

Testing for Software Engineers

Learn everything you wanted to know
about testing but didn't dare to ask.

Mikael Vesavuori

Testing for Software Engineers

Mikael Vesavuori

2024

Contents

Copyright	8
Found anything wrong?	9
Introduction	10
What will you learn?	10
Why write this book?	11
Audience	12
The project source material	12
How the book is structured	13
What is a test?	15
A basic unit test	16
Is there a need for dedicated QA?	17
Checking vs testing?	17
Continuously testing your software	18
The testing-industrial complex and how it denies competent practices . . .	20
Knee-deep in it	20
Hijacking Agile	22
Why test?	25
Why automate testing?	27
In closing	29
Can something be untestable?	31
The potential role of TDD	32
Good code is the way to testability	32
Manual testing is useful, but not often	34
The first time you do something	34
Collecting initial test data	34
Basic checks and verifications	35
Exploratory testing	35
In closing	38
Project resources	39
Getting started	40

Pre-requisites	40
Building testable systems	42
What our “good” looks like	43
Getting Unstuck from the Tyranny of Local Optima	43
Build better without more resources	50
Using a yardstick to measure team performance	50
Pushing for change	54
Build deterministic systems	56
It starts with knowing what you are actually trying to do	56
No good test without good code	58
Coupling and cohesion	60
Inversion of control	61
In summary	62
Integrate continuously, test continuously	64
Use competent CI/CD tooling and automate all of it	64
Insist on multiple integrations and deployments per day	67
In summary	68
Separate deployments from releases with feature toggles	69
A simple example	70
Some concerns you will have to address	72
Use modern technical practices	74
Work in a DevOps model	74
Own your CI/CD pipe	75
Configure your code with infrastructure-as-code (IAC) tools	75
Migrate to Typescript if you are using JavaScript	75
Refactoring	76
Use a well-considered software architecture	77
Be literate with the cloud, including microservices and serverless	77
Consider TDD	78
Design and think before coding	79
Write, make bullet points, doodle, or diagram your way to a design	80
Use scratch files to remove complexity from your problem solving	81
Write the “API” before the code	82
Lean towards use-case driven APIs	83
A system as a set of use cases	83
Using Clean Architecture as our foundation	84
In closing	87

Provide an API schema and examples	88
Offload some validation to the API level	90
Testing validation	90
Towards modern testing practices	91
Types of testing	93
Why so many types of testing?	94
The purpose of a test	95
What to focus on	96
Does anyone actually run all those tests?	97
Pre-production testing vs production testing	98
In closing	98
Testing models	99
Testing ice cream cone (unknown origin; God?)	100
Testing pyramid (Mike Cohn)	101
Testing trophy (Kent C. Dodds)	102
My recommended model?	104
Choosing a model	104
Confidence-based testing	105
Choose high-quality automatable tools	108
First: How can you test your software?	108
Open source tooling or purchased?	108
In-context tests	109
Don't fall in love with your test tooling	109
Writing good tests	110
The FIRST principles of testing	111
Use the Arrange Act Assert (3 A's) format	112
Test for behavior, not for implementation	112
Mock as little as possible and avoid specific tools for it as far as possible	113
Should I have one test per class/function/whatever?	113
What is the test coverage to go for?	115
Cultivate a gut feeling for how much testing is "enough"	115
Scoping the test	117
Example with AVA	118
Example with Jest	119
Prefer fast and reliable: On unit vs integration testing	121
Understand and question your goals with testing on infrastructure	123
Handle and mock side effects	126

Temporary environments	127
Tagged data	127
Mocking	127
Test environments	132
What is the impact of a caught, or uncaught, issue?	132
Where do we run the tests? Splitting either by hardware or software	133
What do you think you are gaining from testing on separate infrastructure?	135
Use infrastructure-as-code to keep environments exactly as configured and as similar to each other as possible	136
Set up unique test environments for each system	137
Benefit from the upsides of using ephemeral environments	137
In closing	140
Testing in production	141
Test data management	146
What about duplication and/or sharing of test data?	147
Advice for test data management	148
Testing serverless isn't that different	153
How serverless (and the cloud in general) clarifies responsibility	153
Impact of microservices and serverless on testing	156
In closing	158
Testing considered harmful	160
Over-reliance on QA and/or "over the wall" testing	160
TDD misinterpretations	161
Too many tests (i.e. redundant tests)	162
Too few tests	164
Blind trust in code coverage	165
Brittle and flaky tests	166
Be overly framework-dependent	166
Running tests in practice	168
Static code analysis	169
Using ESLint and Prettier in the IDE	170
As a tool during CI	174
Unit testing	176
Considering the limits of unit tests	177
Examples	180
Mocking API dependencies (HTTP calls) with MSW	183

In closing	184
Smoke testing	185
You still need to handle side effects	187
API (integration) testing	188
Running tool-centric tests	189
Writing our own integration testing tool	189
Are you integration testing someone else's thing?	195
In closing	196
Synthetic testing	197
Comparing to RUM (real user monitoring)	200
In closing	201
Load testing	202
System testing	207
A partial example of a system test	209
In closing	214
UI end-to-end testing	215
E2E and its bad reputation	217
Using Codecept to do UI testing	218
Good or bad demonstration?	219
Remaining scenarios	220
In closing	221
Contract testing	222
Examples	224
In closing	229
Continuous testing in CI	230
Scripting a CI pipeline to enable continuous testing	231
Need for speed: Pushing CI times	236
Breaking down a big script into smaller ones	237
Example testing scenarios	239
Front-end application	240
Start with cleaning up, then introduce unit tests and static code analysis	241
Checking the full flow (or as much of it as possible)	242
Errors need to go to some tool	246
Back-end relying on third parties	247
Improving the API	248
Writing the unit tests	248
Integration tests	249

Serverless, distributed, event-driven application	250
Define the API	251
Define events	251
Unit test the interfaces	252
Ensuring fit with a system test	253
In summary	254
References and resources	256
Books	256
Code	257
Tools	257
Websites	260
Articles	261
Thank you for reading this book	264

Copyright

© 2024 Mikael Vesavuori. All rights reserved.

No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including but not limited to physical copies, photocopying, recording, electronic books (eBooks), PDFs, digital downloads, or any other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law, such as under “fair use”.

Cover image adapts photographic material shared by Joel Filipe on Unsplash. All relevant ownership of the original photograph remains with Joel Filipe.

Published by Mikael Vesavuori on LeanPub and Gumroad.

Found anything wrong?

Writing technical books is challenging. While concepts and ideas may remain relevant for years, practical examples that rely on ever-changing technologies can become outdated quickly.

If you find anything incorrect, not working, or otherwise unusual, I'd greatly appreciate your feedback. I'll do my best to incorporate updates as soon as possible.

Find contact details on my website, mikaelvesavuori.se.

Introduction

Inspection does not improve the quality, nor guarantee quality. Inspection is too late. The quality, good or bad, is already in the product. As Harold F. Dodge said, “You can not inspect quality into a product.”

— [W. Edwards Deming](#)

In a modern technical environment, testing, development, DevOps, and business merge—for better or worse. With 20+ years of Agile practices behind us, helping us stay true to a better ideal of how we *should* work, I will in this book present a developer-driven approach to testing, focusing on automation and quality instead of tedious and expensive manual labor.

This book discusses some of the core concepts around testing and what “testability” means, and also practically demonstrates a number of testing types and patterns that should be helpful in testing, especially if you are building distributed microservice architectures.

Overall, this will be a friendly, highly opinionated, and impassioned rollercoaster ride through a subject that’s commonly misunderstood and underappreciated. At least that’s what I’ve seen! Throughout, I will liberally point to other materials, as my aim is to shape an attitude and approach with hands-on examples than to be a compendium tome of all there is to know on testing. I’m not that smart, and no one has that kind of patience anyway. There’s too much great material written that I can’t compete with all of it—but I hope this will be the content that drives hordes of developers to *actually become interested in testing*.

What will you learn?

This book will get you in the trenches of writing testable code, understand modern test practices and how to pursue them, as well as demonstrate several testing methods applied to common microservice use cases with public front-end applications and serverless back-ends. So the next time the trap door is sprung underneath your feet—“caught you now, developer-who-hates-testing!”—you will have at least a grounded understanding of what to do and that testing doesn’t have to be the pain some make it out to be.

The testing types we will discuss and I will demonstrate to you are:

- Static code analysis (linting etc.)
- Unit testing
- Smoke testing
- API (integration) testing
- Synthetic testing
- Load testing
- System testing
- UI end-to-end (“E2E”) testing
- Contract testing

...and of course, we’ll take a spin doing continuous testing during your CI stage.

Information

Throughout the book, I will often use the term **API (Application Programming Interface)**. You should understand the term in a broad sense, not just as a term for *web APIs*, but rather for any system that has a surface that can be accessed by someone or something else.

Why write this book?

The weightiest reason for writing this book—or compiling it, rather, as it also contains some of my previous work—is that **I find testing to be woefully lacking in the professional contexts I’ve been in** for the duration of my career. In fact, testing is something I myself got enthused about simply *because* of the price paid in abject failure—sometimes by myself, sometimes by my teams, and sometimes by my employers lacking good ways to conduct testing in the first place.

As such, I wouldn't be surprised if you have been in similar situations or situations where the business side overcompensates for poor quality with an insistence on manual testing efforts. It's a classic example of good intentions causing more pain.

The second reason is to clarify how testable code is good code—and the same goes the other way around. I will spend quite some time considering what good code, and tests, look like. Additionally, to test something, you should *know* what you are testing. Therefore, we will also touch on a few techniques and conventions to add relevant documentation of various kinds to our code base.

Audience

I am writing this for several intended audiences:

- **The frustrated but curious software developer:** You are out there, somewhere, building your services, and maybe you feel frustrated about things breaking for no good reason. Or you have a manual testing flow that you *just know* could be removed. Or you just want to ace this software development thing, and testing happens to be your weakest skill.
- **Colleagues:** Because we talk a lot about these things and since nothing beats actually showing what we mean.
- **Myself:** As a way to learn more and hone my didactic, communicative, and technical skills.

The project source material

I've tried to adapt this to something that hopefully is enlightening to more senior developers, but accessible enough for a tester who may have had less programming time. You should be able to tag along by following the instructions given throughout the book. You don't need to understand the details of the provided code.

The majority of examples and scenarios will use code bases from multiple applications I've written in the recent past:

- [get-a-room-ddd-example](#)
- [testable-systems-starter](#)
- [microservices-testing-workshop](#)
- [acmecorp-microservices-webshop](#)
- [better-apis-workshop](#)

Information

Notes before starting Since my main language of choice is TypeScript and modern Javascript/Node, there will be a slant towards that in tooling and any code examples, though I'm pretty sure most of the concepts should suit your language just as well.

Generally speaking, these applications follow modern cloud-native conventions, using serverless services like AWS Lambda, API Gateway, DynamoDB, and EventBridge. The applications will be detailed a bit more later in the book.

How the book is structured

The very first sections are meta-materials around the book, such as an extended introduction to the subject; a declaration of the book being open source; that I'd love sponsorship if you enjoy this or my other books and materials; and where to find the project resources. In `Getting started` you'll be able to read a bit more in-depth on how to set up and understand the related projects.

An absolute prerequisite of doing testing at all is building testable systems. This is what the section `Building testable systems` addresses. I've already written quite a bit on that topic in (for example) [Better APIs](#) so we'll try to address the most test-adjacent concerns this time around.

The real deal, in terms of testing, begins with `Towards modern testing practices` — this is where we'll start getting to the point about practical testing and the questions you

may have wondered but didn't dare ask. This section addresses the critical theoretical knowledge you need (as far as I am concerned, at least!).

The following, similarly large section is named `Running tests in practice` and covers just what you'd expect. I will cover the majority of "reasonable" test types with commentary, comparisons, recommendations on when to use the type of test or when to avoid it, and also show-and-tell examples using the respective methods. Each page in that section should give you enough to proceed with your own testing, and certainly so when armed with the conceptual knowledge of the previous sections.

Continuing to `Example testing scenarios`, in which I will guide you through how I have attacked three different scenarios in a modern cloud landscape, from front-end-driven testing to complex event-driven microservice scenarios.

Finally, I will wrap things up in the chapter `In summary` and condense the most crucial points I hope I've made.

Finally, for the nerds and deep divers out there, I've prepared the `References and ↩ resources` to contain enough material to keep you reading 'til your eyeballs drop out of your skull.

What is a test?

Man. This question! It can only lead to something bad.

Warning

It should be clear that my goal is not to be authoritative, final, or encyclopedic in this book or in my definition of tests. However, so we have some common ground, I will lay down what I mean with them.

My main experience with building software is in the web development area.

Information

I have some experience with regulated industries as well and feel confident from my work in those areas that their *expected approaches* are several years behind the curve. This helps neither the respective standards, the industries that are regulated, the products to be built, or ultimately, the customer or end-user. Of course, this also only makes things worse for the software engineer.

The following is the basic definition I would propose, being incomplete and all:

A test verifies the actual performance of software against its expected functionality. There exist different **testing types** that are each in their own ways capable of doing such verification. Such verifications apply a variety of levels of *knowledge of the software* (code level, user level...) and expected *granularities* (unit level, API level...) of the response to better determine whether or not the operation completed as expected.

Information

There are certainly more terms and concepts, such as test oracle, test harness, test suite, and many more. However, they are not *useful* in common parlance.

The above definition *should* adequately resolve most pragmatic concerns on the subject.

Also let's be clear that **tests are not technically hard nor should a good test be**.

Let's look at a unit test to clarify this point.

A basic unit test

Our “**system under test**” (SUT) will be a function that adds two numbers and returns the result.

```
function add(firstNumber, secondNumber) {  
  return firstNumber + secondNumber;  
}
```

And our test—for brevity written without any testing framework—is as simple as:

```
const expected = 5;  
const result = add(2, 3);  
  
if (result === expected) console.log('Test passed!');  
else console.error('Test failed!');
```

If you'd run this, you'll see that this will pass when given the numbers above.

But is this not over-trivializing testing? Yes and no.

- **No**, because, well, this is software development and things will undoubtedly grow to be more complex than the above. You'll also have to learn nitty-gritty details like how to deal with unreachable code lines, what to do about asynchronous failures, and such. Nothing that's impossible, but as always: The devil is in the details.
- **Yes**, because well-structured code (and code paths) and the **right type of test** using the **right boundary** will make testing marginally more difficult than the above example. A good test *is not* a complex thing! More on that later.

Is there a need for dedicated QA?

No, I don't believe there is in most cases, as long as you can work with relatively deterministic circumstances. I'd be less negative to dedicated QA in places where there is an extreme degree of emergent behavior and non-determinism, such as in the games industry and probably some very complicated parts of digital-physical products and manufacturing. Given that the majority of development does not happen in such places, I'd still propose "no" as the most sensible binary answer for most people.

As should be clear from the title of this book, the primary audience for this book are software engineers. In the spirit of Agile and actually well-working development strategies, **the quality assurance or testing role should be minimized and be an integrated concern in the day-to-day development and operations.** Experts are certainly always welcome, and definitely so when we are dealing with really tricky situations or when a team needs qualified coaching on conducting better testing. We also need to separate the roles of *subject matter experts* to support us in better understanding the domain and how we implement it truthfully in the code. Given the choice of better subject matter experts that help the team write better, crisper, more testable code or simply having a tester whack out tests on a sidetrack to the regular development, then the better, crisper code (leading to easier testing, leading to testing being done at all, asking the dev team to write them) is always a better option.

Checking vs testing?

This is a [red herring](#) subject that seems to come and go.

The "[checking vs testing](#)" issue seems to have revolved around what we call *testing* (more exactly, *automated testing*) actually being **semantically closer to checking than testing.** *Checking* is said to be a rote task that a machine can do because we look at the known parts of a system. *Testing* is said to be something more complex and creative that only a dedicated expert (human) can do, and that testing fundamentally is about uncovering *unknown* information about the system.

Information

The semantics should be similar enough to not really warrant this dialog, and I'm not really seeing a scenario in which we will be discussing *automated checking* any time soon.

The real issue isn't about differentiating between non-creative labor and creative labor. The problem is that we also start creating a box into which we can put select people and roles. Hence we are potentially creating an ecosystem that is constrained to the testing community and not to the people who built the systems. There are some arguments why you might want someone else to inspect or fiddle with something you built, but we already have decades of talk about pair programming, pull requests, and mob programming to deal with that. Unless you are already running such practices it simply seems more reasonable to evolve towards ideals that have been proven to work, instead of setting up a testing gulag on the perimeters of your development team.

If you have something to win by pushing on the “human” card, then some will do it.

Information

It seems to me the more useful and concrete outcrop of this led to the creation of the *exploratory testing* approach which we will address later. That *actually* has some (limited) value.

Continuously testing your software

When we further extend the approach of having testing as part of our *definition of done*, then we've already started going down the path of *continuous testing*.

Many development teams now use a methodology known as continuous testing. It is part of a DevOps approach – where development and operations collaborate over the entire product life cycle. The aim is to accelerate software delivery while balancing cost, quality, and risk. With this testing technique, teams don't need to wait for the software to be built before testing starts.

They can run tests much earlier in the cycle to discover defects sooner when they are easier to fix.

— IBM

This marks the natural progression of ensuring that the testing (partially or all, if possible) is done throughout the IDE development experience, through to pre-commit hooks, and then the CI environment itself running the full set of tests. Doing so creates a practically completely transparent view of the expectations put on your code and what it takes for it to go to production.

Given that tests are meaningful, truthful, and fast then we can start taking steps to even revamp how we do any manual verification whatsoever. And very soon, you might not need that, nor the complex array of environments, given that you've written more, better, faster, more truthful tests and run them all the time and continuously integrate your codebase several times per day.

That will be a true game changer for most people who might today be used to weird, breaking tests, many complex environments, manual verification, and other travesties of the trade. The best thing is: All of this is completely attainable.

