also changing your codebase such that errors are less noticed by (& less impactful on) your users. It's actually a nice positive design pressure that can vastly improve your users' experiences overall. It's really funny to me how some folks went "users generate more errors" -> "users see more errors" -> "you don't care about your users!" Every link in that chain of reasoning is weak.*

This hits the nail on the head. Pushing 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* for failure when the heretofore default was coding for success. And that's a notion, I'd wager, a substantial number of developers aren't too comfortable with.

What to test in production and what to test pre-production?

Inasmuch as testing of services is a spectrum, it's important for *both* forms of testing to be primary considerations at the time of system design (*both* architecture as well as code), since it then unlocks the ability to decide what functionality of a system absolutely *must* be verified pre-production and what characteristics (more like a very long tail of *idiosyncrasies*) lend themselves better to being explored in production with the help of more comprehensive instrumentation and tooling.

As to where the boundaries lie and what functionality falls precisely where on the spectrum is something only the development and Operations teams can decide, and to reiterate, it *should* be a part of system design. A "top-down" approach to testing or monitoring by treating it as an afterthought has proven to be hopelessly ineffective so far.



Cindy Sridharan

@copyconstruct

Hiring an "SRE team" won't sprinkle some reliability on your services 🙄

The top-down approach to reliability and stability hasn't worked so far.

Charity Majors gave a talk at Strangeloop this year, where she spoke about how the difference between Observability and “monitoring” really boils down to the known-unknowns and the unknown-unknowns.

“Photos are loading slowly for some people. Why?”
(microservices)

Any microservices running on c2.4xlarge instances and PIOPS storage in us-east-1b has a 1/20 chance of running on degraded hardware, and will take 20x longer to complete for requests that hit the disk with a blocking call. **This disproportionately impacts people looking at older archives due to our fanout model.**

Canadian users who are using the French language pack on the iPad running iOS 9, are hitting a firmware condition which makes it fail saving to local cache ... **which is why it FEELS like photos are loading slowly**

Our newest SDK makes db queries sequentially if the developer has enabled an optional feature flag. **Working as intended; the reporters all had debug mode enabled. But flag should be renamed for clarity sake.**

wtf do i 'monitor' for?!

Monitoring?!?

From Charity Majors' Strangeloop 2017 talk

Charity is right—these aren't the sort of things you'd ideally want to be “monitoring”. In the same vein, these aren't also the things you'd want to be “testing” pre-production. Distributed systems are pathologically unpredictable and it's impossible to envisage the *combinatorial* number of quagmires various parts of the system might end up in. The sooner we come to terms with the fact that it's a fool's errand to try to predict every possible way in which a service might be exercised and write a regression test case for it, the sooner we're likely to embrace a less dysfunctional approach to testing. As Fred Hébert in his review of this post noted:

*... as a large service with many computers involved grows, there's an increasing chance that the system will never run while 100% healthy. There's always going to be a partial failure somewhere. **If the tests require 100% health to operate, you know you've got a problem.***

In the past I've argued that "monitoring everything" is an anti-pattern. I feel the same philosophy can be extended to testing as well. One simply cannot—and as such *should not* attempt to—test *everything*. The SRE book states that:

It turns out that past a certain point, however, increasing reliability is worse for a service (and its users) rather than better! Extreme reliability comes at a cost: maximizing stability limits how fast new features can be developed and how quickly products can be delivered to users, and dramatically increases their cost, which in turn reduces the number of features a team can afford to offer.

*Our goal is to explicitly align the risk taken by a given service with the risk the business is willing to bear. We strive to make a service reliable enough, but no **more** reliable than it needs to be.*

If you replaced the word "reliability" in the above excerpt with "testing", it still would read just as sound a piece of advice.

Which then begs the question—pray *what* is better suited to be tested pre-production and what lends itself better to post-production testing?

Exploration is NOT for pre-production testing

Exploratory testing is an approach to testing that has been around since the 80's. Practiced mostly by professional testers, exploratory testing was something deemed to require less preparation on the part of the tester, uncover crucial bugs and prove to be more "intellectually stimulating than execution of scripted tests". I've never been a professional tester nor worked in an organization that had a separate team of software testers so that might explain why I only learned about this form of testing recently.

The book Lessons Learned in Software Testing has a piece of really good advice in the chapter *Thinking like a Tester*, which states: **To test, you must explore.**

*To test something well, you have to work with it. You must get into it. This is an exploratory process, even if you have a perfect description of the product. Unless you explore that specification, either in your mind's eye or by working with the product itself, **the tests you conceive will be superficial**. Even after you explore enough of the product to understand it deeply, there is still the matter of exploring for problems. Because all testing is sampling, and your sample can never be complete, exploratory thinking has a role throughout the test project as you seek to maximize the value of testing.*

By exploration, we mean purposeful wandering: navigating through a space with a gentle mission, but without a prescribed route. Exploration involves continuous learning and experimenting. There's a lot of backtracking, repetition and other processes that look like waste to the untrained eye.

The emphasis is mine, and if you replace every occurrence of the word “product” with “service” in the excerpt above, it captures my belief about the best we can hope to achieve with pre-production testing of microservices, in that, **pre-production tests are largely superficial**.

Moreover, while I definitely agree with the importance of being able to explore, I don't think it necessarily belongs in the pre-production testing phase. The book goes into further detail about how best to incorporate exploration in testing:

Exploration is detective work. It's an open-ended search. Think of exploration as moving through a space. It involves forward, backward and lateral thinking. You make progress by building better models of the product. These models then allow you to design effective tests.

Forward thinking: Work from what you know to what you don't know.

Backward thinking: Work from what you suspect or imagine back toward what you know, trying to confirm or refute conjectures.

Lateral thinking: Let your work be distracted by ideas that pop into your head, exploring tangents then returning to the main thread.