Information

Some recommended reading:

- [Wikipedia: Software testing](#)
- [Wikipedia: Software testing controversies](#)
- [Wikipedia: ISO/IEC 29119](#)
- [DevOps tech: Continuous testing](#)
- [IBM: What is software testing?](#)

The testing-industrial complex and how it denies competent practices

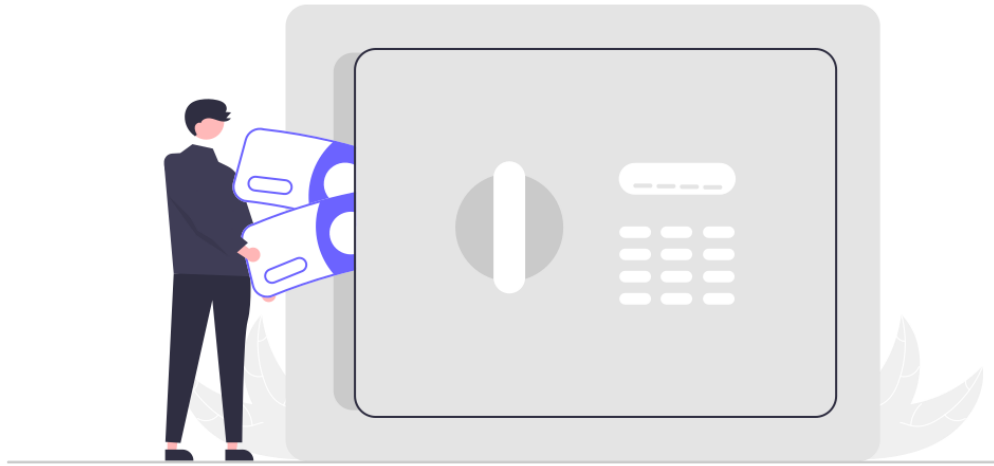


Figure 1.1: Agile consultants stowing money, having helped your company only superficially change. You got 10-minute stand-ups and QA consultants but got no continuous delivery or test automation

As stated in the introduction, my own experiences, and seemingly those of many others, seem to indicate that testing is often not carried out as an integrated development activity in any meaningful capacity. Sometimes it doesn't even happen at all! More often, though, we may have some degree of manual testing and verification combined with a degree of test automation, and teams feel caught in a swamp: How do we evolve from this situation?

Knee-deep in it

My experiences stem from running my own businesses, creating and maintaining open source projects, working at a consultancy, an agency/studio, and working directly for the digital department of a manufacturing company. The range of clients I've had is broad and covers many sectors, maturity grades, and sizes. I am only one person, but again, when I talk with others and listen to their experiences they echo similar sentiments to what I've seen myself.

If I were to somehow express the situation that many of us have to deal with these would be some of the details in such a picture:

- There is often no expressed, unambiguous *definition of done* (DOD).
- The DOD is not necessarily aligned with the business stakeholders, i.e. misalignment between the development and “making money” sides.
- The software development life cycle (SDLC) is either regulated in minute detail (typical for big enterprises) or non-existent (typical for agency contexts and low-tech-maturity orgs).
- Quality, a concept that should easily also encompass testing, is *said* to be important but lacking clear conceptions of “what does good look like” (i.e. an expressed DOD with non-functional criteria) there is no accountability on either side of development or business or elsewhere.
- Most organizations are not adapting wholeheartedly to productive, modern software delivery models.
- The inspection model even Deming found deplorable is still in strong favor.

And for the poor individual software engineer, we find that:

- The job market for software engineers and developers is in very high demand but is chronically undermanned; even senior positions are filled by juniors or potentially completely miscalculated hires at a rate that is actively damaging companies and their output.
- For better or worse, you can be entirely self-taught or have completed a post-doctoral thesis on compilers but still have similar job-related skills or expectations.
- There is no single or universal career path in software engineering. Adding to the previous point, consequently, it’s easy to get stuck not using all of your potential.
- Testing and possibly most other quality-enhancing areas are probably one of the lesser discussed parts of the science and craft.
- Combining poor skills with unmoving orgs, we end up retaining much of the manual toil despite decades of attempts to remove it.

My theory is that given the incredibly fluid nature of skills and standards in software engineering, combined with the majority of organizations (that are paying for software engineers) not *being software organizations*, has resulted in an untenable situation where it's become a guessing game around whether or not one is "up to standards" and even what such standards mean. Because process is often emphasized over practice in typical orgs, it's more important what the diagrams say than how we actually produce good, tested software in practice. Modern engineering practices require (among other things):

- Frequent cross-functional investment from other stakeholders in an organization,
- Relative autonomy in the performance of the engineering work itself,
- The capability for the org to adapt to incremental continuous delivery,
- A very high degree of automation to minimize toil and lead times,
- Consequently a forced adaptation of human tasks to machine tasks, most importantly perhaps in the testing area.

The historical precedent and reason for this is, of course, Agile and DevOps, which accelerated how we can package an approach to do those things. Let's discuss Agile a bit – the more controversial of these two.

Hijacking Agile

At this point in history (I'm writing this in 2024), on paper for many, the *only* model of software engineering has been Agile. At least that's what we might say.

Success

The classic book on hardcore Agile is Kent Beck and Cynthia Andres's [Extreme Programming Explained: Embrace Change](#). I can see this having been "*The Satanic Verses*" of its time in corporate environments. It still carries a punch. For a refreshing, more recent read on Agile, check out Robert Martin's [Clean Agile: Back to Basics](#). As opposed to many opinionators in Agile, Martin *was there* when the initial Agile principles were written. You'll find his book equally amusing, cleansing, and frustrating that we are where (many of us) are.

But looking at the recent past it becomes clear that **Agile became adapted, rather than adopted** by many companies. And with that move came Scrum and its Scrum Masters with many a pact signed with the project management devils; SAFe that watered down Agile into a model that is so hard to grok that you need to certify even to talk about it; and the rise of cargo cult Agile, where (for some reason) the rituals like standup meetings are seemingly more important than the [technical practices](#), including continuous integration and test-driven development.

Suffice it to say that Agile never really panned out to what we would typically mean by the concept.

At least two industries weathered blood and money during this adaptation and went to work: Agile coaches and QA. How better to support an organization that “wants to do Agile” rather than “be Agile”? Millions of hours were billed, and are still being billed, as an effect of that. Don’t get me wrong, there are competing and newer ideas that are not “Agile” — but they are “agile”. The problem is that foundational philosophical concerns of this (and related) movements were never embraced truthfully by many organizations, only parts of the management-leaning practices and relatively inconsequential rituals. So while orgs may be crying to the bank about their agile coaches and testers, they prop up the defense against something worse: A critique of the entire way they deliver software in the first place.

Information

See my article [Software Delivery That Makes Sense](#) for more on this.

If your true goal is to get to continuous delivery/deployment and otherwise use the Agile technical practices then it becomes clear that there is no clear place for the traditional QA/tester role, as they will undoubtedly add latency and friction to the mix, rather than help resolve it. Even handing over code to be tested by someone else will be less effective as soon as an engineer has sufficient skills to use the tooling to write and perform the test. Unless your explicit goal is to either not test at all, or you have no use for testing skills, then you’ll gain momentum already in the very short term by writing the tests yourself. Plus, you get the significant benefit of using tests as a development tool and letting the tests expose how well-constructed your code actually is.

I don’t hate testers. They should just turn into becoming full-fledged engineers as that’s

the role they are already role-playing.

Agile testing is therefore, conceptually, something we should see as testing *within the Agile context* rather than “this is a poor sod trying to write tests for someone else; it happens to be a pseudo-Agile team”. Getting to being agile/Agile is not about micro-managing or setting up specializations, it’s about reducing waste; handoffs are waste if you can do the work in the first place.

If on the other hand, your heartbeat is driven by billable hours to keep organizations not moving forward, alas, efficiency is not really part of that calculation. But then you are part of the problem, not the solution.

Information

For more on Agile testing, see [Test Driven Development: By Example](#), [Agile Testing: A Practical Guide for Testers and Agile Teams](#), and [Growing Object-Oriented Software, Guided by Tests](#).

Why test?

It is not enough for code to work.

— Robert C. Martin

Go for “built-in quality” instead of raising money to buy an off-shore QA department.

My background (whatever that means these days...) is not in testing. But as a software developer and architect, of course, the concern of testing and quantifying quality has been something that I’ve had to deal with for years. But what is quality? What do tests have to do with it?

[Software quality](#) as an overarching concept is incredibly hard for most people to grasp. It tends to be an “I know it when I see it” type of thing. Fact: **Quality is in fact highly regulated in many industries**, basing itself on the bars set by standardization bodies such as the ISO. In most heavy industries, it would be unacceptable or perhaps even illegal to start meddling with the quality of the product.

Information

Without being right or wrong, we can use ISO’s definition of software quality for now: “[The] **capability of a software product to conform to requirements**”. See the concept [system quality attributes](#) for a related, extended scope of such requirements.

This problem, though, seems to grow even harder in software engineering, as it can feel obtuse and opaque for non-programmers, requiring specialized labor and the work product itself is immaterial and hard to judge. One part of our history should be clear, however, and that is that **we do have concepts and champions that clearly expound quality as an integral part of competent and serious software engineering**, examples being:

- **Robert C. Martin’s** [many well-known books](#), such as Clean Code, emphasize the craftsmanship, ethics, and professionalism of the trade.
- Modern development approaches such as [Agile](#), which take quality seriously.

- **ISO standards** such as [ISO 25010](#), which attempt to codify the criteria for quality.

When we build software, in other words, conduct **software delivery**, it is usually at the request of someone. Most teams will have a “[definition of done](#)” that includes some set of criteria determining whether or not a piece of work is ready to ship or not. This definition usually includes [functional](#) and [non-functional/cross-functional](#) requirements.

The **functional requirements** deal with the need itself: Does it do what we want it to do? The **non-functional requirements**—more and more we hear these called quality attributes because that sounds less “optional”—are instead about the *quality* of that execution: Does it do it at the expected speed? Is it secure? Is it reliable? Is it maintainable? And so on.

Without some set of these questions answered in competent and well-constructed ways, there is simply no guarantee that we can *trust* the provided solution. Here the spirit of Robert Martin hovers around us: “**It is not enough for code to work**”. It may be immoral, illegal, impossible to maintain in a team, work poorly, or exhaust system resources at an unsustainable pace. If that’s the case, someone would be hard-pressed to accept that the *code working* was good enough.

In a world where more and more of our reality is transmogrified into code, it becomes imperative that those in charge of leading software delivery do so without sacrificing quality.

The greater question is of course: **How many, and which, quality attributes will you need to support?** We could for example use the [Consortium for Information & Software Quality](#)’s (CISQ) [primary selected characteristics](#) to guide our work:

- Reliability
- Efficiency
- Security
- Maintainability

[All of these can be assessed](#) in relatively well-known and automated ways with a plethora of tools. For our concerns in this book, we will broadly keep these in mind for the rest of the material.

Information

Testing as presented here won't affect all of those attributes in any big ways. My number one argument will be steadfast: We need to write deterministic, well-structured, good code with the *minimum amount of test scaffolding* and with *the smallest number of tests to reach maximum confidence* (and most likely, near-100% test coverage). Our tests and test tooling should help us understand our reliability, efficiency, security, and maintainability.

Why automate testing?

A venerable topic which obviously many articles and books have been written about by people far greater than myself.

Let me briefly give you my personal highlights:

Tests are fundamentally about ensuring stability

With modern development practices like CI/CD, trunk-based development, and putting complex distributed systems in production multiple times per day, one of the great risks becomes whether one can *guarantee* to some extent that **what is released is stable**. Without mincing words: If you have no test automation, you're dead in the water.

You can certainly deploy and release fast *without tests*—I can attest to this practice since I did it for years before “growing up” and taking responsibility—but at some point, you'll probably be faced with clients or users calling in about issues, rather than *you* observing issues first.

The easy answer is: Test your things. You might still get novel new problems, but at least that dumb small thing you checked in for your testing is not out in the wild causing problems for everyone. Deterministic systems are fun to build, easy to test and are what we need to strive for. These conditions provide a very good foundation for stable, well-behaving software.

We test because that's the way we set some level of guarantee on our delivery

How do you assess the completion of the work, and that it has the expected level of quality? Automated testing ensures that you can quickly and objectively verify these things. While it's not unheard of that you'd need manual acceptance tests, that is by extension more of a feature-oriented verification (which we can test and deploy at the time of completion and hide behind a [feature toggle](#)), and in some cases, it's no more than a psychological crutch.

Taken to the extreme, this is where something like [acceptance-test-driven development](#) would shine. And no, you wouldn't do manual such tests!

Automation may replace people, but it will definitely replace the drudgery

Back to the frequent releases: Even in some modern organizations today, I am seeing not small, but *significant*, arguments being made against CI/CD and related concepts. The crutch of manual verification is strong with many.

One of the key wins with automation is that the drudgery and churn around manual verification can disappear, leaving only—in the best case—business validation; which is frankly more about diplomacy and process rather than a technical concern. **No one wants war rooms every 6 weeks just because you will launch something.**

No tests = no modern dev practices

I was recently surprised by how many of the modern development practices I thought were second nature actually reside on **testing as a foundational activity**. So, if my assessment is correct, that testing understood or conducted as often as it should, it's fascinating how people want to pick the good nice bits ("CI/CD sounds awesome") and leave out the harder bits ("yeah my code don't need no testing").

Frankly this invalidates any claims on modern practices. And when I say modern, I mean things that have been almost taken for granted for 20 years or so.

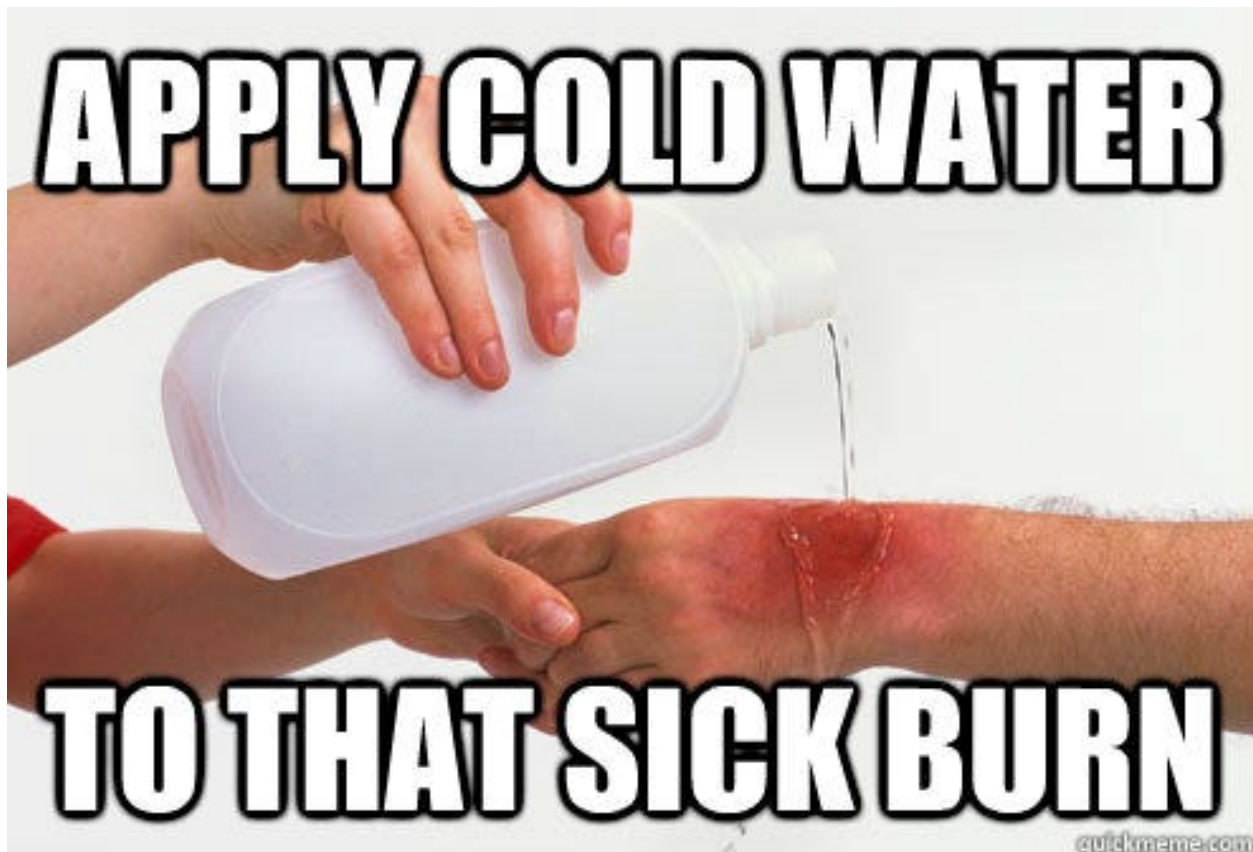


Figure 1.2: Yeah. Sick burn. I felt it too, so it's nothing personal.

You will only get the results you test for

Then, of course, let's not be naive... The fallacy of testing is that you might be tempted to think that full test coverage is a promise of perfect functionality. That *could be true*, but it's not universally true. Good testing will often require several types of testing. And even then, you are still only in the space of known issues that you are testing for!

Tests are brilliant at verifying the “known knowns” of the problem space. You will need other tools, like observability, to catch the “unknown unknowns”.

In closing

Tests are a core part of the development practices most of us have worked with, or at least read about, for the last decades. Without it, we won't get most of the benefits

of CI/CD, trunk-based development, and so on. As for application development in the cloud, and the aggressive timetables of many companies—who are suddenly on a global playing field—we’ve seen the DevOps movement really push far on how we think about infrastructure in a lifecycle similar to that of code. All of this is good, as we upskill millions of software engineers into a new reality. But there is a dark side to all that new shiny stuff...

“Playing DevOps” in a hyperscaler while building modern cloud-native apps, without quality practices such as testing, is like riding a rocket ship going straight to Hell: No other option will continuously deliver you into the shit quite as fast. You need to have some way of controlling things—*test automation is that thing*. ITIL and manual processes are dead and won’t be making an appearance in the cloud-native world. If we don’t want back, we will have to take on the (actually not that heavy) burden of testing our things.

Can something be untestable?

It bears mentioning that nothing is untestable, but very bad code will be very good at feeling untestable.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

— Martin Fowler

Why is it that sometimes we build something, but it tends to be hard to actually prove that it works, other than when we use the thing “for real”?

Software engineering is, like Uncle Bob writes, neither regulated nor a profession in the strict sense of the word. Any software engineer, they’ll have found their own “truths” and hard learnings. The **work becomes more akin to a craft than to a well-informed profession** which you require a sanctioned license to perform.

Unfortunately, that also means that there is—compared to other industries—limited common understanding, common frameworks, common standards, etc. I can hear you from the other side of the screen, “hell no, there are countless things like SOLID, TOGAF, IEEE, WCAG, W3C...!”. And you are right. Yet, any time you go into a new project with new teammates you’ll have to decide on *so many* distracting things: Which flavor of Agile? Not Agile? Linting standards? Which frameworks? What structure? What language? MVC, 3-tier, n-tier, microservices? Cloud-native components or Docker image in a PaaS filled to the brim with open source? What kinds of testing? Nah, just kidding. No one will mention testing.

And that last one thing is the one single, simple reason that forms the core of why this small book came about at all: I see so many developers either not testing at all; thinking it’s useless; opining that testers should do the work; reptile reflexes arguing against learning something new; testing being seen as selective *nice-to-have*; being satisfied with having *any* testing at all...

The most foundational aspect that we should adhere to as professionals, and for the remainder of this book, is that **we have to write clean, testable code. You should also write tests in direct temporal relation to writing the code. Not necessarily TDD, but in the same rough window of time. Don’t leave your code untested.**

Testing is, at least in my thinking, the crowning jewel to prove that you have actually delivered clean, deterministic, and predictable code. **The enemy of good, testable code is unpredictability (indeterminacy).** Tests just validate that your code is not in such a sorry state. It gives you a way to add safeguards to something that does not come with safeguards out of the box.

The potential role of TDD

Test Driven Development (TDD) is one way, but not the only way, to reach testable code. One of the key reasons that TDD is effective is because the tight loop between coding and testing ensures that there will be no good way for you to *not* have both the code and test when you move on to other things. Writing code after the fact is something that a hardcore TDDer would scoff at, for good reason: There is no guarantee that your ethics, team/manager, or backlog will allow for the test to be written as a separate component when you have working code. I think that this is a true and clear-sighted observation of how this works out, in reality.

However, there is nothing that is “absolute” about this stance. I tend to write tests when there is a rough sketch up and (maybe!) running. I use the test as a way to actually, well, test and run the code, rather than doing this manually by running and debugging the code. Using a test framework makes it extremely convenient to actually verify the functionality and I would never go back to a more primitive state of this.

Good code is the way to testability

You will want to write code that behaves in a deterministic fashion, meaning you can always trust it to be in an expected manner given the same input.

Well-structured *clean* code will be easy to test. That’s just the truth of the matter.

Information

There are many good resources for software engineering. Some common recommendations that I have read and feel happy to also recommend include:

- Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship (2007)
- Robert C. Martin: The Clean Coder: A Code of Conduct for Professional Programmers (2011)
- Robert C. Martin: Clean Craftsmanship: Disciplines, Standards, and Ethics (2021)
- Robert C. Martin: Clean Agile: Back to Basics (2019)
- Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm: Design Patterns: Elements of Reusable Object-Oriented Software (1994)
- Martin Fowler, Kent Beck, Don Roberts: Refactoring: Improving the Design of Existing Code (1999)
- Language-specific relevant literature, such as [Effective TypeScript](#)
- And of course: Understand core software engineering concepts like [SOLID](#), know how to work with [object-oriented programming](#) and have a grasp of [modern software and cloud architectures](#).

Anyone who knows me knows that I am heavily indebted to [Domain Driven Design](#) as well. You should absolutely look into books on that topic for more on the level of patterns, strategies, module thinking, etc.

Manual testing is useful, but not often

Let's not kid anyone: **You will have a better product if you automate the boring and repetitive parts and definitely those parts that happen to involve boring and repetitive testing.** However, in reality, you will sometimes find yourself needing to do certain manual verification or “prodding-and-poking”. Should you now look deep into the glass and become a brooding existentialist... a *manual tester*, is that what your parents wished for you?

No, you don't have to brood on this question. There **are some valid reasons** to *some-times* do these things manually. I'll try to address some parts which are totally fine to do manually without being contradictive to our end goal of automating tests.

The first time you do something

In general, **automating anything is something you do for the long term.** Therefore it's not uncommon that when experimenting, designing, and trialing ideas, then you do this manually. At this stage, the code is not ready to be tested because that's not where you are in the lifecycle of whatever you are building.

Collecting initial test data

Inevitably you will need to have *some* amount of data available. If you are building a new service and expect it to integrate with some third-party system then you'll progress roughly through a sequence such as:

- Design a solution, or make a hypothesis
- Find out *which* APIs to integrate with
- Ensure they are documented
- Find out what the expected inputs and outputs look like (i.e. data)
- Verify “toy” functionality, using the API for real (see the next point)
- Collect the test data (inputs, outputs)

Storing the data in your code base means you now have superpowers, making it possible to:

- Write the integration
- Mock the integration so you don't have to use the real deal during tests etc.
- Write the tests of the integration

Yes, it's a chore to get that data but that's nothing to have feelings about, given you win a lot for that tiny investment of manual effort.

Basic checks and verifications

During development, it's convenient and entirely realistic to have an Insomnia [design document](#) or Postman [request collection](#) with your endpoints and typical calls, ready to use at the click of a finger. So, there, I wrote it: Manual checks and verification are OK to some extent when it carries a meaningful impact.

Warning

Handle this like a *troubleshooting tool* rather than as anything else. Whenever the case is something that you systematically have to do, it's time to look into automating whatever it is you are doing.

Exploratory testing

Exploratory testing can be an interesting approach, but should not be done in an *unqualified way* or on *unqualified work*. If a system is strictly deterministic and you've written at least basic safeguards and checks, as well as using a module structure in which there are clear flows that cannot be misinterpreted or abused, then the need for exploratory testing rapidly diminishes.

Exploratory testing, at least for me, seems to be most useful when we truly know nothing (or very little) about the thing we are testing. For most engineers, this should never be the case; rather think of this as sending in a minesweeper to an uncharted war zone to understand what nasty surprises we are facing.

Exploratory testing might also be a ruse

Danger

I am opining this in a totally self-aware snarky tone.

Your number 1 goal is to write deterministic code. Without it, anything could happen! There is no meaningful test that can be written for such situations.

[Exploratory testing](#) is a well-meaning test strategy that has started to gain traction.

Exploratory testing is an approach to [software testing](#) that is often described as simultaneous learning, test design, and execution. It focuses on discovery and relies on the guidance of the individual tester to uncover defects that are not easily covered in the scope of other tests.

— [Atlassian](#)

It aims to leverage *qualified manual labor* to “explore” what a system can do, which to me at least is about trying to find holes in it. To me, it sounds like a strategy not dissimilar from the work done by a penetration tester whose aim is to uncover vulnerabilities in a system. The exploratory tester, however, will have a broader mindset.

Information

One of the more influential names in this area is [Maaret Pyhäjärvi](#). Check her videos, texts, or social media if you are curious.

Don’t get me wrong now, but part of your unspoken job as a software engineer is to *not* have testers around. Just don’t tell them that! Following agile principles and practices,

there would literally exist no testers. See for example [Kent Beck's old rules of Extreme Programming](#) (which is a sister/brother to most of the other Agile you may know):

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

I'm sure you'll already be well aware that old grumpy [Uncle Bob will also have written several books on this](#). Even a corporate moderate like Dan Radigan at Atlassian goes as far as stating that [“let's be clear: scripted manual testing is technical debt.”](#) **There is no manual option. It's that easy.**

So if your number 1 goal is to write deterministic systems, then being “explorative” may very well be meaningful, either in:

- Very ill-performing systems.
- Unknown or undocumented systems.
- Very large (probably undocumented) systems.

To the extent that we need manual testing, I feel entirely confident in writing that such a role should come with abilities include coaching, enabling, being transformative, and being only transitive (disappearing after some short time)—i.e. more along the lines of quality engineer, quality coach, or similar notions. Atlassian also comments on this limited scope:

An exploratory testing session should not exceed two hours and should have a clear scope to help testers focus on a specific area of the software. Once all testers have been briefed, various actions should be used to check how the system behaves.

— [Atlassian](#)

In summary, exploratory testing sounds like something that makes a lot of sense when we want “bang for the buck”, being expensive, laborious, and qualitative in its delivery. We can't rely on it for our *continuous* work.

In closing

While this is in no way an authoritative account of all cases where manual testing makes sense, they should summarize some clear directions:

- Manual testing is unavoidable in many cases while you are writing the first bits with integrations.
- We need to uncover unknown parts of unknown, undocumented systems.
- As one part of our development work (“pushing-and-prodding”).

Project resources

Information

There is no dedicated project for this book. We will instead use several existing projects as a base of reference.

As mentioned before, we will look at several applications and their source code in this book. They are all available at the respective links:

- [get-a-room-ddd-example](#)
- [testable-systems-starter](#)
- [microservices-testing-workshop](#)
- [acmecorp-microservices-webshop](#)
- [better-apis-workshop](#)

All of these are complete, detailed projects with the dedicated `testable-systems-starter` ↩ and `microservices-testing-workshop` even being adapted fully for a practical setting. I will describe commonalities among them in the next section.

Getting started

You can deepen your experience by cloning, running, and learning from several reference projects (mentioned in `Project resources`).

Information

It's of course completely optional to actually run any of the projects. I will be using lots of examples from these projects, but nothing beats learning from hands-on experience. Regardless I think you'll gain something even if you decide to not run the projects.

Depending on the specific project you will have some variability in the exact details. Common factors among all of the reference projects are that:

- They are written in TypeScript
- They are using AWS' cloud infrastructure components, such as Lambda, EventBridge, DynamoDB, and more
- They are thoroughly tested using several types of tests
- The applications are composed using a microservices pattern

While they are all relatively small, most of them are not trivial or “dumbed-down”. The displayed practices and solutions should easily carry across bigger, similarly well-structured codebases.

Pre-requisites

Common prerequisites for the projects include having:

- [Node](#) 18+ installed (or a more recent version)
- [Git](#) installed

- Optional: [Serverless Framework](#) installed (version 3 or later)
- [An Amazon Web Services \(AWS\) account](#)
- [AWS credentials available through your environment](#)
- Some familiarity with programming (Javascript/Typescript/Node) and command-line usage will be very helpful
- A modern IDE like [Visual Studio Code](#) installed
- For calling APIs manually, you might want to use a tool like [Insomnia Designer](#)

The projects should contain all other dependencies locally, so you shouldn't need anything else.

Building testable systems

Before we even try to test something, the system itself has to have some basic qualities so it can be deemed testable at all. I will continue to refer to the system as an API: It doesn't have to be a web API, as the term should cover many common types of systems.

Information

There's a lot more about this general area in my book [Better APIs](#) if you are interested.

This section picks up on those things you can (and should) do as a developer before even spending any time on the testing itself.

Remember [Werner Vogels' advice](#) and think of API as a broad concept:

1. APIs are Forever
2. Never Break Backward Compatibility
3. Work Backwards from Customer Use Cases
4. Create APIs That are Self Describing and Have a Clear, Specific Purpose
5. Create APIs with Explicit and Well-Documented Failure Modes
6. Avoid Leaking Implementation Details at All Costs

Good code has many qualities that are often spoken of when discussing what "good code" actually even means. Commonly mentioned qualities include being readable, maintained, organized, and structured, and following reasonable semantic naming conventions.

For us to build something that is testable *at all*, the code, therefore, needs to be of a certain level. In this section, I'll go into some of what might go into that.

Information

In this area, some relevant literature includes [Principles of Web API Design: Delivering Value with APIs and Microservices](#) and [Building Microservices: Designing Fine-Grained Systems](#).

What our “good” looks like

Insanity is doing the same thing over and over and expecting different results.

— Albert Einstein (supposedly)

Software engineering is such a big business that we today seemingly have more people talking *about* software than actually making anything—that’s unfortunate because it also starts eroding qualitative engineering practices. I’ve seen that in some places we are starting to have relatively young engineers thinking you *can’t* have agile, continuous deployment, trunk-based (not PR-based!) workflows, and 100% test coverage. That’s ironic, as these qualities have been addressed in the literature for more than 20 years! Not to mention they are highly present in “real” software companies, that are actually creating bottom-line results thanks to their software products. Still, in most companies, we see a division between IT and business and we see project managers (rather than true-bred product owners) driving one-time deliveries (rather than iteratively engaging with long-living products) on timeframes unknown to key stakeholders within technology departments, leaving a great big mess of it all.

Experiences like these tell us quite clearly that **bad practices may come from people least knowledgeable in efficient and qualitative software delivery—not the other way around.**

In this chapter, I’ll present some of the ways you can start setting standards for your software delivery and tests, as well as how to think in the team.

Getting Unstuck from the Tyranny of Local Optima

Improvement, both for individuals and teams, is challenging—particularly in software development. Understanding what “improvement” means, and agreeing on it, is the first step. How do we measure performance? Is it through code output, feature delivery, reliability, or something else entirely? Without a shared understanding of “performance” and “quality,” meaningful progress is difficult to achieve.

Information

The area of developer productivity is big and full of mines. I'll try to pass it by gently without losing a leg or two!

When discussing output from a software development team, at least three key axes come into play:

1. **Value of output:** The work provides customer and business value. In other words, the work achieved an ultimate goal of the organization (and its customers). High-velocity output that doesn't put dollars in the bank and/or help customers whispers the story of a soon-to-be-bankrupt company.
2. **Quality of output:** The work is stable, harmonic, safe, and useful. A good feature that barely works is not any good at all.
3. **Quantifiability of output:** That there is actually a stream of work being delivered. Even the most brilliant work which is never delivered is effectively worthless.

All three factors are crucial and interdependent; they are not mutually exclusive. However, in practice, many teams tend to focus disproportionately on the “quantifiability” axis, which is unfortunate because it depends on the other two. Without a clear and engaging goal, team members may feel disconnected, disheartened, or disillusioned if they do not understand how their work contributes to the larger picture.

Quality is influenced by both culture and practices. Focusing on value (and effectively communicating expected outcomes) and emphasizing quality create conditions for faster output. This is often counter-intuitive to traditional industrial logic, but it is essential in software development.

To illustrate, let's consider a hypothetical diagram of some software releases in terms of bugs, issues, team satisfaction, and technical debt.

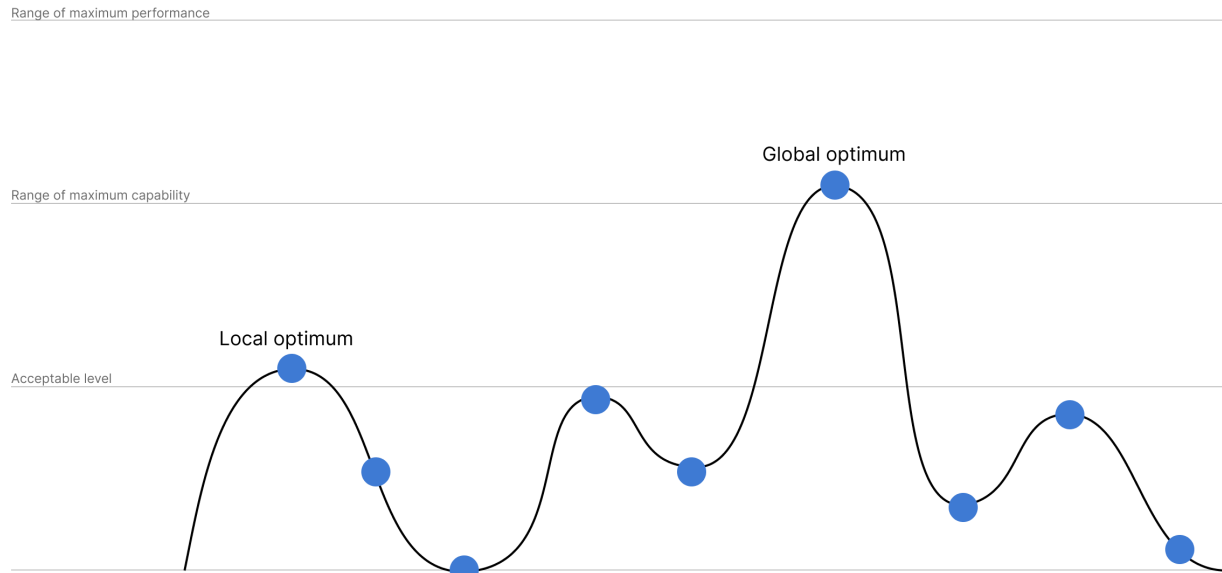


Figure 1.3: Local vs global optima. Adjusted from <https://peachturnspro.wordpress.com/2016/04/19/stuck-in-a-local-optima/>

Although such diagrams are rarely used in software teams, tracking work and output against expected and actual outcomes can provide valuable insights.

Information

The tools to help with this range from conventional options like Jira’s burn-down charts to newer tools such as Cortex, Swarmia, and DX. Just be careful not to get lost in the complexity of these tools!

In our hypothetical diagram, two releases performed above expectations. Ideally, these expectations were clearly defined in a “definition of done” to avoid relying on subjective feelings. However, achieving such clarity is often difficult.

The question of institutionalizing quality—what it means, how it is tracked, and how it is determined—has long been contentious. On one side, there is an emphasis on culture as the driver of excellence; on the other, a focus on reproducible, deterministic factors akin to processes in medicine and manufacturing. Rather than choosing sides, a more viable approach involves experimentation, personal experience, and research, ideally grounded in academic sources.

A good starting point is establishing a shared “definition of done” that includes concrete objectives, such as writing tests for new code or adding documentation where needed. These small, tangible actions can lead to better practices, like automating tasks, committing more frequently, and encouraging collaboration over prolonged code reviews.

Information

For more on this topic, check out resources like [DORA](#), [Accelerate](#), and [Modern Software Engineering](#).

Following these practices can lead to better code, higher reliability, and faster releases. The key is that these improvements result from relatively small behavioral changes. Since change is difficult for individuals and teams, growth is challenging without a clear understanding of what to change, why to change, how to measure it, and how the organization will support the change.

Culture is also incredibly important. But which comes first—practices or culture? Can an organization perform well with excellent practices but a poor culture? Or vice versa? Neither dimension can be ignored. While we can mandate practices and processes, we cannot impose culture. Thus, the safer starting point is to establish foundational practices and let culture evolve around them. A solid set of practices provides the stability necessary for a positive, generative culture.

Returning to the diagram, we also see seven releases below our acceptance level, yet they were released to the public. These releases were likely buggy, delayed, or failed to deliver the desired value.