While it might be crucial to perform this form of testing pre-release while building safety critical systems or financial systems or maybe

even mobile applications, in my experience building infrastructural *services*, such exploration is vastly better suited while debugging issues or experimenting in production as opposed to putting the cart before the horse and prematurely incorporating such an open-ended approach during the service development stage.

This is owing to the fact that a better understanding of the service's performance characteristics is often obtained after observing it in production, and exploration becomes *vastly* more fruitful when undertaken armed with some form of evidence and not pure conjecture. It's also important to note that in the absence of exceptionally operationally savvy developers who can be trusted to safely perform such exploration in production, such experimentation often necessitates state-of-the-art tooling with the requisite safety catches in place, in addition to a good rapport between the Product and Infrastructure/Operations teams. Or as Sarah states:

*This is one of the reasons why it's critical for operations & development folks in an organization to work reeeeeeally closely together. An adversarial relationship between dev & ops kills this. You can't even start if responsibility for 'quality' is siloed on one side. **Certainly hybrid approaches are possible, where there's a small, fast test suite hitting critical code, & the rest is monitored in prod.***

Developers need to get comfortable with the idea of testing and evolving their systems based on the sort of accurate feedback they can only derive by observing the way these systems behave in production. Sole reliance on pre-production testing won't stand them in good stead, not just for the future but also for the increasingly distributed present of even the most nominally non-trivial architecture.

I don't believe everyone is an Operations engineer. But I do absolutely believe that Operations is a shared responsibility. Grasping the basics of mechanical sympathy (understanding how the hardware works) can make one a better developer. The same can be said about *operational sympathy* (understanding how the service works in production).

[@mipsytipsy](#) [@ErikSeaberg](#) [@colourmeamused](#)

1000% Failure gonna happen. Hell, it may not even be something you did. (I'm looking at you S3zure) The reason is not deployment, the reason is Internets. Unless you put your software in a box you work in ops now. Code accordingly.

Developers need to learn to code (and test!) accordingly. I usually find such proclamations more handwavy and aspirational than actionable. What does it mean for a developer to “code accordingly”?

In my opinion, this 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

This includes writing code while concerning oneself with questions such as:

- how a service is deployed and with what tooling
- is the service binding to port 0 or to a standard port?
- how does an application handle signals?
- how does the process start on a given host
- how does it register with service discovery?
- how does it discover upstreams?
- how is 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 or purely single threaded and event driven or 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 the aforementioned examples 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. Case in point—Fred Hébert in his review of this post points out that:

the TCP stack behaves differently when working with the loopback interface and a remote one. For example, if a connection breaks, the loopback interface can let you know about it while doing a TCP ‘send’ operation; on a non-loopback interface, this rarely happens and you need to read the packets from the buffer to see a FIN (RST can kill anyway). There is therefore no way to ever find connection hangup issues of that kind if your test rig does not at least use a non-loopback interface.

Operational characteristics of the dependencies

We build on top of increasingly leaky (and oftentimes frangible) abstractions with failure modes that are not well-understood. Examples of such characteristics I’ve had to be conversant with in the last three years have been:

- the default read consistency mode of the Consul client library (the default is usually “strongly consistent”, which isn’t something you might 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 an evented 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

Debuggable code

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

- instrumenting code well enough
- having an understanding of the Observability 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 Observability format given the requirements of the given service, operational quirks of the dependencies and good engineering intuition

If all of the above sounds daunting, then well, it is, and “coding accordingly” (and more crucially, being able to “*test* accordingly”)

entails being able to appreciate these challenges and bracing oneself toward stepping up to the plate.

Pre-production testing

Having made the case for a hybrid approach to testing, let's get to the crux of what the rest of this post is about—pre-production testing of microservices.

Sarah goes on to state that:

The writing and running of tests is not a goal in and of itself—EVER. We do it to get some benefit for our team, or the business. If you can find a more efficient way to get those benefits that works for your team & larger org, you should **absolutely** take it. Because your competitor probably is.

Quite like anything else in software development, it's possible to fetishize testing of *any sort* (be it traditional testing or monitoring or exploration and what have you) to the point where more than being a means to an end it becomes a religion of sorts. Testing isn't an absolute, nor are there any standardized set of metrics that can serve as a universal yardstick of a well tested system.

However, four axes—the goal, scope, tradeoffs and payoffs—can prove to be a good proxy for being able to assess how effective any form of testing might be.

The Goal of pre-production testing

As stated previously, I view pre-production testing as a *best effort verification* of the correctness of a system as well as a *best effort simulation* of the known failure modes. The *goal* of pre-production testing, as such, isn't to prove there aren't any bugs (except perhaps in parsers and any application that deals with money or safety), but to assure that the known-knowns are well covered and the known-unknowns have instrumentation in place for.

The Scope of pre-production testing

The *scope* of pre-production testing is only as good as our ability to conceive good heuristics that might prove to be a precursor of production bugs. This includes being able to approximate or intuit the boundaries of the system, the happy code paths (success cases) and

more importantly the sad paths (error and exception handling), and continuously refine these heuristics over time.

In my experience, I've invariably been the person writing the code *and* the test. Even when my code is reviewed, it is still by someone on the same team. As such, any scope is heavily curtailed by the inherent bias that the engineers on the team building the system might hold, as well as any of the implicit assumptions the system is built upon.

Given these goals and scope, let's assess the various forms of pre-production testing and look at the tradeoffs and payoffs.