To improve, we need a foundation for dialogue and learning. This is where [psychological safety](#) becomes crucial. With this precondition, along with a concrete, externalized concept of quality, such as a “definition of done,” and supporting tools and processes, teams can adopt rituals like [retrospectives](#) and methods like [correction of errors](#) to systematically identify, focus on, and improve their work. This approach is far from “blame” and is more about collective improvement, much like how critical infrastructure is managed.

Improvement is an ongoing process. Two concepts that help explain this are **local** and **global optima**. As Wikipedia puts it:

In applied mathematics and computer science, a local optimum of an optimization problem is a solution that is optimal (either maximal or minimal) within a neighboring set of candidate solutions. This contrasts with a global optimum, which is the optimal solution among all possible solutions, not just those in a particular neighborhood of values.

— Wikipedia: [Local optimum](#)

Consider the concept of capability. In our example, **one release exceeded our expectations for our current maximum capability**—what we believe or know our team is capable of producing. This release represents our team’s current **global optimum**. It marks the best we could achieve, given our skills, conditions, and so forth. By surpassing this limit, we effectively “level up,” pushing our boundaries and becoming better.

It’s important to note that “maximum capability” is not about superficial metrics like lines of code or features delivered. It’s about our collective output that aligns with organizational expectations. Many teams struggle to measure or discuss capability due to a lack of explicit dialogue about it. In some cases, capability may be crudely defined by lines of code or the number of features, which is problematic as it overlooks the importance of value and outcomes.

In contrast, the first half of the diagram shows a release that was the best in a specific timeframe but not as good overall. This is like saying a local hockey team had a great game against the second-worst team in the league, even though they had a poor season overall. Locally, they performed well, but in the broader context, their performance was subpar.

The concept of maximum capability is dynamic, changing with team composition, circumstances, technical debt, and external factors.

Danger

Simply working harder or faster will not make you more efficient. Focus instead on improving your “system of work”—the processes around the labor—to make it easier and faster to do the right work. Use what you’ve learned here as a starting point!

Our human tendency is to adapt to our current circumstances as if they are the best possible options. Even if our current methods seem effective, failing to critically evaluate and consider broader contexts can lead to complacency. This often results in the “we’ve always done it this way” mindset.

The “tyranny” of local optima refers to this complacency—believing that one success means we are on the right path without considering that progress can vary—both for better and for worse. It’s important to stay vigilant and question whether we can do better, even after successes. Are we at our best, competitive level? Or are we stuck in a local optimum?

Being stuck in a local optimum is akin to analysis paralysis or other forms of organizational stagnation. For example, we might spend excessive resources on manual testing to barely meet acceptable standards. Fear of change can prevent us from moving beyond this state, even if our true potential is much higher.

What you want is to consistently perform at the top of your game, achieving the maximum capability of your team. To do this, you need to clearly outline the **delta** between your current state and your desired future state.

Information

Not tracking your team’s performance and delivery? Consider tools or methods such as:

- [SPACE metrics](#), a follow-up to DORA metrics with broader categories.
- Tools like [Swarmia](#) or [Cortex](#).
- Conducting activities like retrospectives and surveys to assess team performance.

Success

Create accountability for yourself, your team, and your managers. Every change is an investment. While the payoff may be unclear or delayed, you can still de-

fine what the future state should ideally look like. Ask these questions of your stakeholders:

- If not us, then who?
- If not now, then when?
- If not for this particular payoff, then for what?

In software engineering, our primary constraints are **time** and **competency**. A skilled engineer could automate tasks that others might take days to complete manually. This is not an ethical debate; it's about **automation**. Competent automation can save time, which can then be reinvested elsewhere.

If you lack the necessary skills, you have two basic options:

- Hire someone with the required expertise.
- Upskill your team, assigning work that builds their competencies.

To move beyond local optima, you must aim to continuously exceed your perceived maximum capacity. This requires change.

Information

Remember Einstein's wisdom: "Insanity is doing the same thing over and over and expecting different results."

The flip side is that over time you will more easily be able to exceed the capability if your team is actively *learning* how they can minimize time spent on less-important activities. Everyone who has ever trained or been working out knows that the biggest hurdle is at the start...then suddenly you are running 10Ks easily, just for training!

Build better without more resources

I guess one of the big questions many of us building software in the impure reality of companies that need to build software to compete (but would much rather not build software at all) and do not have engineering-centric leadership is “How on earth will we support *better* deliveries when we barely get by today?”.

I believe the answer must include a strong sense of mission (unity, purpose) as well as a rock-solid core around automation:

- **By adjusting or even totally dismantling manual processes and handovers we create time where none existed before.** You’ll be hard-pressed to find any manager that will suddenly, out of thin air, invent a new time for you. Suck it up and start creating the possibilities.
- By continually feeding a quality-aware culture we **make everyone a stakeholder of the quality.** How do we do this without awareness or intentional changes in how we deliver? You don’t; it will end up being one person burning up when it has to be a team effort. No one is too “small” to aid in delivering towards common quality goals.

Success

Some companies and cultures might be too broken to fix. If that’s truly the case, well *c’est la vie*, as a software engineer you’ll find a new job quickly if you decide to pull the plug. Don’t bleed for your job.

Use machines to do machine work that no human should do. Let it run the tests, lint the code, scan for security issues, package and deploy your artifacts, and whatever else you can think of.

Using a yardstick to measure team performance

While there is practically an entire science around how to measure software engineering performance, with [DORA becoming a household name](#) and the [Accelerate book](#) being a

must-read for any engineering manager, it's safe to say that in the early 2020's the [DORA metrics](#) reign supreme as one of the primary metrics sets we can use.

The four “classic” DORA metrics are:

- **Deployment Frequency**—How often an organization successfully releases to production
- **Lead Time for Changes**—The amount of time it takes a commit to get into production
- **Change Failure Rate**—The percentage of deployments causing a failure in production
- **Time to Restore Service**—How long it takes an organization to recover from a failure in production

— [Google Cloud Blog](#)

Note that in 2023, Dora introduced a fifth metric, reliability, though this is (currently) less talked about. Let's see how this changes over time.

Though perfect metrics do not exist, these particular ones do tend to show that high DORA scores can only realistically be attained by teams who work with modern practices. Elite-performing teams meet the highest DORA marks by combining both speed, quality, and stability:

- Release many times per day.
- The lead time for changes and moving into production is less than a day.
- The time to restore service is less than an hour (low performers take between a week and a month).
- The change failure rate is zero to 15%.

— The New Stack: [Google's Formula for Elite DevOps Performance](#)

The below diagram summarizes very well what goes into transforming into a high-performance organization and thus driving the metrics up. You'll see on the left side that we have more than a dozen technical practices that, in total, lead to continuous delivery, that correlate to positive effects such as less rework and burnout.

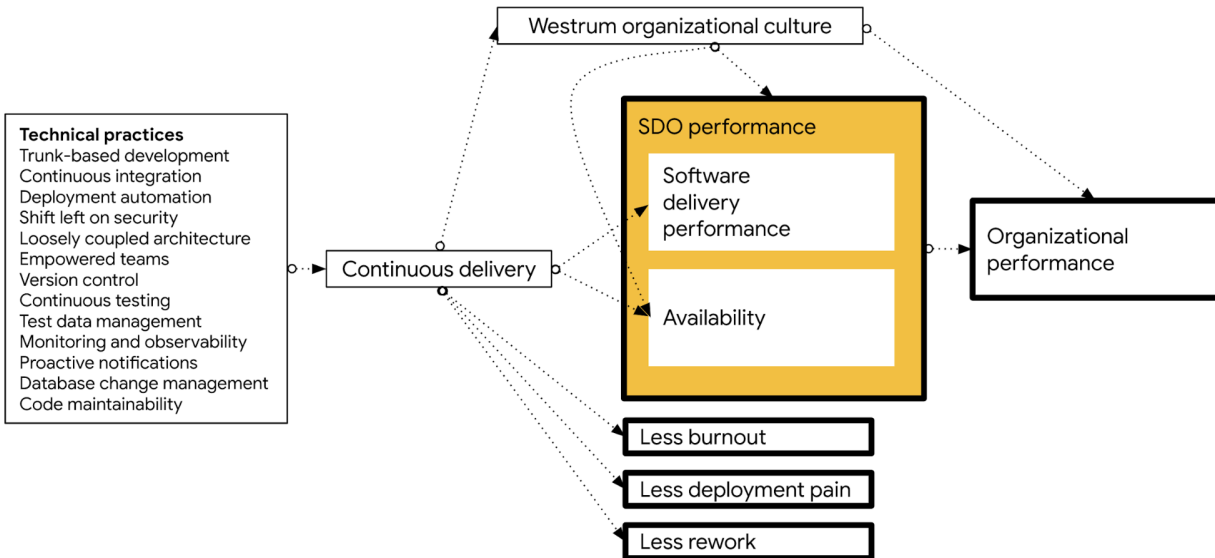


Figure 1.4: Image from <https://cloud.google.com/architecture/devops/devops-tech-continuous-delivery>

Now we can start to plan from the end goal, rather than from the “start” toward some unclear final destination. If we want better organizational performance we see that we should implement, among other things, these technical practices. Starting with updating our practices is a smart thing as it’s actionable, takes collective action, and affects the actual work rather than just the management or “performing” and ritualization of work (e.g. sitting in meetings, doing stand-ups, and similar).

Information

The book you are reading is about testing so I’ll keep it relatively clean here, but we can see that testing is just one of many parts that might need changing to fully transform the circumstances in your organization. For the context here, I will refrain from any additional mention of core practices such as version control, empowered teams, and several of the other practices as they stretch the scope a bit too much. I highly recommend reading [Google’s materials on DevOps](#) to appreciate the many things that go into being a high-performing technology organization.

You can certainly track DORA performance over time, but a quick and painless first start

would be to do the [DORA Quick Check](#) every month to see what the current (experienced) state is. While less scientifically correct than having those calculated based on actual work being done, it should set you in the correct half of the chart at least.

We now have an understanding of practices being correlated to performance, and that we should focus on the things relating to testing. So what would a fuller picture tell us, so we can visualize for our inner eye the end state of our work?

What we want

If we only look at our testing concerns, then our ideal scenario which would go high on the DORA metrics and tick many of Google's technical practices boxes would be something like this:

- We have a pipeline that is fully automated and minimally cognitively complex.
- We can always push code to the pipeline and it will run and complete in less than 10 minutes.
- We are confident in our code and that automated tests catch all potential issues that *we know are not according to expectations*.
- We are “testing in production” and catching (alerting etc.) any issues that are *outside of expected or known boundaries* (manual or AI-assisted thresholds).
- We have built observability into the state of our software, so we can easily understand what went wrong (and for who) if there is ever an incident.
- We have described in our definition of done what is expected of a unit of work to conform according to our standards, and for it to fulfill the above ideal scenario.

That's a pretty concrete vision of how we'd perhaps want our future state to be. By taking the individual steps in writing (formulating the definition of done) you can help make sure that activities are performed and done to a required level that will move you and the team to that ideal scenario with less fuss and a greater sense of purpose and clarity.

Pushing for change

Lastly, let's end with some other concrete things your team can do.

Understand that optimal circumstances will never organically exist; you need to work on creating them. Under which conditions would the team make their best work?

Some ideas might be:

- **Requirements are poor.** We will write and provide a better form that sets directions we need to have considered for our requirements or design proposals.
- **Our definition of done does not include testing.** We'll put testing as a requirement and ensure that all pull requests have tests included.
- **Test automation efforts are going too slowly.** We are going to add a minimum coverage threshold to our CI pipeline starting next month, creating an internal goal and pressure to have this in place.
- **We are writing too many brittle tests.** Therefore we will spend some time inspecting which parts of our codebase might be too coupled and "untestable". We will skip these tests for now and use that time to improve the root causes, by refactoring the code and writing better unit tests.
- **Our manual testers don't provide enough value,** but they are happy to start doing test automation for certain test types so we'll give them time to learn and then write tests starting next week.
- **We have too many meetings affecting most of the team.** We will have meetings only between 8:30am and 10:00am to encourage having dedicated and uninterrupted "flow time" after 10am.

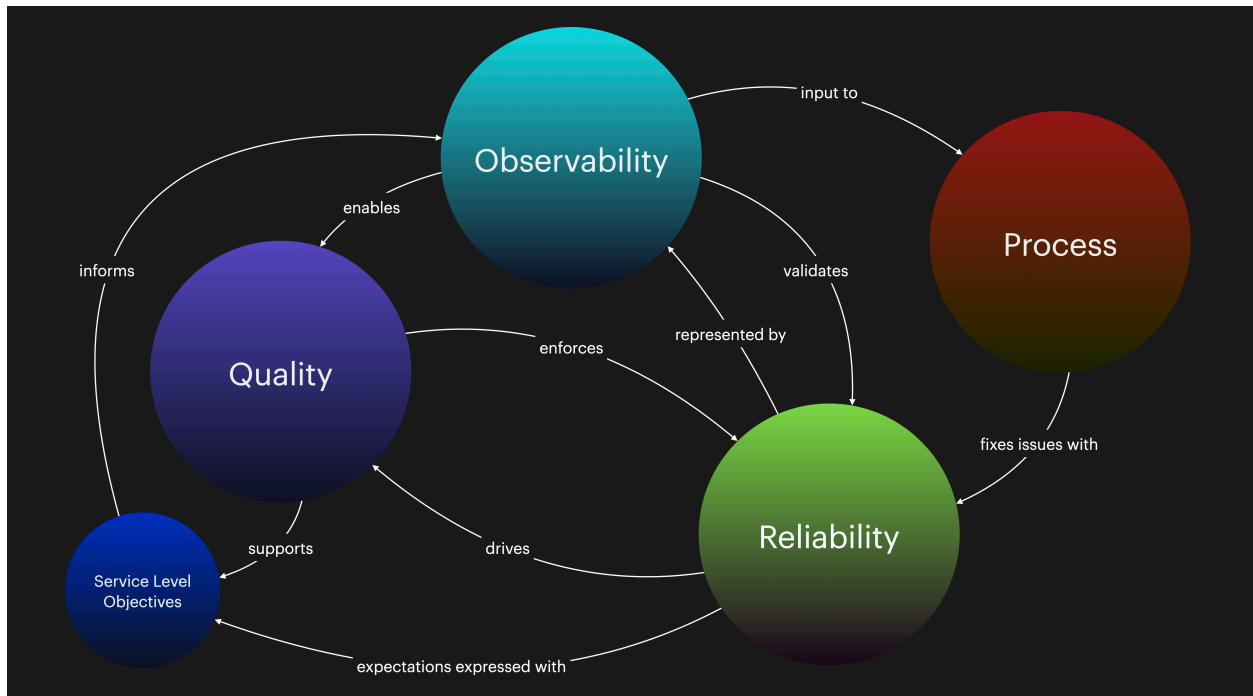


Figure 1.5: To reach qualitative software, it will have to be a whole-team effort, and it won't involve just testing.

What factors are not in your hands? What does need to be addressed to someone else, and if so, who is that?

Discuss and agree on what these things might be. Communicate them to others. Make it so.

Build deterministic systems

Boring, predictable code should be the primary output of your job.

If the software isn't perfect, some of the people we go to meetings with might die.

— Bill Pate, NASA engineer

In the overall concept of quality, you'd probably expect *testing* to take some form of place. However, if you look at the [Consortium for Information & Software Quality](#) and their wording, there is no testing there. What gives? I think it's pretty easy: **Testing is just about verifying the [quality attributes](#) of what we built.**

And who else should do this, in terms of work ethics and competence, than the software engineer who wrote the code?

Danger

This is one of the strongest arguments I hold against testers and other professionals whose work solely relies on that small sliver of the big picture.

It starts with knowing what you are actually trying to do

Your #1 job is making working software. “Working” means *verified* working, and being *predictably* working.

If you don't know what you build and how it should behave, then you will have unclear and/or unclear code. Alas, such code cannot be meaningfully tested; at best you will have a brittle test, meaning it breaks or works on a whim. Such tests are even more dangerous than having no tests as they provide no clear answers, only more “questions”.

Success

The [functional programming paradigm](#) is one way in which we can optimize for highly deterministic systems since it heavily encourages [pure functions](#). We don't need to go quite so far to write software that works intentionally. Also, remember that JavaScript (if that's what you are using) is a multi-paradigm language that can [support many of the features of a functional programming language](#).

Nowadays, when pretty much everyone is a cloud engineer, the evolution of *DevOps* has reached the previously called *configuration management* realm and transformed it into *infrastructure-as-code*. With this notion, not only the code deterministically built, tested, and deployed—now we can do the same with “hardware” and infrastructural components, just as if they were malleable software. **Therefore, in a modern technology environment, the coding and testing will not only have to do with software, but also with infrastructure.** They are effectively tied together very closely, so old ITIL-style management won't really cut it anymore in the cloud-native world.

This is both good and bad:

- Good because it is an efficiency enabler for the well-versed engineer.
- Bad because it does require new skills that not all engineers find appealing.

Nevertheless, with all of these changes, I would argue that *generally*, conditions for writing, testing, and deploying good software are rapidly improving.

Testing to a high, confidence-creating degree should be one of the major parts of an adequate “definition of done”. Anything less will invite making testing and quality-enforcing procedures optional, which to be frank, means they won't get done at all.

No good test without good code

Information

Yet again, while the code is TypeScript and it should work just fine, I am doing this frameworkless and very raw so that you have a minimum of distraction in the way. The solutions are not necessarily the “best” either. If you want to try them out, you can just paste them into the [TypeScript Playground](#).

You’ll remember this from the introduction:

```
function add(firstNumber, secondNumber) {  
    return firstNumber + secondNumber;  
}
```

With the test being:

```
const expected = 5;  
const result = add(2, 3);  
  
if (result === expected) console.log("Test passed!");  
else console.log("Test failed!");
```

This will be a simple test that outputs 100% test coverage on all aspects: [line](#), [statement](#), [branch](#), and [functions](#).

The above function implementation is naive as it has no verification of the types of `firstNumber` or `secondNumber`. Let’s say we use TypeScript, then we could just do:

```
function add(firstNumber: number, secondNumber: number) {  
    return firstNumber + secondNumber;  
}
```

Voilà, types in action.

No doubt, that is better. In the context of the IDE, we will now be able to create warnings

or errors based on misuse of the function, at least in terms of the input types. With a `strict` configuration, we can even break compiling this code if someone tries to do something naughty.