Unit testing

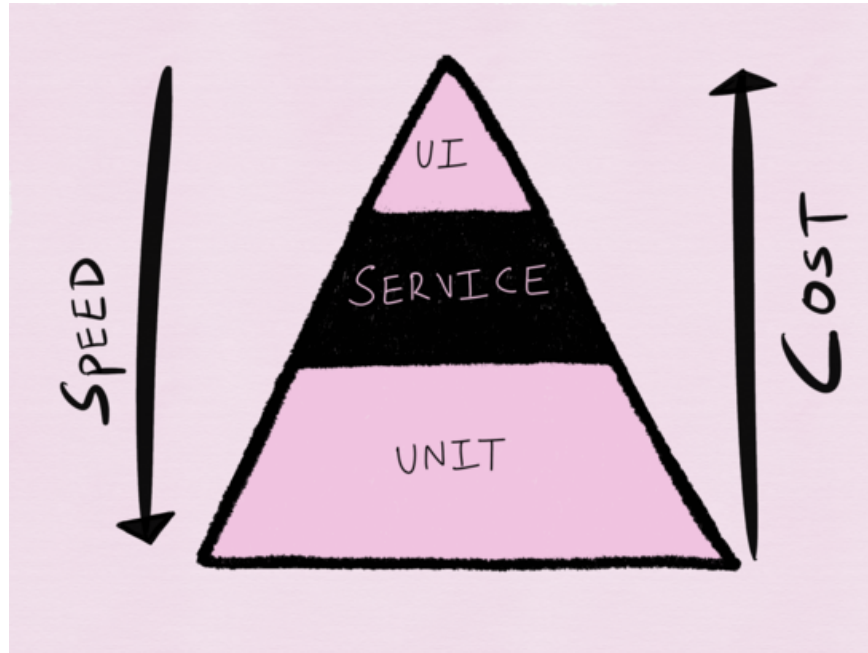
Microservices are built on the notion of splitting up units of business logic into standalone services in-keeping with the single responsibility principle, where every individual service is responsible for a standalone piece of business or infrastructural functionality. These services then communicate with each other over the network either via some form of synchronous RPC mechanism or asynchronous message passing.

Some proponents of microservices call for using a standardized template for intra-service organization of components, so that all the services end up looking structurally alike and are comprised of many components that can be layered together in a manner that is conducive to testing each layer in isolation.

Put differently, this approach (rightfully) champions for the code-level decomposition of every individual service in order to decouple the different domains, so that networking logic is separate from the protocol parsing logic, business logic and data persistence logic. Historically, libraries were used to achieve these goals (so you had a library to parse JSON and another library to talk to your datastore and so forth), and one of the oft-repeated benefits of this form of decomposition is that each of these layers can be unit tested individually. Unit testing is universally considered to be a cheap and fast way to test the smallest possible unit of functionality.

No discussion on testing is complete without mentioning the test pyramid proposed by Mike Cohn in his book *Succeeding with Agile*, and cited by Sam Newman in his book *Building Microservices: Designing Fine-Grained Systems*. At the bottom we have unit tests which are

considered to be both fast as well as cheap, followed by service tests (or integration tests) and then UI-driven tests (or end-to-end tests). As you ascend the pyramid, the speed of the tests decreases along with a concomitant increase in the cost incurred to perform the test.



Mike Cohn's test pyramid

The test pyramid was conceived during the era of the monolith and makes a lot of sense when we think about testing such applications. For testing distributed systems, I find this approach to be not just antiquated but also insufficient.

In my experience, an individual microservice (with the exception of perhaps network proxies) is almost always a software frontend (with a dollop of business logic) to some sort of stateful backend like a database or a cache. In such systems, most, if not *all*, rudimentary units of functionality often involve some form of (hopefully non-blocking) I/O—be it reading bytes off the wire or reading some data from disk.

Not all I/O is equal

Cory Benfield gave a great talk at PyCon 2016, where he argued that most libraries make the mistake of not separating protocol parsing from I/O, which makes both testing and code reuse really hard. This is certainly very true and something I absolutely agree with.

However, I also believe that not all I/O is equal. Protocol parsing libraries, RPC clients, database drivers, AMQP clients and so forth all perform I/O, yet these are different forms of I/O with different *stakes* given the bounded context of the microservice.

For example, when testing a microservice that's responsible for managing users, it's more important to be able to verify if users can be created successfully in the database, as opposed to testing if the HTTP parsing works as expected. Sure, a bug in the HTTP parsing library can act as a single point of failure for this service, but all the same HTTP parsing only has a supporting part to play in the grand scheme of things and is *incidental* to the primary responsibility of the service. An HTTP library is also something that's going to be reused by all services (and as such could benefit from rigorous fuzzing) and thereby becomes a *shared abstraction*, and the best approach for making peace when working with abstractions—even the leakiest ones—is to (grudgingly) repose trust in the promised contract. While hardly ideal, this tradeoff makes most sense to me in respect of getting anything shipped at all.

However, when it comes to the microservice itself, *the* abstraction it is offering to the rest of the system is one encapsulating state transitions that directly map to a certain piece of business or infrastructural logic, and ergo it is *this* functionality that needs to be tested as an airtight unit.

Similarly, given a network proxy that uses Zookeeper for service discovery to load balance requests to dynamic backends, it's vastly more important to be able to test if the proxy responds correctly to a watch triggering by changing its internal state and (possibly) also setting a new watch. *That* is the unit under test there.

What is of the essence here is that *the* most important unit of functionality a microservice provides happens to be an abstraction of the underlying I/O involved to its persistent backend, and as such should become *the* hermetic unit of base functionality under test.

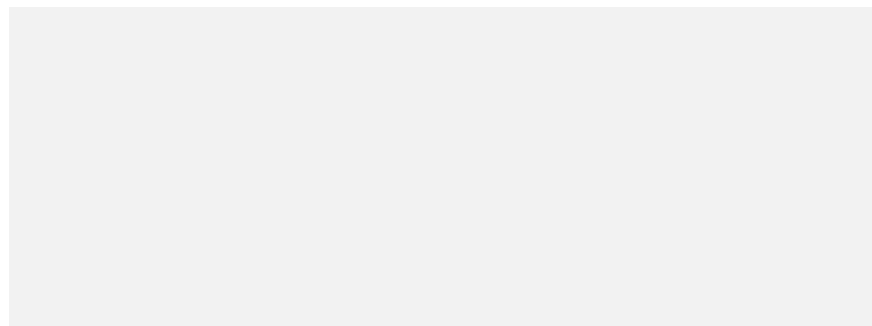
And yet, the prevalent best-practice of testing such systems is by treating the underlying I/O not as an integral part of the unit under test but as a nuisance that needs to be warded away with mocks, to the point where *all* unit-testing these days has become synonymous with heavy mock usage.

Unit testing such service-critical I/O with mocks inherently embodies a sellout since it not just sacrifices accuracy at the altar of speed, but also ends up shaping our mental model in a way that's almost *entirely dissonant* with the actual characteristics of the system we're building. In fact, I'd go so far as to say that unit testing with mocks (we might as well call this *mock testing*), for the most part, is tantamount to validating our incomplete (and also very possibly flawed) mental model of the most business critical components of the systems we're authoring and serves as one of the most insidious forms of confirmation bias.

[@copyconstruct](#) I like the thoughts on how mocking impacts our mental model. Mocks respond predictably. Networks and databases do not.

— [@taotetek](#)

The biggest weakness of mocks when used as a *testing* tool is that both when simulating success as well as failure, mocks are a programmer's mirage of a *sliver* of a future they cannot fathomably appreciate or even approximate in their minds during feature development time.



Now one could argue that this problem can be better tackled by observing in production *all possible failures* at the network level and accordingly beef up the test suite with additional mocks to account for *all of these cases* previously unaccounted for, but more than being guaranteed to account for every possible network level bug one could run into, this approach is more likely to lead to a bloated test suite with

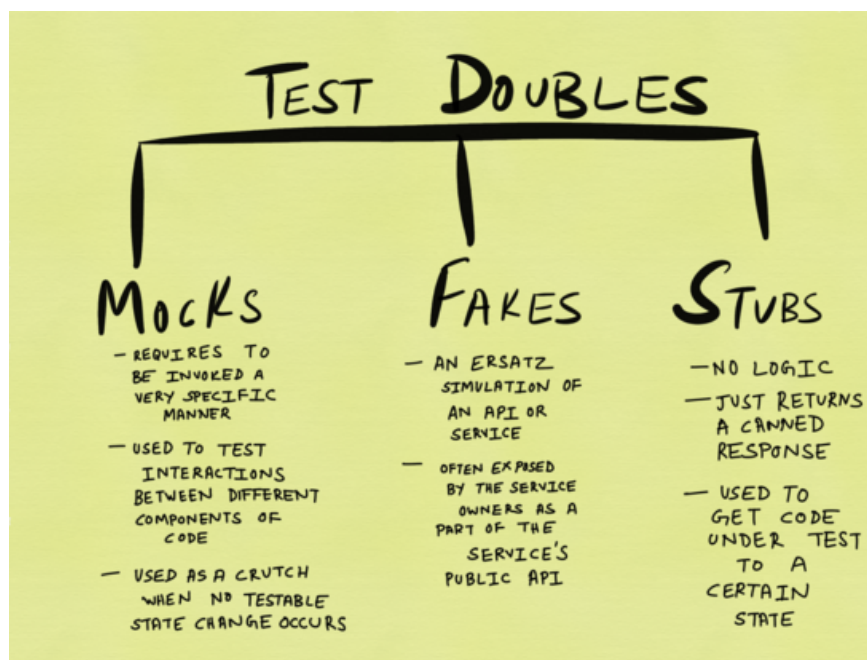
a variety of mocks for the similar functionality which subsequently becomes a maintenance burden.

[@kartar](#) [@copyconstruct](#) Third: by pouring concrete around everything, you now have 1) brittle, hard-to-mutate codebase with much higher changeset write amplification, 2) 3x duplication of domain in a) real-world; codebase; tests. The programmer with 3 watches never knows what time it is what time it is.

Speaking of the maintenance burden of test code, another downside of mocks is that they can make the test code incredibly verbose and/or difficult to comprehend. In my experience, this is especially true when *entire classes* get mocked out and the mock implementation is injected into the test as a dependency, only for the mock to then assert if a certain method on that class was invoked a certain number of times or with certain parameters. Such form of mocking is extensively used in some circles for behavior verification.

[@kartar](#) [@copyconstruct](#) My synopsis of mock objects in tests: you end up testing the implementation, not the behavior, in nearly all cases. This is stupid: the first rule of tautology club is the first rule of tautology club. Secondly, you multiply your codebase horribly. Result: much harder to change.

Making peace with mocking

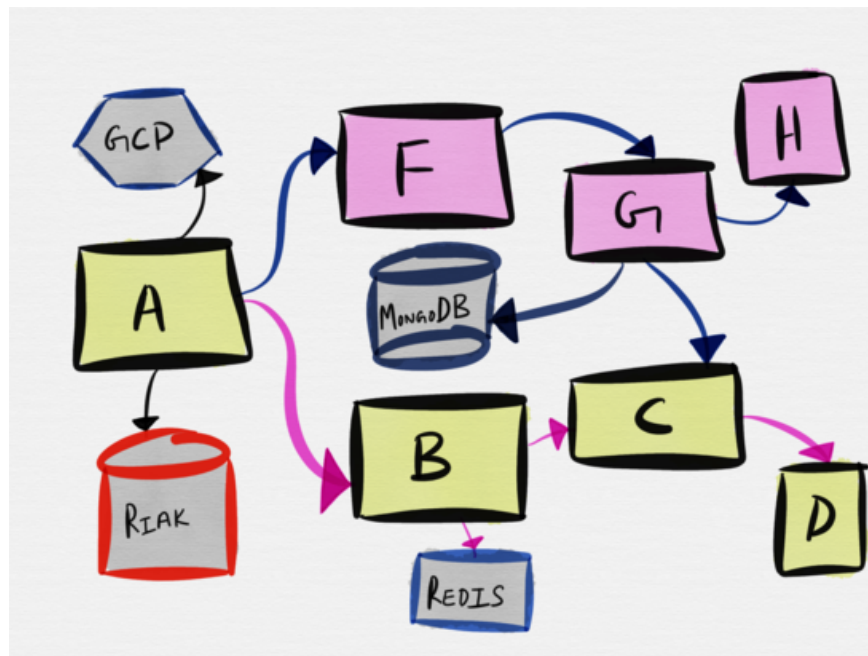


From Google's testing blog on knowing your doubles

Mocks, stubs and fakes are forms of what's called *test doubles*.