This however won't stop anyone from *using* the code in a problematic way, for instance, if what you build is a library that others can use post-compilation time.

Additionally, it takes but a simple `// @ts-ignore` comment above your usage of the function to dispel any such rigor. And if you are allowing others to use the code, such as through a library, then you'll definitely need to ensure the type definitions are retained. What I am saying is not that TypeScript adds *nothing*, I am simply saying that it removes *many* of the errors that happen in the development phase and library usage or such, but there are still ways to crash the software. Types by themselves do not do all the heavy lifting; we need more.

In the vanilla JavaScript iteration, given that strings and numbers can be easily interchanged, we wouldn't necessarily break anything when it comes to simple additions of a numerary value provided as a string and one provided as an actual number. But what if some evil person starts passing in arrays, objects, symbols, or other things?

One solution could be:

```
function add(firstNumber: number, secondNumber: number) {
  if (!isNumber(firstNumber) || !isNumber(secondNumber))
    throw new Error(NonNumericErrorMessage);
  return firstNumber + secondNumber;
}

const isNumber = (item: any) => typeof item === "number";
```

This is better. We now have an actual, implemented safeguard to check the correctness of the input type. In turn, we get an additional function, `isNumber()`, that can also be tested. However, **it will already have been tested by default in the existing test**, so no need for extra work. Why is this? It's actually quite logical – your tests exercise (use) the same logical flow, so you will still hit the part of the code that runs `isNumber()`. Given your test inputs, you will end up with lower coverage on the branch statement if you don't also run a test with a non-numerical input.

Regardless, the code is still clean and nothing is “untestable” here.

Information

The check we did here is contextually relevant. For private class methods and “deeper” functionality that can’t logically be misused as easily we can afford to skimp on this kind of rigor. Your opinion may vary, but it’s just about keeping it simple and clean, and if there are firm checks surrounding a bit of code, there is no need to do *yet another* similar kind of check and end up with unnecessary test duplication.

Since we now do check for correctness, let’s add a test that checks that it does what it should:

```
// Add to previous code

function nonNumericTest() {
  console.log("It should throw an error if passing in non-numeric ↔
↔ values");
  try {
    // @ts-ignore
    return add("2", "3");
  } catch (error: any) {
    return error.message;
  }
}

const expected = NonNumericErrorMessage;
const result = nonNumericTest();

if (result === expected) console.log("Test passed!");
else console.log("Test failed!");
```

And it does!

Coupling and cohesion

Separate your business logic from the infrastructure. We will assume you are using AWS Lambda. In parts of the interwebz you might hear people claiming that you’ll need Lo-

calstack or something else to “test” Lambda functions. No, that is not correct. We can do things far easier and better.

Again I will take to quoting myself:

A relatively common “misimplementation” is to think of the Lambda handler as the *full extent* of the function. This is all straightforward in trivial contexts, but we gain a significant improvement by being able to remove the pure setup and boilerplate from the business side of things.

The semantic concept of “handler” is somewhat particular to how we talk about *function handlers* or *event handlers*. On a more generic software architecture note, this layer could often be translated into what goes into the “controller” term in the [MVC](#) school. I’ve been known to use the “controller” term and set a dedicated folder in the structure at an earlier stage in my career, but I now refrain from it and go with “[adapters](#)” instead, simply as it’s an ever wider concept and since we now open for *any* type of driver of our functions.

— [Domain Driven Microservices on AWS in Practice](#)

To be clear: Failing to separate business logic from the Lambda handler (or any other infrastructure) will put a quick end to testability. You end up having to set up a local stack or emulator just to verify basic functionality. You have almost no reason at all to care about the very first layer—“controller” or “adapter” or whichever nomenclature you prefer—if you know that it does nothing important. Expected things for the adapter to do would be to form the primitive shape of the input to, for example, the use cases (business logic). The use cases, being separated and all, are then completely in your control to test with a conventional unit testing framework to your heart’s content.

Inversion of control

Of the SOLID principles, perhaps the D ([dependency inversion principle](#)) offers the most power, albeit maybe not at first glance. By relying on abstractions rather than concretions and providing concrete implementations to our objects we can assemble our systems more Lego-like, making principally every object interchangeable. **If they are interchangeable, they become testable and mockable (if needed).**

In summary

We saw how TypeScript, compared to JavaScript, enforces strictness and makes the surface error for issues smaller. If you primarily work with another language the takeaway should be that strictness is a great quality and makes writing testable code easier as we let the compiler and language work for us. A smaller surface error with less logic is easier to test and typically relates to most conventional notions of code quality too.



Figure 1.6: Various books on better programming, design, and testing.

This is an area that without a doubt leads me to believe that it is only logical that a developer writes their own tests: It makes for a good litmus test of the structure and design of the code, as well as enforces the verification of the functionality according to expectations. Thus, tests help us write better code or at least qualify the code we wrote

against some external force.

Further, by writing both “positive” (happy flow) and “negative” (unhappy flow) tests, we also improved the code itself and made sure we have tests that accurately exercise the code for intended as well as unintended cases. In terms of celestial beings, unhandled exceptions should be treated as Satan – to be avoided at all costs, not by closing our eyes, but by preemptively handling such cases.

Integrate continuously, test continuously

Build a rock-solid, minimally complex pipeline to flow code to your users and shift culture to deliver frequently using that pipeline.

For the software delivery process, the most important global metric is cycle time. This is the time between deciding that a feature needs to be implemented and having that feature released to users. [...]

Projects concerned with the quality of their software often choose to measure the number of defects. However, this is a secondary measure. If a team using this measure discovers a defect, but it takes six months to release a fix for it, knowing that the defect exists is not very useful. **Focusing on the reduction of cycle time encourages the practices that increase quality, such as the use of a comprehensive automated suite of tests that is run as a result of every check-in.**

— Jez Humble & David Farley, *Continuous Delivery* (p. 138)

In the last chapter, I wrote about deterministic software. This determinism goes all the way to Continuous Integration (CI) and Continuous Delivery/Deployment (CD)—often put together as CI/CD. Continuously integrating was, and is still, one of the most foundational practices of agile; without it, it's practically impossible to be agile in any real sense of the word.

Use competent CI/CD tooling and automate all of it

A good workflow (pipeline) is fully automated, minimally cognitively complex, and puts value in people's hands as quickly and regularly as possible; often this assumes daily or several times daily. [DORA's research](#)—also available in the excellent instant classic *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*—similarly show a high correlation between elite-performing software teams and their use of CI and [trunk-based development](#):

Analysis of DevOps Research and Assessment (DORA) data from [2016](#) (PDF) and [2017](#) (PDF) shows that teams achieve higher levels of software delivery

and operational performance (delivery speed, stability, and availability) if they follow these practices:

- Have three or fewer active branches in the application's code repository.
- Merge branches to trunk at least once a day.
- Don't have code freezes and don't have integration phases.

— [DevOps tech: Trunk-based development](#)

Of the [DORA metrics](#), *Lead Time To Change* measures the time from committing code to it being deployed to a production environment.

Information

Did you know... [Since 2021, there is also a Reliability metric in the DORA metrics](#), which is maybe less known than the original four metrics.

For an elite-performing team, the *lead time to change is less than an hour*. Can you go from committed code to production in less than that? If not, I'd wager we have one or both of these general classes of issues:

- **People issues:** Such as mandated manual inspection, testing, confirmation, or similar processes such as code freeze.
- **Technical issues:** Such as suffering slow or flaky build and test automation, or not even having full automation at all.

With the focus of this book, let's turn to the technical issues.

Information

Don't have CI/CD today, or you're still on some godawful thing from 20 years back? There are lots of good CI/CD tools these days, such as GitHub, GitLab, Azure DevOps, Harness, Bitbucket... I'm sure you'll be happy with practically any of the major modern players out there. I'm not going to go into my personal

review of tooling here and now though!

The one thing in common with all CI/CD tools is that they all are scriptable and that you'll most often write a YAML-formatted file or specification that you keep with your source code. When the code gets pushed, the CI environment will pick it up and start performing its tasks according to your specification. It's not that dissimilar from scripting on any old Linux machine (you can often run other operating systems than Linux, but most commonly you'll find some Linux variant being used).

Success

If you've never scripted a CI workflow before, [take a gander at how it might look using GitHub Actions](#) which follows conventions that are pretty common among such tools.

Some of the things you should expect to see in a CI/CD pipeline could be steps for:

- Compiling code
- Installing dependencies
- Running quality and compliance tools (code quality tools, open source license checks...)
- Running unit tests and any other tests
- Deploying to any number of environments

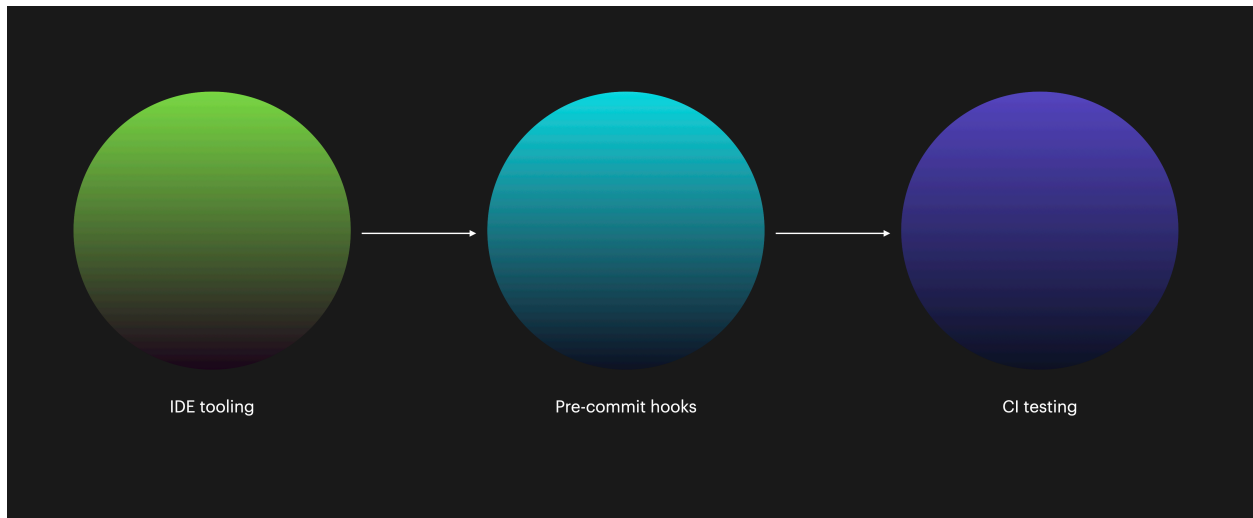


Figure 1.7: Trust no single place in the build chain. Run most of the same tools, all the way from your IDE to the CI build.

While it's certainly possible to write really elaborate, complex CI/CD strategies, the heart of it is really about building, testing, and deploying your things. Aim for simplicity over elaboration in your CI/CD setup.

Insist on multiple integrations and deployments per day

Once you have a good CI pipeline it's however not necessarily golden times just yet. Now comes the moment of cultural change, as you'll want to get your team to potentially change how they've grown accustomed to working for years, maybe even decades.

Information

For more on these types of subjects, see Dave Farley's extensive set of [videos shared on YouTube](#). You'll find some hot takes and well-considered notions on things like Agile, TDD, and TBD.

Regardless of your overall CD strategy ([continuous delivery](#) or [continuous deployment](#)), what you want is to minimize the code sitting idle, being outside of the production environment. The whole phenomenon of Continuous Integration insists that the code ends up in the mainline branch and that it goes to the production environment, not that it

stays idle in a lower environment or on your machine.

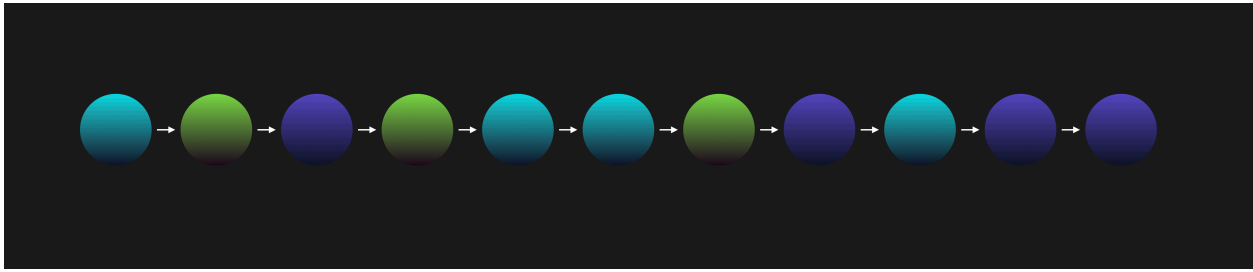


Figure 1.8: Trunk based development—just a long chain of things happening after each other.

Besides having competent technical tooling available which you can trust and rely on, you'll want to feel confident in your code not behaving incorrectly. No surprise: Test automation is the best way to do that. **Testing your code often is a well-established way to know that your code and tests are well and truly exercised.** By minimizing the time between commits you will end up with, relatively between two adjacent commits, a smaller set of changes at play in each instance. While a change, large or small, can be breaking it's usually true that **bigger changes create more surface area for something to fail.** Infrequent and big commits lead to more problematic code—just go ahead and start pushing small commits frequently! Don't forget to run all the tests every time while you're at it.

In summary

In closing, view anything that hinders your team from integrating and deploying as often as possible as a tangible, clear risk to your quality and delivery.

Going from traditional and often problematic ways of doing CI/CD is not always easy, and you'll have to be a good leader if you're the one taking your team from such a state to CI/CD nirvana.

Separate deployments from releases with feature toggles

Deferring decisions on code paths to something outside of the code itself is a really smart idea if you balance the trade-offs that come with that power.

To mitigate some of the risks that still might exist (i.e. the unknown-unknown class of problems), we can **separate the deployment, as a technical concept, from the release, as a feature availability concept.**

A **deployment is never more than a technical construct**—this is the point at which new code is pushed out to the hardware that will eventually serve it to users. It is often unfortunately and falsely conflated with a *release*, a residual aspect from an age of “dumb hardware” in which the hardware simply runs the software on it, being oblivious to any dynamic switching of code paths. However, there are now (and have been for a long time) very simple ways in which this inflexible factor can be overcome: One way is using *feature toggles*.

[Feature toggles \(or feature flags\)](#) are a mechanism by which a piece of software can, either during run-time or build-time, dynamically alter its behavior with what is effectively no more than an “if-else-then” implementation in the software. Feature toggles make it easy to separate a deployment from a release if we can “hide” new features behind such a toggle and enabling it only for the appropriate audiences. We can safely release new software without worrying about new functionality being used until the “feature is flipped on” for everyone. It’s an imperfect pattern, yes, but it has stood its test against time by removing more problems than it causes.

The secret sauce is very simple: Separate the configuration (called the *feature* or *flag*) from the code that will run that feature. In effect, we **defer something that is typically baked into the code to an external resource** which will respond with some relevant information on what the code should do. It would work something like this:

- [CODE] My context is that I am running for a logged-in beta user. I am now at a branching point in the code regarding the display of a new feature. I’m asking you (the feature toggle service) whether or not to display a panel with new, not generally available options.
- [FEATURE TOGGLE SERVICE] For this user type you should display the panel.

- [CODE] OK, thanks, I'll show the panel.

A simple example

Enough said, time for a coded example, taken from [one of my Gists](#).

```
/**
 * @description API that caters for current (production) requirements, ↵
 * ↵ as well as future (beta) data needs.
 * Uses a kind of primitive feature toggle to switch version, without ↵
 * ↵ needing to deploy and maintain two separate backends.
 */
function api(event) {
  const clientVersion = event?.headers["X-Client-Version"];

  // Return data structure expected of current version
  if (!clientVersion || parseFloat(clientVersion) < 1.1) return ↵
  ↵ dataCurrent();

  // Return new data structure
  return dataBeta();
}

/**
 * @description This is the current production response.
 *
 * @version 1.0.0
 * @deprecated Will be deprecated after version 1.1
 */
const dataCurrent = () => ({
  response: "This is the current production response",
});

/**
 * @description This is the new beta response.
 *
 * @version 1.1.0
 */
const dataBeta = () => ({
  data: {
    text: "This is the new beta response",
  },
});
```

```
});

/**
 * @description Call API with both current and future application ↔
 * ↔ versions.
 */
const responseProd = api();
console.log("Production response:", responseProd);

const responseBeta = api({ headers: { "X-Client-Version": "1.1" } });
console.log("Beta response:", responseBeta);
```

It's a very basic piece of software but it still manages to make a rather big point that at least I am not seeing enough in production software in my own experience: We can use a basic mechanism (in this case, a header) to drive traffic dynamically rather than resort to heavy-handed infrastructural segregation or using network/DNS segregation.

Client A (v 1.0, the public production version) calls backend with:

```
GET www.domain.com/api
```

Client A expects:

```
{
  "response": "This is the current production response"
}
```

Client B (v 1.1, the limited beta version) calls the same backend with:

```
GET www.domain.com/api
```

This one expects to get back the following shape:

```
{
  "data": {
    "text": "This is the new beta response"
  }
}
```

While the technical artifact is always one and the same it is “intelligent” enough to dynamically respond to two different code paths. Using good software architecture and competent development practices, the actual coded implementation itself should be of robust quality, being decoupled and well-structured to allow for this behavior without cringe-worthy logical holes.

Information

Another useful pattern when dealing with multiple active code paths is [branching by abstraction](#). Instead of using branches in your version control (Git) you do this in your code. A key benefit is that you can maintain continuous integration—which you effectively can’t if you only sporadically merge back as with feature branches—and logically handle any unfinished code.

Some concerns you will have to address

Take note that a few concerns will pop up as a consequence of using feature toggles:

- Will you buy or build a tool?
- Will you have a significant cost attached to using/building/running such a tool?
- Do you also want to make a cultural move and train your business colleagues to use such a tool to release features, or do you still let developers handle releases, but just with another tool?
- From a solution architecture side, how are you changing the characteristics of your software if you retrieve toggles for every call (latency, cost, usage quotas, added complexity...), or is it more appropriate to “bake in toggles” during build time? What are the trade-offs you want to make?
- How will you handle fallback states if you don’t successfully retrieve a feature toggle?
- How will you ensure proper lifecycle management of feature toggles in code as well as in configuration? You won’t want if/else statements from an old feature lingering about longer than necessary.