Everything I've written about mocks above is largely applicable to other forms of test doubles as well. However, that isn't to say mocks (and other variants) don't have any benefits at all, nor am I suggesting that unit tests should *always* involve I/O in lieu of test doubles. It only makes sense for unit tests to involve I/O if the test involves a *single* I/O operation sans any further side-effects that need to be accounted for (which is one of the reasons I don't see the value in testing publish-subscribe workflows in this fashion).

Consider the following topology—a very plausible example of a microservice architecture.



Service A's interaction with service B involves service B talking to Redis and service C. However, the smallest *unit* to be tested really is **service A's interaction with service B**, and the easiest way to test that interaction is by spinning up a fake for service B and testing service A's interaction with the fake. Contract testing can be especially helpful for testing these sort of integrations. Soundcloud famously uses contract tests extensively for testing their 300+ microservices.

Service A also talks to Riak. The smallest unit to be tested there involves the actual **communication between service A and Riak**, therefore spinning up a local instance of Riak during test makes sense.

When it comes to integration with third party services like GCP (or AWS or Dropbox or Twilio), the ideal scenario is when the language bindings or SDKs provided by these vendors come with good fakes to incorporate into one's test suite. It'd also be of great assistance if these vendors provided the ability to make real API calls but in a test or dummy mode, since having such flexibility makes it feasible for developers to test in a more authentic manner. Stripe, for example, does a really good job of this by providing test mode tokens.

All things considered, test doubles have their place in the testing spectrum. But they aren't the *only* means of performing unit tests and in my opinion work best when used sparingly.

The myriad unsung benefits of unit tests

There's more to unit testing than what's been discussed heretofore in this post. It'd be remiss to not talk about property-based testing and fuzzing while I'm on the topic of unit testing. Popularized by the [QuickCheck](#) library in Haskell (since then ported to Scala and other languages) and the [Hypothesis](#) library in Python, property-based testing allows one to run the same test multiple times over with varying inputs without requiring the programmer to generate a fixed set of inputs in the test case. [Jessica Kerr](#) has [the best talk](#) I've ever seen on the topic of property-based testing. [Fred Hébert](#) has an [entire book on the topic](#) for those interested in learning more, and in his review of this post, cast more light on the different types of approaches of different property-based tools:

On property-based testing, a good portion of tools make it close-to-equal to fuzzing, but with a twist that you can do it in a whitebox manner, and with a finer-grained approach. To put it another way, fuzzing tends to be about finding whether a part of the system crashes or not, whereas property-based testing tries to check whether a certain behavior or set of rules (properties) are always upheld in the system. There are 3 big families of property tests:

— **The Haskell Quickcheck variants:** *Those are based on using type-level information to generate data that will exercise the test. Their biggest selling point is how concise they make tests for the amount of coverage you get, and the breadth of examples they'll find. They are difficult to scale up due to 'shrinking', directing the framework how to simplify failing counter-examples*

— **The Erlang QuickCheck variants:** *Those are based on dynamic data generators that are composable functions. The type-level approach of basic Haskell Quickcheck is still there, but the frameworks also come with a set of stateful modeling primitives. The most concise way to describe it is that this would be equivalent to doing model checking, but instead of doing an exhaustive search, a probabilistic one is done. So we're leaving the fuzzing area and getting closer to model-checking, which is an entirely different family of test practices. I've seen talks where they ran the property tests over a specific cloud providers and used it to find hardware errors*

— **Hypothesis:** *which has a unique approach to things. It is based on a fuzzer-kind of mechanics where data generation is based off a byte stream that can get higher or lower in complexity. Hypothesis is unique in how it*

runs under the cover and probably the most usable of all variants here. It does have a different approach to how things run and I'm not too familiar, but it must be said it offers a lot more than what Haskell variants do as well.

Fuzzing, on the other hand, concerns with feeding knowingly invalid and junk inputs to an application to see if the application fails accordingly.



There are different sorts of tools available for fuzzing—there are coverage based fuzzers like [afl](#) as well as tools like the [address sanitizer](#), [thread sanitizer](#), [memory sanitizer](#), [undefined behavior sanitizer](#) and the [leak sanitizer](#) to name a few.

Unit testing can have various other benefits other than just verifying something works as expected for a certain input. Tests can act as fantastic documentation for the API exposed by an application. Languages like Go allow for *example tests*, where functions that begin with `Example` instead of `Test` live along with the normal tests in the `_test.go` file of any given package. These example functions are compiled (and optionally executed) as part of a package's test suite and are then displayed as a part of the package documentation allowing users to run them as tests. The mainspring, as [the docs state](#), is that:

Having executable documentation for a package guarantees that the information will not go out of date as the API changes.

Another benefit of unit tests is that it applies a certain design pressure on programmers to structure the API in a way that's easy to consume. Google's testing blog has a fantastic post called **Discomfort as a Tool for Change**, which sheds light on how requiring API authors to provide fake implementations reverses the pain and makes them empathize with consumers of the API.

Like SRE but for testing API's - from the Google testing blog on the importance of why API's should ship with fakes as well as using discomfort as a tool for change.

— [@copyconstruct](#)

When I used to run the Python Twisted meetups in San Francisco in 2016, one of the topics that came up quite often was how making the event loop in Twisted a global was a mistake (something which Python 3 since went on to repeat in the asyncio implementation, much to the dismay of the Twisted community) and how much easier both testing as well as usage would've been had the Reactor (which provides basic interfaces to all manner of services, including network communications, threading, event dispatching and so forth) been passed around to the consumers as an explicit dependency.

There's a caveat, however, and it is that good API design and good testing are two entirely independent goals in their own right, and it's very much possible—though not perhaps *advisable*—to design fantastically usable and intuitive API's that lack sufficient testing. More commonly, however, it's *eminently possible* to design API's that strive to achieve one hundred percent code coverage (measured either in terms of line coverage, branch coverage or other such yardsticks) but end up being an prematurely-abstracted and DRY-to-the-point-of-scorched artifact of incongruity. While mock-based unit testing can certainly *help* guide good API design, it doesn't necessarily *ascertain* the code will invariably function as expected.

VCR—or replaying or caching of test responses

I find the overuse of mocks/fakes/test doubles/stubs to be brittle enough (but sometimes necessary), but it becomes worse to debug test failures when there's vcr style caching/replaying of stubs/fakes in action. That's like double faking, IMO. Anyone else feel this way?

What I find absolutely preposterous is given the inherent fragility of mocks, some communities take it one step further and insist on *recording* the responses and saving it as a test fixture. I find this ineffective for not just speeding up integration tests (it's really not an integration test if it's using one of these techniques), but moreover when some try to speed up *unit tests* in this fashion. The cognitive overhead involved while debugging test failures with such additional layers of indirection simply isn't worth the trouble.

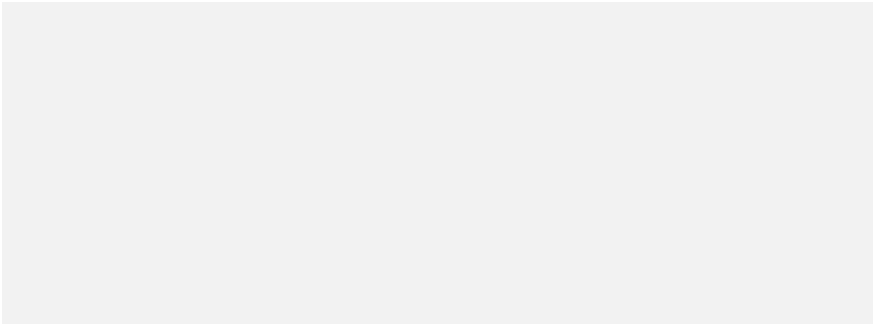
Integration Testing

If unit testing with mocks is so fraught with fragility and lacking in verisimilitude, does that mean integration testing is then the remedy to cure all ills?

In the past, I've gone on to state that:

This might sound radical - but good (and **fast**) integration testing (both local and remote) as well as good instrumentation often serve better than striving to achieve 100% unit test coverage.

— [@copyconstruct](#)



I had some people reach out to me to point out the inherent contradiction in these two statements, because seemingly on the one hand I was calling for more integration tests in lieu of unit tests, but on the other hand I was also arguing that integration testing wasn't scalable for distributed systems. These views aren't mutually exclusive and perhaps my first point requires more clarity here.

The main thrust of my argument wasn't that unit testing is *completely* obviated by end-to-end tests, but that being able to correctly identifying the "unit" under test might mean accepting that the unit test might resemble what's traditionally perceived as "integration testing" involving communication over a network.

In the most trivial of scenarios a service talks to a single datastore. In such cases, the indivisible unit under test must *unequivocally* involve the attendant I/O. At other times, the transactions are distributed in which case it becomes a lot trickier to decide what the single unit under test must be. The only company that I know of that has dabbled with distributed sagas is Uber and when I asked one of my friends who works there about how they approach testing, I couldn't get any more than that "it remains a challenge". Hypothetically, were I to implement this pattern in a new service I'm authoring, the prospect of using *mocks* or *fakes* to verify if, say, a distributed transaction aborted correctly or if the compensating transaction was applied as expected doesn't seem very foolproof. The *unit* under test becomes the long lived transaction.

Event sourcing is a pattern that's enjoying its time in the sun along with the rising popularity of Kafka. The Command Query Responsibility Segregation pattern (not without its fair share of anti-patterns) has been around for roughly a decade now and premised on the notion of separating state change (writes) from state retrieval (reads). Oftentimes what I find missing in talks about such nontrivial architectural patterns is best practices to *test* such architectures, other

than mock at the code level and have end-to-end tests at the system level.

Usually, leaving such verification to fully blown integration testing would not just invariably slow down the feedback cycle but also become the warrant for an integration testing apparatus of complexity ridiculous enough to rival or even *surpass* that of the system under test, requiring an inordinate number of engineering cycles (and very possibly an entire team) to prop up.

[@thramp](#) [@copyconstruct](#) Integration testing is not just ineffectual for complex distributed systems, it is also a black hole for infinite engineering cycles with no commensurate payoff.

— [@mipsytipsy](#)

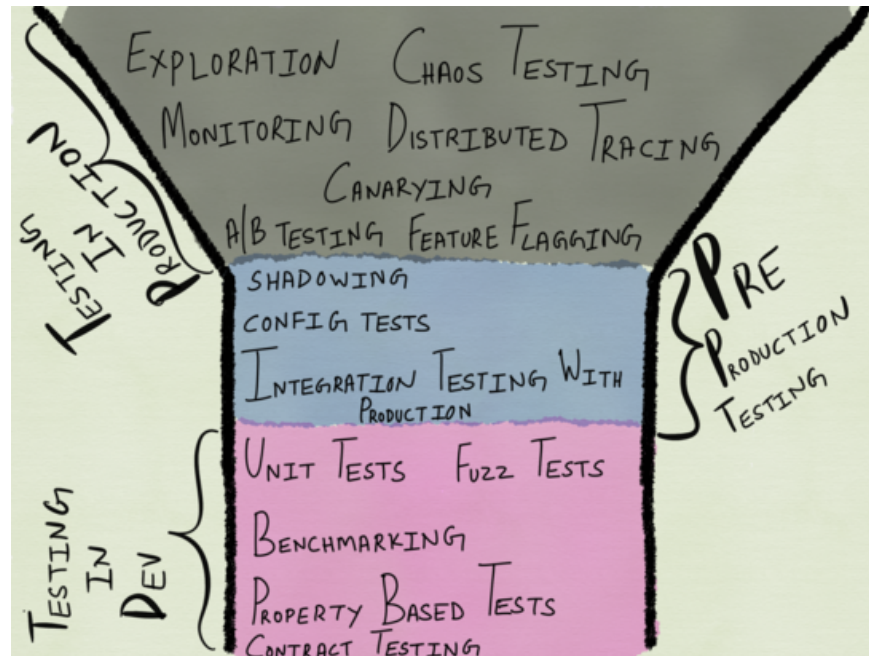
Yet another drawback of integration tests for complex systems is that it then demands maintaining separate environments for development, test, and/or staging. Many organizations try to keep these environments as identical and “in sync” as possible with production, which usually involves replaying all live traffic (or at least *writes*) to a test cluster, so that the persistent stores in the test environment match production. Either way you slice it, this involves a significant investment in automation (and personnel) to monitor and maintain.