While it seems like we are inviting a lot of new problems into our house by using feature toggles, they do offer a powerful, easy-to-understand way of achieving a critical benefit that is well worth fighting a bit for.

Information

Feel free to also read more in-depth material at the [feature flags article on Martin Fowler's site](#) as well as on the LaunchDarkly-sponsored [FeatureFlags.io site](#).

Use modern technical practices

“I have nothing against the professionalisation of programming; it would be a good thing. But the fact remains that the great majority of programmers are not engineers. We are tradesmen, a bit like plumbers or domestic electricians. There’s nothing to be ashamed of in being a skilled tradesman; for a good part of my career, a plumber earned more than I did.”

— User “[denton-scratch](#)” on YCombinator

We’ve seen (depending on your specific area) languages come and go, and trends blossom and die. Whether you are a “skilled tradesman” or professionalized software engineer, have one foot in conventions, standards, wisdom, and the core foundations of software engineering and the other foot in the flow of time and the novel ideas and tools that come along to stay informed.

For that second part, the following are a few of the things I’ve picked up that have certainly made my deliveries better in terms of quality.

Information

Google’s DevOps site is a fantastic resource that I highly recommend you look through.

Work in a DevOps model

Few things are as powerful in human psychology as making someone feel responsible for something. One of the key learnings in DevOps is that it’s very little about technology, and very much about creating accountability in the team to “own it” from start to finish throughout all of the software’s phases, which of course also includes testing. Given the tools to act on that ownership, few will look back nostalgically to their old-school “throw it over the wall” dev teams.

Testing benefits: Faster feedback, increased ownership.

Own your CI/CD pipe

We've already discussed this, but it's a complete game changer if you aren't yet familiar and comfortable with having the power of taking your code from laptop to customer. Of course, this is a cornerstone technical feature of the above DevOps aspect too. Push as often as you have code that deserves being pushed and run your tests on every commit.

Testing benefits: Continuous testing.

Configure your code with infrastructure-as-code (IAC) tools

It's a very old complaint among developers to rant about environment issues, misconfigured infrastructure, or environment disparity. Using an IAC tool like Terraform, AWS CDK, or Serverless Framework, you get the possibility to define (in code) what the cloud infrastructure should be and you can safely leave it to the tool to converge those states for you. You end up with a whole lot fewer of those complaints when you start using IAC.

Testing benefits: Less brittle and flaky infrastructure, greater clarity into system state and configuration.

Migrate to Typescript if you are using JavaScript

If you, like me, develop primarily in the JS/Node stack I can't really stress enough the quality-of-life improvements that Typescript adds, not to mention the impact it makes to overall quality and ensuring "dumb bugs" are more or less instantly vanquished. It may feel like a bump in the road at first, but you should know that TypeScript offers the possibility to gradually move into it, even with pre-existing codebases. Many JavaScript/Node developers can be a bit shaken by all the new things coming in — interfaces, types, generics, class orientation, and more — but again, this can be taken in gradually. For me, it took me back also 10 years to when I was writing a lot more of that kind of code, and it offered me much clearer traction on applying [design patterns](#) ([see here for examples](#)) than regular JavaScript gave me.

Information

The one book to get on TypeScript is Dan Vandenkam's [Effective TypeScript: 62 Specific Ways to Improve Your TypeScript](#). Do it; get it now.

Testing benefits: Greatly increased language capabilities to write better code, reduces some needs in validation and dumb checking.

Refactoring

The one skill that marks the kings and queens of software engineers is being good at refactoring, the systematic improvement of your code. A common unspoken belief seems to be that code is *good* (or not) instantly and at the point of being written. In fact, code *evolves* as we pass the code over from its crude first implementation to an increasingly better, more readable, more maintainable, less error-prone state. Therefore **we need to accept that code grows better and the application art/science of doing so is refactoring.**

You don't ask for time to refactor (at least unless it's critical and something that will set you back significantly in time and effort). The point of refactoring is that we evolve code from one state to another, better one. **Passing over code and seeing that this could be improved and light-handedly improving it, say in under a few minutes, is not a task to be ticked off the list – it's part of the continuous positive evolution of your codebase.** This works well with the ubiquitously-mentioned boy scout rule ("always leave the code better than you found it").

However, for significant changes, refactoring in a stricter and more precise sense still matters – it's just that you may need more tools from the toolbox!

Information

One of the better online sources on this would be [Refactoring.guru](#) and the undisputed book is [Refactoring by Martin Fowler](#).

A brilliant thing about your tests is that, given you have tests, they should not fail after

having refactored your code.

Testing benefits: Continuous refactoring leads to “better” code, often making testing easier.

Use a well-considered software architecture

Using a conventional and well-working architecture such as any of the [hexagonal types](#) (I propose the [Clean Architecture](#) derivation) will put your code in a more rigid and understandable place. Hexagonal architecture is also good at decoupling relevant hierarchies of the code and is a surefire way of improving the logical separation, especially if you are a more junior developer.

Understandable and well-decoupled code is testable code. More on this later in this chapter.

Testing benefits: Better understanding of the code base and the way code interacts, may lead to better code and therefore easier and better tests.

Be literate with the cloud, including microservices and serverless

More and more of our technical solutions are cloud-based or cloud-native, and it’s getting a lot more common to see microservices architectures being used, as well as the steady growth of serverless technologies.

Both of these concepts are major and respectively disrupt a lot of old truths and assumptions on how we build, run, and test software. Being adept at these styles and technologies is something that does not come from prior experience with virtual machines and classic architectures — you will have to take the time to be proficient in them. Also, respect that they change the testing game a bit (though not always in the expected ways). In our context, we’ll assume both microservices and serverless so I’ll do my part in training you.

Testing benefits: Certain types of tests get more important than others, which may lead to focusing on lower-level tests (e.g. unit tests) over more complicated types. Also means greater separation between “your code” and infrastructural stuff from the cloud

host – don't test third-party code.

Consider TDD

Maybe you are surprised this is at the bottom?

Personally, I have never tried hard enough to make [Test-Driven Development](#) (TDD) “my” game, and I think most folks are in unison that as long as there is the required degree of testing done *before* actually committing your code, then everyone wins. I am personally happy to write tests and then do any minor mods to my code, *after* writing the first complete-ish bit of code.

The main point is that there is, at a minimum, always sufficient testing and that it happens close in time to the writing of the code.

Testing benefits: You always have tests, code is structured to provide the absolute essence of the need/use case.

Design and think before coding

Focusing on the use and problem, rather than everything around it, makes development more effective.

Looking back at [Verner Vogel's advice on APIs](#), we read:

Work Backwards from Customer Use Cases

As we can see, there are two important pieces here, the first being “working backwards” and the other being “customer use cases”. Let’s start with *working backward*:

The Working Backwards product definition process is all about is **fleshing out the concept and achieving clarity of thought about what we will ultimately go off and build**. It typically has four steps [...]

Once we have gone through the process of creating the press release, faq, mockups, and user manuals, it is amazing how much clearer it is what you are planning to build. We’ll have a suite of documents that we can use to explain the new product to other teams within Amazon. We know at that point that **the whole team has a shared vision** on what product we are going [sic] the build.

— [Werner Vogels](#)

Information

Concerning “working backwards”, to understand that specific part of the advice, see:

- [Working Backwards](#) by Werner Vogels
- [Working backwards: The story behind the AWS Cloud Development Kit](#) by AWS, as an example of this

This is a brilliant, cheap, and powerful way to get to the core of what you need to build instead of jump-starting with building something before we know what it should even

be. If Amazon, notoriously good at producing new ideas and services, can do it, then why not at least try to learn something from this, flipping the script for a bit?

At the time of starting to write your code, you already have some idea what the API, or user's interface and needs towards your code, will be.

And what about code at this stage? Without rehashing all kinds of quotes and books, a way to write better code could include an understanding of code as being about layers, either being exposed or unknown. While such a concept is very tangible with classes (`private` and `public` methods) I mean this in a wider sense: Each layer is on its own as simple as possible and utilizes indirection to let other functions/classes/etc. take care of what they do best. By writing the simple code possible, locally, we compose, globally, a more sophisticated solution. Each part is easy to understand, well-written, and easy to test. If this sounds like something you learned early in your training, then I can only say, "yes" indeed. But why do we lose sight of the training we've taken and the things we've learned? There is a reason for such concepts and principles.

We need a way to connect the *product vision*, its *design*, and its *implementation* in a flexible yet correctly aligned way. As this book is not on product vision or team exercises, we'll leave that part off the table. But when it comes to design and implementation, we definitely can say a few things about that! Let's now look at some actionable ways to *work from the back*.

Write, make bullet points, doodle, or diagram your way to a design

Which comes first: Bad code or a bad solution? I strongly believe that most *really bad* ideas can be fended off if we think about what we are doing first. One under-utilized capacity we have is to think in a concentrated and structured way. Many of us are too keen on jumping the gun, hovering over the keyboard, and smashing out a solution, but:

Code is not important — your solution to a problem is. The code is a vehicle to deliver the solution — it's not necessarily the same vehicle that also solves the design itself (i.e. the solution).

There are plenty of ways to do this:

- Write in plain text what you are trying to solve

- Tell someone in plain English (or whichever language you use) how you are trying to solve a problem; also known as [rubber duck debugging](#)
- Make bullet points of how you can solve the problem sequentially
- Make a doodle or diagram of some kind to express how the solution works
- Explain or write what the solution is supposed to solve and for whom

This works just as well with others, as with yourself. In such cases, consider appropriate tools such as whiteboards (or collaborative online whiteboards) or anything physical or digital that is collaborative in nature. Even something simple as co-designing a JSON blob can do wonders when problem-solving!

Use scratch files to remove complexity from your problem solving

When it's time for at least some code to be written, I will sometimes do my coding in a scratch file—maybe not even in TypeScript, just regular old JavaScript—and continually run it, either manually or ideally with something like Jest which can watch files for edits (in effect being close to TDD, though not strictly TDD). When I mean scratch file, I simply mean a `.js` or `.ts` file that I will run outside the context of the code base. This will be removed later. You can also easily do this in a big pre-existing project if you want to isolate the logic in a crude but effective manner.

The scratch file is not more efficient per se, but gives an undeniable focus on just solving that specific part of the puzzle. This way of encapsulating the problem solving is, at least for me, a way to decomplexify the circumstances to the minimum required details. I won't *always* do this, but it works especially well when you are in complex circumstances, and testing the specific feature you are building is not necessarily super easy.

Once we are done we can easily transport the code into the actual code base. At this point, you will already have actual tests, or at the very least, some test data you've had to construct.

Write the “API” before the code

A divergent but related evolution on the above is thinking of the initial work as *designing the API*, or in other words, the public interface that will be used.

In the actual web API community there is something called literally the API-first approach, in which the API definition is written before any implementation. You could take inspiration from this and, given that you work with literal APIs, also write your specs in OpenAPI or AsyncAPI or whatever to design the actual API. There is no code needed for this, yet you take on many of the design challenges without encumbering yourself with needless implementation detail.

The good thing with this is that you could now mock the expected behavior very quickly and work “backward” from the intended final public state to filling in the missing implementation details needed.

Once you start to intuitively separate the outer shell (the API, what others see) from the inner workings you will get much closer to being “good at testing”. Given that tests don’t check internal implementation detail (which is invisible to the test; only the request and response are known) you are stuck with the fact that the API or outer surface has to be “good” – expressive enough for the user/caller and rich enough to pass in the required information for the processing to be made correctly.

This can be done either as full-on TDD, in which case you are effectively writing what the expected usage should be first anyway, or writing some of the (known) logic from a user’s perspective first. Doing this is no harder than setting up (as above) a scratch file to toy around with, or writing “fake” docs in your README.md, an approach that is somewhat similar in spirit to the [Amazon 6-pager](#) though not quite as long!

Lean towards use-case driven APIs

Centering on use cases in your application helps focus on the actual needs. By directing attention first to the *need*—and designing, exposing, and mocking a solution for it—we are able to create a realistic product rapidly and fill in the detail as needed to the extent we require later on. **This should be an iterative rather than an incremental phase, as we continuously move toward a more evolved state.** In incremental development we add work like bricks to a wall – the wall is never complete and functional without all of the bricks, making change sometimes hard to account for. In the iterative mindset, however, we begin with a strip of tape (the most minimal substitute for the wall) and gradually learn and progress until we have the desired qualities. Maybe we didn’t need a brick wall, but a plaster wall – learning this with the incremental approach always results in waste and is seldom a good thing; in the iterative approach, it’s the opposite. That’s a lot closer to the spirit of Lean and Agile, and also much more fruitful as an approach for software engineering.

A system as a set of use cases

We use the word “*use case*” but it’s perhaps time to understand it a bit more_. Use-cases_ can certainly be generally understood as broadly meaning “something the system does”, but in the context of software engineering and Agile in particular, the term is attributed to [Ivar Jacobsen](#) who minted the term with a specific intended meaning.

“For instance,” says Jacobson, “**a system with a dozen use cases is a reasonably good piece of software.** But when you scale it down – or zoom in as we say – you’ll get maybe several hundred user stories, maybe even thousands of user stories. The problem with all these user stories is that, to understand the bigger picture, you’ll end up drowning in the details, and user stories are usually not maintained after you have developed the system. Whereas **with use cases, the big picture is more easily available to help teams understand how a system will be used, and the value it will provide to its users and other stakeholders.**”

Agile is touted as being a means to respond to market feedback, user feedback, and customer feedback in the business setting. When asked how use cases support organizations becoming more Agile and responsive, Jacobson shared

that, “with use cases in Agile development, you don’t throw away everything when you go from release to release – you keep most of it. And the value in having this persisting use case picture is that you can gradually grow it and add more use case slices” as you receive feedback from customers and the market.

“Every use case slice is a number of user stories that bring value in some way to the customer. But value is generated only if the product you’re delivering is actually embraced and used by your intended audience, so it is much better to focus on delivering usable value than on long lists of the functions or features your product will offer.”

— [Use Case 2.0 and its Role in Agile Software Development](#)

Consequently, accounting for your system as a set of use cases makes it easy to reason about what your system does in plain language.

A software architecture approach that takes use cases as a literal layer is [Robert C. Martin’s Clean Architecture](#). I’ve used it and evolved my own use of it for some years and feel that it provides many tangible benefits to my work as a software engineer. I highly encourage trying to use it as a disciplined approach.

Using Clean Architecture as our foundation

Information

This section is lifted from [my book on DDD and its “Modules” section](#). Consider reading that as well, since there is quite some overlap.

The “Clean Architecture” is a relatively well-known variant of the onion/hexagonal/ports-and-adapters school of architecture.

Many have tried and many have failed when it comes to setting up a folder structure for DDD. For my part, I’ve found that Robert C. Martin’s “clean architecture” is a better (and simpler!) elaboration of where so many developers have tried to find a way. It’s not magic, just a very nice mapping (and [blog article](#), and [book](#) for that matter!).

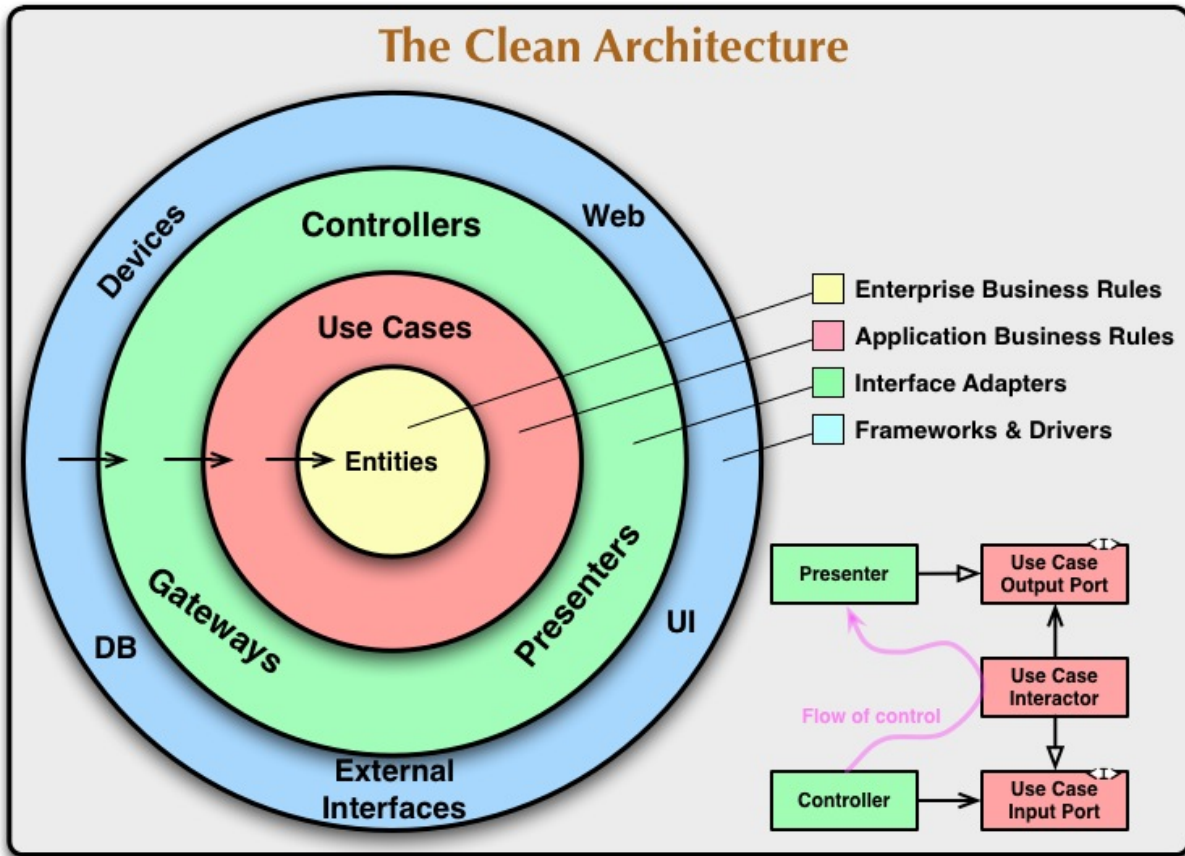


Figure 1.9: From Robert C. Martin's blog. <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>, 10 August 2012.

I find it the most immediately effective and neat variant of these, as it:

- Introduces very little in terms of novel concepts;
- Is almost directly compatible with how DDD envisions structure in the software realm;
- Powerfully exploits the *dependency rule* for well-working and testable software.

Robert Martin writes about the *dependency rule* like this:

The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.

The overriding rule that makes this architecture work is *The Dependency Rule*. This rule says that *source code dependencies* can only point *inwards*. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the inner circle. That includes functions, and classes, variables, or any other named software entity.

By the same token, data formats used in an outer circle should not be used by an inner circle, especially if those formats are generated by a framework in an outer circle. We don't want anything in an outer circle to impact the inner circles.

— [Robert C. Martin: The Clean Architecture](#)

The intention with all of these ideas for how to structure an application is all well-meaning, but I've also seen and reflected on how a higher level of "layers" or "circles" can complicate things quite quickly.

Let's at least look at the levels and some examples of what would go into each, respectively.

- **Entities:** "Business objects of the application"
- **Use cases:** "Use cases orchestrate the flow of data to and from the Entities, and direct those Entities to use their enterprise wide business rules to achieve the goals of the use case"
- **Interface adapters:** "A set of adapters that convert data from the format most convenient for the use cases and Entities, to the format most convenient for some external agency such as the Database or the Web"
- **Frameworks and Drivers:** "Where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside where they can do little harm"

Ultimately: **The farther in something is, the less likely it is to change. Any inner layers must not depend on the outer layers.**

In closing

Using the Clean Architecture is one way you could use a disciplined and relatively well-known approach to segmenting your code, helping you write better-structured code that will most likely be easier to test.

Provide an API schema and examples

Having an API schema is the bare minimum acceptance criteria for a system to be used by others. It's common that *not having it* leads to a lot of questions coming from other teams and stakeholders, filling up your time helping them, instead of building and improving on your solution. While it sounds altruistic to produce a schema and docs “for others”, in fact, you win a lot by doing so:

- Other people know what your thing is, and how it works
- It moves the design surface into a less technical place, meaning you can work on the API easier and more nimbly
- It's easier to onboard new team members

The simplest you can go, with basic systems, is to just write down endpoints and some example usage in your README.md file. While not a schema per se, it's a good start.

An example of this from my project <https://maxslaofmy.systems> is below:

```
POST https://maxslaofmy.systems/api

[
  {
    "name": "amazon-api-gateway"
  },
  {
    "name": "aws-lambda"
  },
  {
    "name": "custom-my-heroku-backend",
    "sla": 97,
    "description": "Just some backend I built on Heroku"
  },
  {
    "name": "amazon-dynamodb"
  }
]
```

```
Status 200
```

```
96.893
```

Already this, together with other documentation, gives end-users a useful way to start interacting with your API.

As soon as the API starts becoming complex or it needs to have a portable definition—for example for [CI-based linting and validation](#), [document generation](#), or other machine-readable contexts—then following a standardized convention is something you’re absolutely going to want to do. The two major formats today (for the web context, at least) are OpenAPI and AsyncAPI.

Information

Read a short introduction to [doing this in practice with AsyncAPI in my DDD book](#). Learn more about writing schemas at:

- [SwaggerHub OpenAPI 3 tutorial](#)
- [AsyncAPI tutorial](#)

For more on technical documentation writing, two great resources are:

- [Diátaxis: A systematic framework for technical documentation authoring](#)
- [Principles of technical documentation](#)

The minimum you should aim at providing is:

- Technical documentation with relevant headings; [see an example template here](#)
- API schema and/or API usage examples (requests, responses)
- That the above is located somewhere other teams and stakeholders have easy access to the material

Offload some validation to the API level

Information

Read more about this in practice in the [“API schema validation” page on Better APIs](#).

If you are able to use a managed API solution such as AWS API Gateway, then you should be able to add request validation at the API level. This will act as a further guard against invalid POST requests.

Request validation happens on the **structural and pattern-based layers**. While this allows for a relatively rich validation manner, it is not something that works for logical validation or other types of checks that you’d need conventional programming for. Such checks are still firmly in the territory of application layer validation and testing.

If you already have an API schema, constructing the validator schemas (in the AWS case, this is in [JSON Schema](#) format) is merely a question of copy-pasting a bit of that data into a separate document. For more on these details and how it actually works, see the above link.

Testing validation

What does this mean for testing, then? You could for example **validate the validation logic** during your integration tests. You could create a number of test cases that ensure that expected inputs work, and that invalid inputs throw the correct error codes.

Does this replace any other testing? No, not really, but it does complement it. We are really just enforcing our solution with one additional layer that goes on top of the API. In case your system isn’t even an API, well then none of this matters anyway!

What you do get is some guarantee that invalid input should not be leaking into your application layer. If you are already doing logical input validation you’re still going to need that, but you can perhaps start shaving off any tedious structural validations you’ve set up.

Towards modern testing practices

To me, legacy code is simply code without tests.

— Michael Feathers, [Working Effectively With Legacy Code](#)

While the previous section was devoted to delivering competent software engineering, this section proposes directions to thinking and practically conducting our testing.

The approach we should follow is called **continuous testing**, something which Google describes in the following way:

The key to building quality into software is getting fast feedback on the impact of changes throughout the software delivery lifecycle. Traditionally, teams relied on manual testing and code inspection to verify systems' correctness. These inspections and tests typically occurred in a separate phase after "dev complete." This approach has the following drawbacks [...]

Instead, teams should:

- Perform all types of testing continuously throughout the software delivery lifecycle.
- Create and curate fast, reliable suites of automated tests which are run as part of your continuous delivery pipelines.

Not only does this help teams build (and learn how to build) high quality software faster, [DORA](#)'s research shows that it also drives improved software stability, reduced team burnout, and lower deployment pain.

— [DevOps tech: Continuous testing](#)

In this section, we will address many of the concerns and theoretical parts relating to testing in a modern technical environment and how to actively move towards working with them. Among some of the topics, you'll get to know:

- What types of testing exist
- Some of the testing models and how they compare to each other

- What goes into writing a good test
- Why conventional wisdom on testing environments and test data is wrong
- Why testing serverless isn't that different, if at all

Types of testing

It is impossible to test absolutely everything, without the tests being as complicated and error-prone as the code. It is suicide to test nothing (in this sense of isolated, automatic tests). So, of all the things you can imagine testing, what should you test?

You should test things that might break. If code is so simple that it can't possibly break, and you measure that the code in question doesn't actually break in practice, then you shouldn't write a test for it...

Testing is a bet. The bet pays off when your expectations are violated [when a test that you expect to pass fails, or when a test that you expect to fail passes]... So, if you could, you would only write those tests that pay off. Since you can't know which tests would pay off (if you did, then you would already know and you wouldn't be learning anything), you write tests that might pay off. **As you test, you reflect on which kinds of tests tend to pay off and which don't, and you write more of the ones that do pay off, and fewer of the ones that don't.**

— Kent Beck: [*Extreme Programming Explained*](#)

And:

[...W]hen anyone starts talking about various testing categories, dig deeper on what they mean by their words, as they probably don't use them the same way as the last person you read did.

— Martin Fowler, [*On the Diverse And Fantastical Shapes of Testing*](#)

There are several dozens of testing types that you could painstakingly perform. Even worse, it's not uncommon to find contention around the exact terms and meanings of them! So, yeah, testing has grown into a minefield. Just like with code itself, tests can be done in so many different ways, shapes, and forms that finding what tends to “pay off”, as Beck wrote, can be quite the gamble.

Why so many types of testing?

Tests are instruments to validate (assert) hypotheses about the expected behavior of our software. Given that software is complex, changing over time, multi-modal, multi-platform, and doesn't have generically applicable standards the same way we could expect of many other industries or professions, we end up with a huge array of testing types.

Further impacting the situation is that test tooling has to adapt based on whether or not we can (and want to) allow testing of the source code, or if the testable source is happening from (for example) the user interface. So, while a test may often be performed in several ways, there is an obligation on us to make a wide and informed "cut" across which testing types should be considered for a given system. The occurrence of a specific sort of test will depend on your use case; a backend will not need a visual test, however, the frontend using said backend will certainly want to test against it (since it "consumes" its services). Factors we might want to consider when choosing testing types are:

- **What** can I access (the code, the UI, something else)?
- **When** do I run the test?
- **Where** do I run the test?
- **How** do I run the test?
- **How frequently** do I run the test?
- **Why** is this the best testing type for my need?

If I had been a smarter person I would have a diagram tying it all together... Luckily, for our intents and purposes, we can just pop up Cindy Sridharan's illustration of some of these techniques and how they stack across an application lifecycle and use that as starting point:

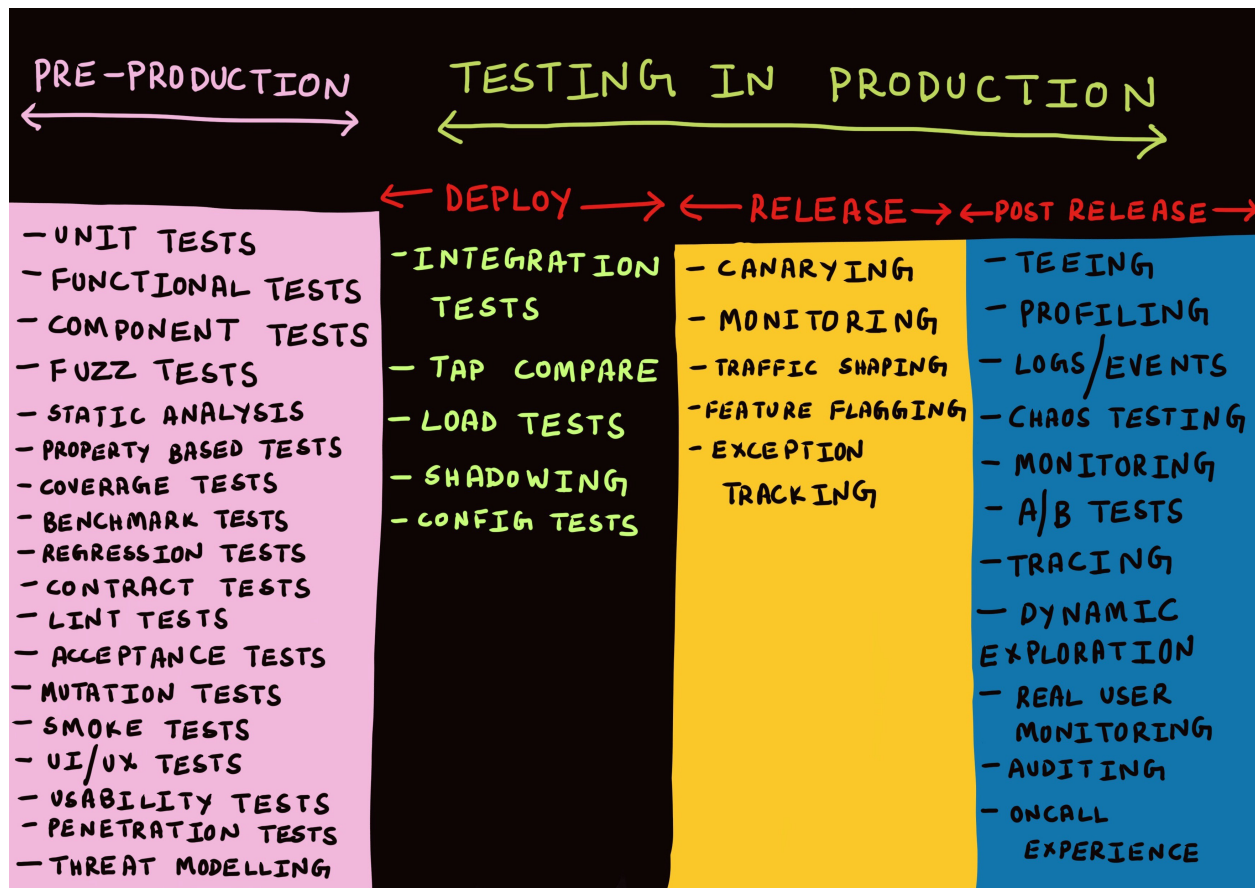


Figure 1.10: From <https://copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1>

What I love about the diagram is that it puts the temporal (i.e. time) perspective on top of all the test types. Not all testing types are made equal, and they have their place in different parts of the lifecycle; some are great at finding errors in a known context (linters, compilers) while some tooling will never be meaningful *before the fact* (monitoring, tracing, load tests). This is a good thing, as we can reduce the mass of types into something much more manageable.

The purpose of a test

Every testing type has a *purpose*. While that may sound self-evident, I do believe that many miss this point in actuality. If something has a purpose, it must be ideal to use that thing (testing type) for that specific purpose, wouldn't you say? Let me give you a bad example of this: It's not uncommon for developers to love the idea of integration tests

and start writing them and making them validate the business logic of the tested system. Why on earth would you do that?

- The integration test’s purpose is to *ensure the integration between points A and B works*. This test does something much more detail-oriented.
- The integration test is *outside the code scope of the system under test* and is therefore in effect a *third party*—you don’t test third-party code.
- The business logic (or similar) should be validated by one part only, the system itself. This is because the misguided attempt is fundamentally about completing a *logical assertion*, not doing an infrastructural test.
- Clearly, the purpose of the test is misguided and unit tests might serve as a better option. If deemed important enough, perhaps the integration test could be reduced and simplified and kept as a set of “smoke tests” to ensure typical valid cases send back the correct status code. Now we are coming closer to the purpose of each test.

Success

Push tests “to the left” An argument I have made, and will continue making, is that we need to start pushing our tests “to the left”, meaning we should get developers to write them and overall go for “simpler tests” (such as unit testing and static testing). The vast majority of needs you have when building a piece of software are about the software and its logic and much less about infrastructural concerns.

What to focus on

From the above, the ones I think you should know *well* are:

- **Static analysis:** Under this broad term **lint testing** can also be considered a part,
- **Unit testing:** A concept that should by itself include **coverage/functional/component/regression** tests from the above table,

- **Integration testing:** Also known as **API testing**.

These following, additional ones I wouldn't strictly call *necessary* in the same way as the first ones, but they all still provide useful value for special cases:

- Contract testing
- Smoke testing
- Synthetic testing
- Load testing
- System testing
- End-to-end testing (tested through the UI)

Again, all of the above will be covered in the Running tests [in](#) practice section.

Does anyone actually run all those tests?

With “anyone” I feel confident to say that highly tech-oriented organizations definitely seem to run a very rich set of tests. If you think about it, for an organization like Google or Spotify it would make a lot of sense to run the majority of tests you saw in the table.

Do any of the places I've worked at have such rigorous schemes? No. I guess that would go for probably most companies or organizations.

Should *you* run all types of tests? No. Run what makes sense. Also, remember that different parts of your technical landscape may have different ideal testing schemes.

However, with this said, the top three ones—unit testing, integration or API testing, and static code analysis—should be around without exception. Their relative simplicity, ease of execution, and high-value-low-effort touch make them brilliant in any scenario where you have direct source code access.

Pre-production testing vs production testing

Another effect I am seeing *a lot* more pronounced in tech-heavy companies are the mature and rich practices used for “testing in production”. I’ve yet to actually work in a company that does this at any mature level.

We will look at testing in production later in this chapter, but the takeaway here and now is that despite your best intentions, tests won’t ever cover all the potential issues your software might run into. Modern systems, running in distributed landscapes, across an almost magical ecosystem of technologies, are fantastic in many ways; but that magic is frail. Testing in production is a reality check that conventional testing is first and foremost about the known states of your software and that a different toolbox is needed for the unknown things—which will not happen in your pre-production environment but typically inflicts itself on customers in your production environment. Hence the name.

More on this later.

In closing

We’ve raced into the great big area of testing and also taken in a few different practical notions that can guide us later:

- All types of tests have a temporal place in a lifecycle.
- What tests make sense in a given case is context-dependent.
- Every test type has a purpose and a part of your job is to apply the best type of test to each need.
- We need to accept that tests are pretty much only about that which we know, so we need other tools for the unknowns.

Next up, we will discuss testing models.

Testing models

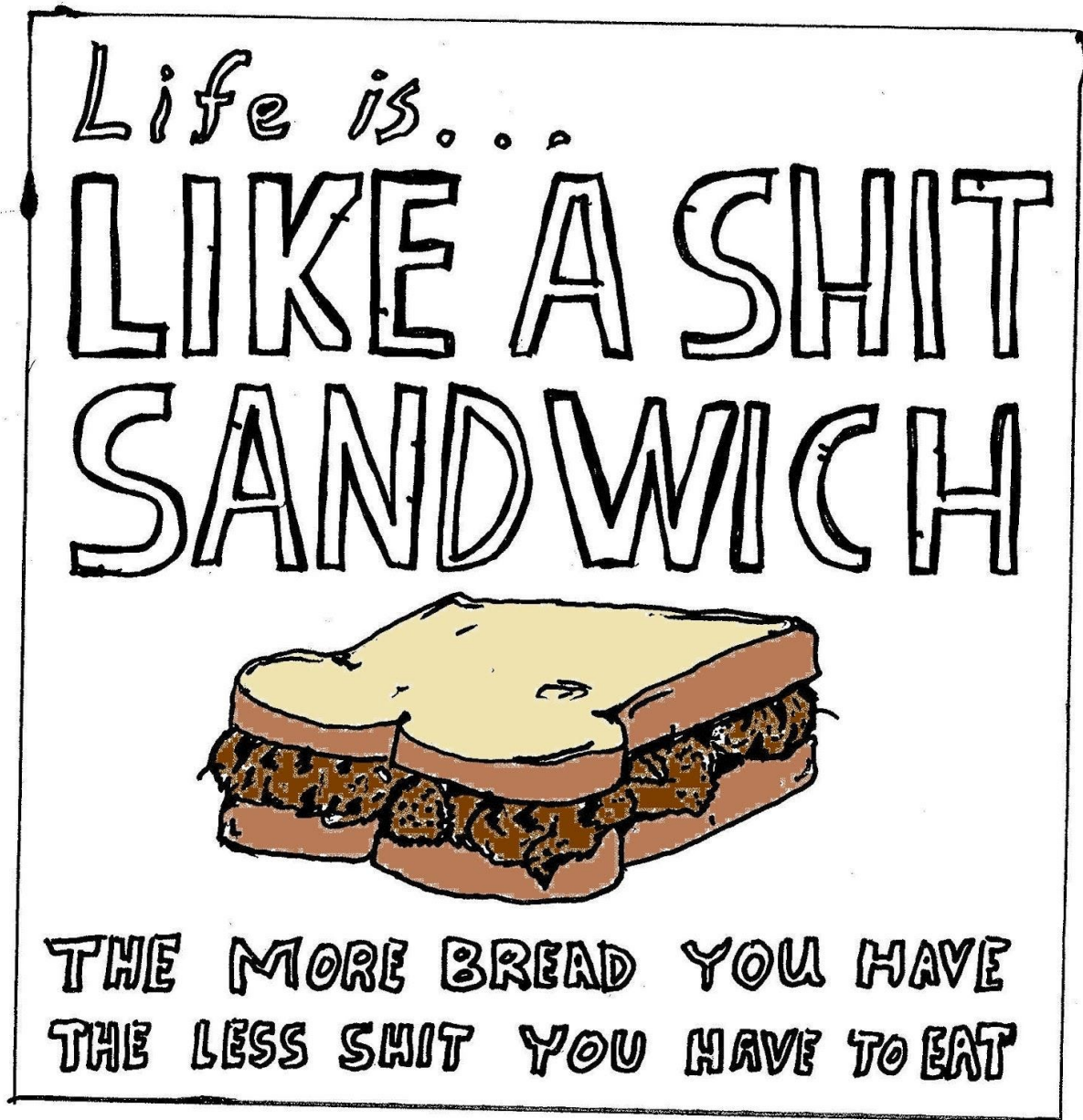


Figure 1.11: The “one true” model that intuitively comes to my mind. Substitute “bread” with the simplest possible test to provide you confidence. Unknown author. Found at <https://www.pinterest.se/pin/707839266404139008/>“><https://www.pinterest.se/pin/707839266404139008/>

All models are wrong, but some are useful.