Given these constraints and pitfalls, any discussion on how best to design integration tests involves a compromise, and the best compromise that I feel can be made is what I call the “Step-Up Rule”.

The Step Up Rule

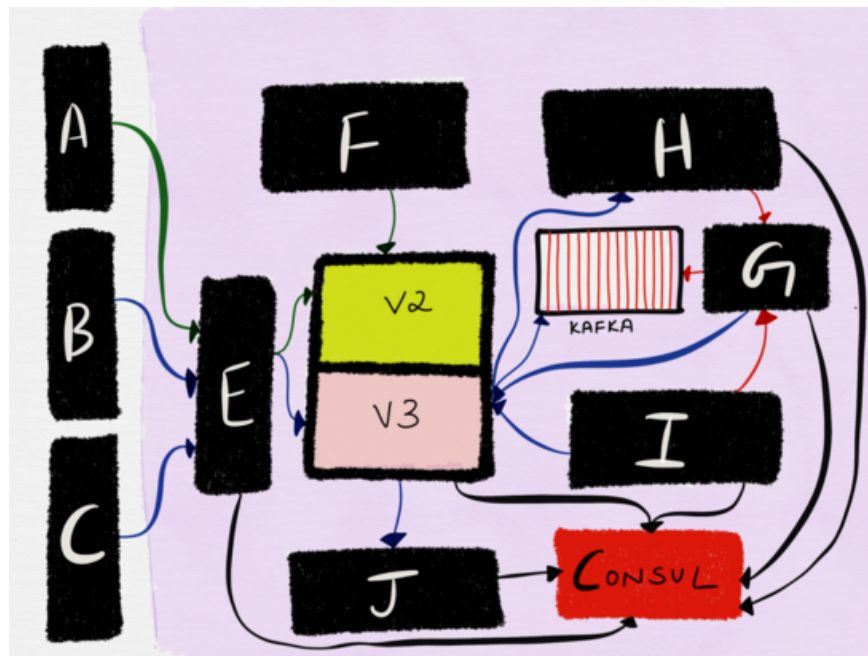
I coined the term “step-up testing”, the general idea being to test at one layer above what’s generally advocated for. Under this model, unit tests would look more like integration tests (by treating I/O as a part of the unit under test within a bounded context), integration testing would look more like testing against real production, and testing in production looks more like, well, monitoring and exploration. The

restructured test pyramid (test funnel?) for distributed systems would look like the following:



The Test Pyramid for Distributed Systems

To see how this might be applicable in a real world scenario, let's use a grossly simplified architecture (I've omitted all the backing datastores, Observability tools and other third party integrations) of the services I've been the developer on and have been on-call for the better part of the last two years as a case study. What's described below is only a small part of the entire infrastructure, and moreover, it's important for me to state upfront that the boundaries are *extremely conservative*. We only split when it cannot be avoided given the requirements. Like for instance, there's no way a load balancer can also be a distributed file system or a distributed multi-tiered caching system, hence these are different systems.



It doesn't matter what each service really does, except that they are all disparate services written in different languages which serve distinct traffic patterns. All of these services use Consul for service discovery. These services also have different deployment cadences—the central API server (with versioned endpoints) gets deployed multiple times a day whereas service G is deployed maybe twice every year.

The notion of spinning up *all* of these services on my local Macbook while developing the API service is risible (and no, I don't do it). Equally risible is spinning up all of these services in a test environment and running integration tests for every build. Instead, the decision on how best to test each individual service is made by making certain tradeoffs and bets, given each service's functionality, guarantees and access patterns.

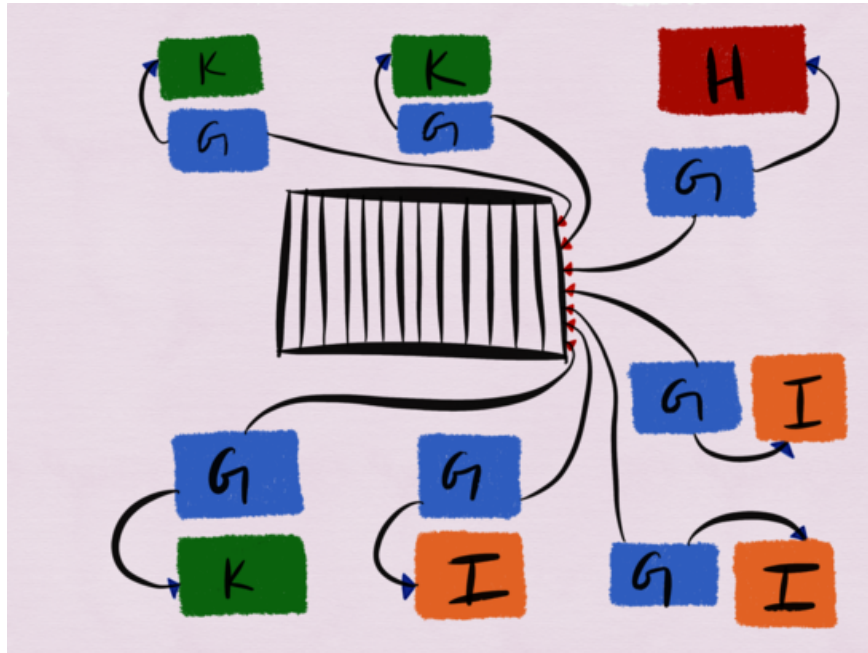
Unit tests look more like integration tests

Well over 80% of the functionality of the central API server involves communicating with MongoDB, and thus the vast number of unit tests involve actually connecting to a local MongoDB instance. Service E is a LuaJIT auth proxy that load balances traffic originating from three different sources to different instances of the central API. One of the most critical pieces of functionality of service E is to ensure that the appropriate handler gets invoked for every Consul watch it sets, and thus some unit tests actually spin up a child Consul process to communicate with it, and then kill the process when the test finishes.

These are two examples of services that were better tested with the sort of unit testing that purists might frown upon as integration tests.

Integration testing looks more like testing in production

Let's take a closer look at service G, H, I and K (not shown in the diagram above).



Service G is a single threaded Python process that subscribes to a Kafka topic to consume user driven updates and subsequently reloads service H, service I and service K's configs. Service G is colocated on every host services H, I and K are deployed to, and at any given time we have anywhere between 15–20 instances of service G running (though the diagram depicts only 7). Service G's primary responsibility is to ensure that the messages it consumes off the Kafka topic are propagated to services H, K and I.

H, K and I are all independent services in their own right. Service H is an nginx (Openresty, to be specific) frontend to a massively distributed system (not shown in the diagram) at the heart of which lies a distributed file system, in addition to a LevelDB write through cache, a MySQL metadata store, metadata trackers, file pruners and file fetchers written in Go. Service K is, for all intents and purposes, HAProxy, and service I is a LuaJIT HTTP server. Furthermore, it's required that

services H, I and K *converge* on the configuration updates in Kafka, though thankfully we don't need strong consistency guarantees.

Which begs the question—how then to best *test* the eventual consistency of these systems?

The way these services are tested *isn't* by spinning up *all* of these services in concert and running an end-to-end test to verify any publish to the Kafka topic ends up being consumed by service G correctly, followed by services H, I and K converging on this configuration. This is the sort of thing which, even if verified in a “test environment”, wouldn't be a harbinger of production correctness, especially given:

— service K is HAProxy (and true zero downtime HAProxy reloads wasn't really a thing until very recently)

—service H in production serves hundreds (to sometimes thousands) of requests per second *per process*, something that can't be easily reproduced in a test environment without significant investment in tooling and load generation automation. Also it'd require us to test if reloading the nginx master process of service H correctly results in the launch of a bunch of fresh worker processes. A known production issue with service H is that a reload of the nginx master process occasionally *doesn't* result in the old worker processes being killed in a timely manner, leading to stale workers hanging around and serving traffic, an occurrence an official nginx blog post called “very rare”:

Very rarely, issues arise when there are many generations of NGINX worker processes waiting for connections to close, but even those are quickly resolved.

In practice, at least with HTTP2, “rare” wasn't really the case until recently, and in my experience, the easiest fix to this problem was to monitor for the stale nginx workers and killing them. Convergence lag of the three services H, I and K is again something that's monitored and has an alert in place for.

This was but one example of a system that didn't stand much to benefit from integration testing and where monitoring has worked *much* better. While it's certainly possible that we could've simulated *all* of these failure modes in an integration test environment, it still would've remained something that required monitoring anyway, and any

possible benefit that integration testing would've provided us here wouldn't have been worth the overhead.

Traffic Shaping with Service Meshes

One of the reasons I'm so excited about the emerging service mesh paradigm is because a proxy enables traffic shaping in a way that's *extremely* conducive to testing. With a small amount of logic in the proxy to route staging traffic to the staging instance (which can be achieved with something as simple as setting a specific HTTP header in all non-production requests or based on the IP address of the incoming request), one can end up exercising the actual production stack for all *but* the service in question. This enables performing real integration testing with production services without the overhead of maintaining an ornate test environment.

My belief was that integration testing in "test" clusters is useless and should die - live traffic testing is the future. TIL Facebook's been doing it for a long time. <https://t.co/zMrt1YXaB1> With service meshes unlock so much potential for better testing to us unwashed masses.

Of course, I'm papering over the nitty-gritty of security compliance, data integrity and so forth, but I genuinely believe that live traffic testing with good Observability into the impact of the tests being conducted is the way forward for testing microservices.

Opting into this model of testing has the additional benefit of incentivizing better isolation between services and the design of better systems. Requiring engineers to think hard about inter-service dependencies and how these might affect data integrity is a neat architectural design pressure. Being able to test one service against *all others* in production would also require that the service in test doesn't have any side-effects on any of its upstream or downstream dependencies, which seems like a very reasonable design goal to me.

Conclusion

The goal of this post wasn't to make an argument for one form of testing over another. I'm hardly an expert in any of these things and I'd be the first to admit that my thinking has been primarily shaped by the type of systems I've worked with, the constraints I've had to deal with (chiefly very limited time and resources) and the organizational dynamics of the companies I've worked at. It might be very possible that none of what I posited in this post hold water in different scenarios with differing contexts.

Saying this again because it's worth reiterating: Listen to what the so called "experts" say, but please think for yourself. "Experts" often generalize advice. *You're* the expert given your specific context and needs. Thinking cannot be outsourced. It's an example of laziness

Given how broad a spectrum testing is, there's really no One True Way of doing it right. Any approach is going to involve making compromises and tradeoffs.

Ultimately, every individual team is the expert given the specific context and needs.

Thinking cannot be outsourced.