— George E. P. Box

I'll keep the theory to a minimum here, as I'm writing this section to cater to any more junior developers who may want some framing of this entire thing.

Note that for an organization, or even a complex multi-part system, we might even want to—like Vladik Khononov writes in his 2021 book on Domain Driven Design—want to default to certain models based on the overall architecture of a particular system. So don't necessarily think of a given model as something that is good for *every* case.

Testing ice cream cone (unknown origin; God?)

The primeval model that has been passed down through aeons...

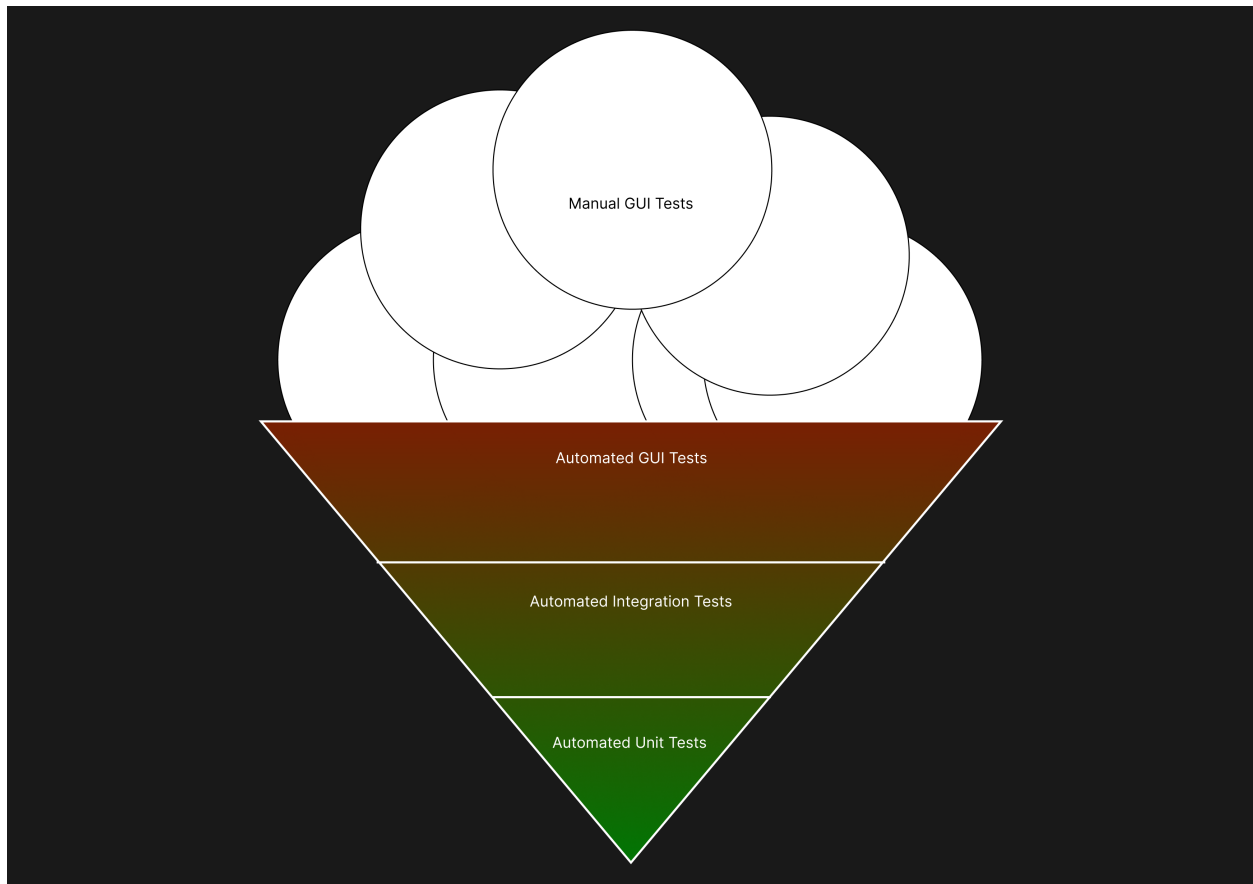


Figure 1.12: Ice cream is delicious – however the testing ice cream cone is a deception meant to lure low-tier consultants into the marketplace.

How about we start with *the one no one actually argues for*, but it seems like a lot of organizations pay good money to get: [The testing ice cream cone](#).

In this not-so-uncommon model, we get a small number of unit tests, a larger number of integration tests, a bunch of GUI-oriented tests, and finally, a whole dollop of manual GUI-oriented tests. While indeed this model contains a fair degree of automated tests, their relation is incorrect—by which I mean we write quantitatively more user interface-facing tests than code-facing tests. With the generous filling of manual tests, this is quite a toxic blend, perhaps mostly because this is an *actual model* (!) rather than just a mere circumstance.

In short, the problem here is that we focus on the least stable, furthest-from-the-code areas. The one single upside of this model is that it is absorbable for someone unfamiliar (or not professionally knowledgeable) with code, given that most of the tests happen “on top” of the running system. As I’ve already written several times, however, this “upside” becomes problematic as we now begin to invite non-developers into testing the quality of the system.

Testing pyramid (Mike Cohn)

The [testing pyramid](#), instead, is the saner, more well-functioning version of the ice cream cone, and kind of a classic notion of how to segment testing efforts in a meaningful way.

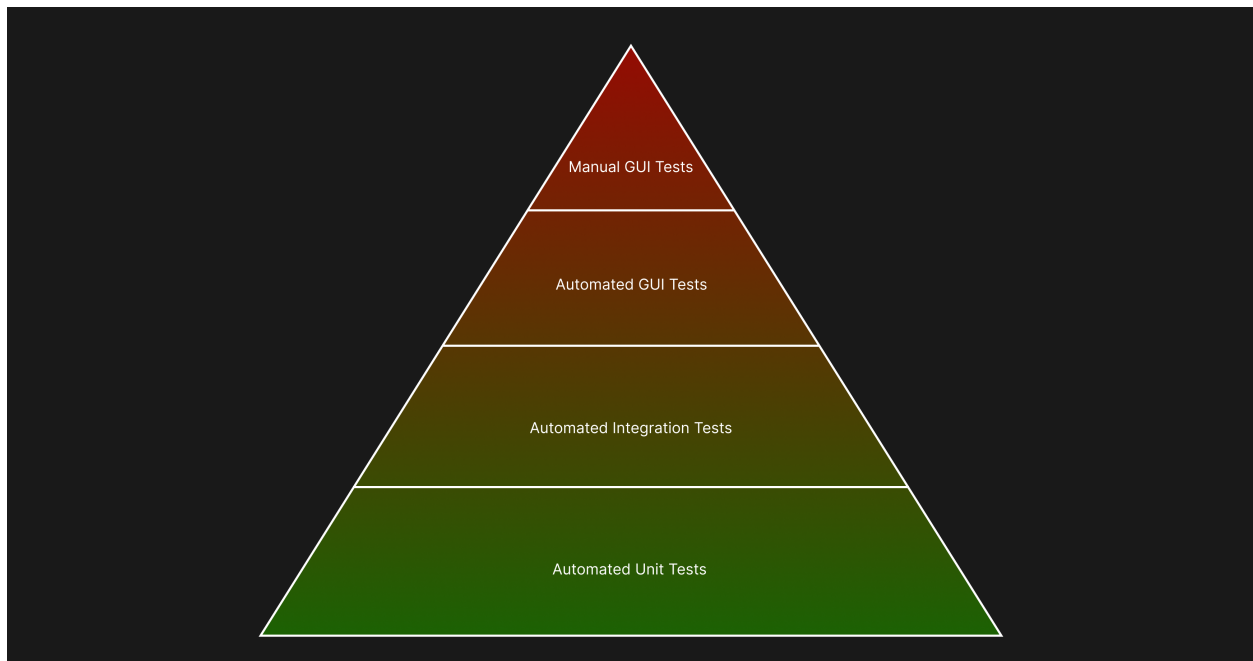


Figure 1.13: The inverse of the ice cream cone is a much more reasonable model.

It bears mentioning that the relative proportions are of course totally dynamic, but that the focus on automation is at least outspoken in this model and that the tolerance for manual testing is low but not necessarily non-existent.

This is the model I will still personally recommend in the majority of cases, at least as a starting point. It brings a sober attitude — which unfortunately is still for many organizations radically different than their current state. Seasoned developers may find it easy to get cushy and think of this as “old hat”, but the fact remains that you might struggle even getting this model working everywhere.

Consider starting here and adapting the relative proportions, favoring automation in general. Skew towards the simpler types, such as unit tests.

Information

One problem with the testing pyramid is that the proportions may optically *seem* to be similar in distribution. Mathematically we can clearly understand that this is not true, but the way they are depicted sometimes makes it feel like they are the same.

I would modify the diagram to place a *possible*, but not necessary end-to-end testing portion. Given that I am highly critical to its place in most systems it's not something that should be required at any rate, but accepted if needed. At any rate, the number of such tests will be strictly in the lowest range attainable.

Testing trophy (Kent C. Dodds)

A newer model is the [testing trophy](#), that as far as I know, was coined by Kent C. Dodds.

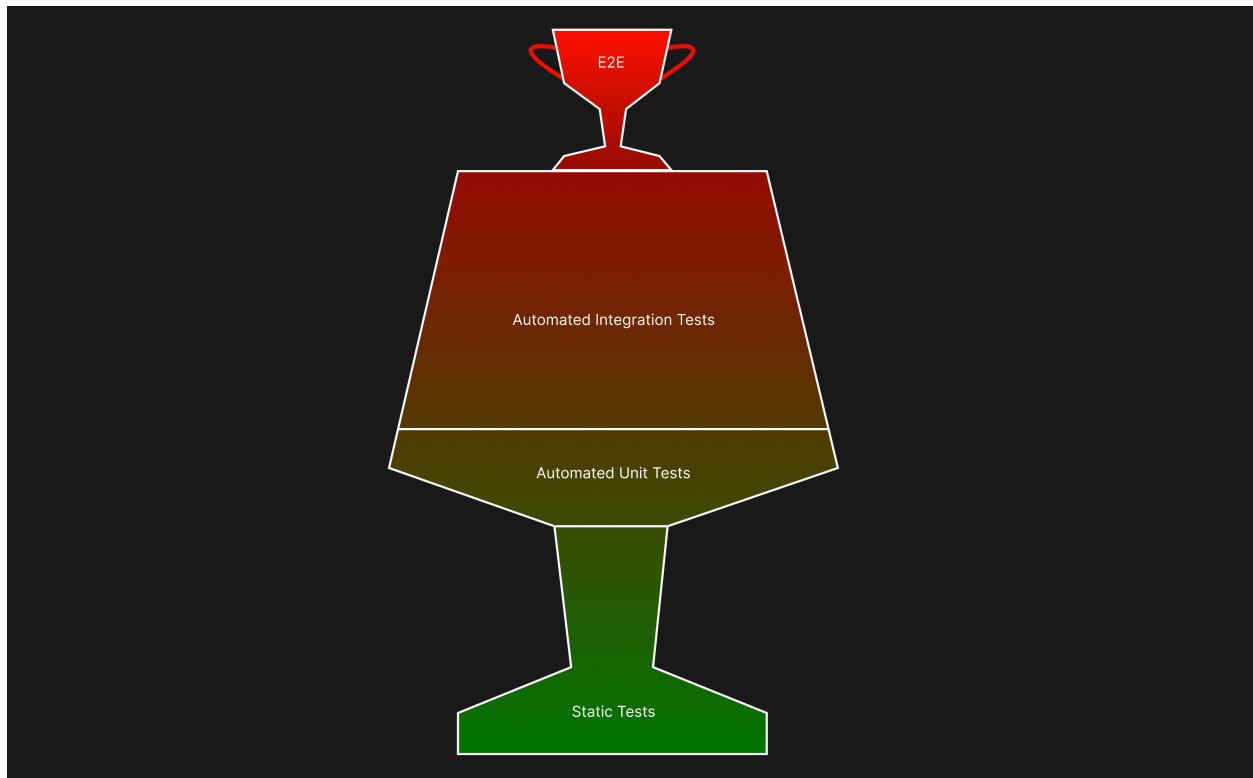


Figure 1.14: This model emphasizes proactive tooling (static tests) and puts systems in a sociable context, interacting with others.

This is a model that shares a common concern that many contemporary tech leaders make a big point about: Focus on integration testing. Given that most organizations today run fairly complex systems and most likely have some smattering of microservices, then it makes sense to imagine that integration testing would bubble up as the principal testing concept.

The testing trophy, as opposed to certain similar models such as the [testing honeycomb](#), include static code analysis and linting as the “base” of the trophy. I personally love this addition as it’s too easy to miss out on that part when discussing testing strategies; good static analysis should be a given today, using a mix of IDE-centric tooling and deeper, CI-side testing.

Success

While the other models don’t typically point out static tests, please do take static testing to heart as a requirement of a solid test approach, regardless of any spe-

cific model you might favor.

My recommended model?

If you ask me for components for a model, I'd answer:

- **A solid basis of static tests (everything that makes sense to use!)**: Run in IDE, as pre-commit hooks, in CI, and add any optional quality tooling running on pushed code. Examples of such tools are Codacy, SonarCloud, and DeepSource.
- **Automated unit tests (~80%)**: Checks the coherence, logic, and functionality of the system.
- **Automated integration tests (10%, if applicable)**: Tests particulars around the system being exposed as an API or whatever you build; this is more of infrastructural checking than any logical tests which would be covered by unit tests.
- **Key end-to-end flows (10%, if applicable)**: If it makes sense to verify that key UI flows work, then go ahead and test them! But make sure you are not testing the logical functionality of a back-end (third-party system; out-of-scope; already tested in the backend itself) or infrastructural concerns (covered by integration testing).

You will perhaps note that in my model, I allow for only a very slim selection of integration and end-to-end tests and that I even discourage them unless applicable and meaningful.

Choosing a model

As Mateusz Roth writes in [Why the test pyramid is a bullsh*t](#), it is wise to remember that **different models make sense for different types of systems**. A lean process for front-end apps with several end-to-end tests might be a decidedly different notion from a lean, unit test-leaning approach for a server-side library.

The model is just that, a conceptual model. It only gives your strategy and how you approach testing and where you place the relative effort – you can always change!

Confidence-based testing



The only test coverage goal that makes any sense is 100%. It's an asymptotic goal. You'll likely never get there. But you should never stop trying.

— [Robert C. Martin](#)

The actual, real value that you want to get from your testing is confidence. So regardless if you love or hate to write tests, confidence is the currency you should expect to be remunerated with. Nothing matters quite as much.

Solid quality engineering combined with rigorous, good tests ensure that you can, among many other things:

- Commit and deploy without a pull request
- Deploy before leaving on a Friday
- Be a human being, not always remembering all the details that may break
- More easily onboard newcomers (say, new hires or open-source contributors) to a project
- Clearly understand the functional requirements of your software

You gain confidence that unexpected behaviors and regressions are caught and don't slip out. You gain confidence in the code doing what you actually expect it to do. You have a relative guarantee that no (known) surprises should show up and stop you.

Confidence that is unverified, unverifiable, and unquantifiable borders on ignorance. You need some kind of proof and determinism to go with that sweet feeling. Sometimes it's easy enough that you know that a couple of quick cURL calls show the code is working. Maybe the thing you build can't fail in any spectacular way. Then that's enough. But for most of us, there's no valid way to get there without testing, because we build things that are non-trivial. Non-trivial things tend to have more failure modes than success modes.

Again, confidence is verifiable. Tests are not strictly equal to confidence; you can have untested code that you trust. But if I would build something with any degree of failure potential, then I would want it to promise the same reliability that any consumer product does—something as close as I can get to 100%. Something that works in the expected ways 100% of the time I can feel truly confident about. Because to be frank, it will still be able for it to fail. But likely not in the expected ways.

The opposite of confidence is of course **distrust**. If you distrust your code, you can never be quite sure that it works. If it works or doesn't, you never really know. Your code may break for people, set an end to a revenue stream for your company, stop you from distributing your band's new record, or worse, actively hurt someone who cannot get the medical attention they seek. The fallout of your problem space is certainly peculiar to your own circumstances. It may not even be that big of an issue!

However, without confidence in our code, and without being plain ignorant, we get anxious and worried, stressed and confused, instead of just fixing the problems, becoming proud of our achievements, and trusting the work we have done.

Confidence is for new coders *and for* seasoned software engineers. But many lose it along the way, which is sad to see. A confidence-based model is, in my opinion, the only realistic (and pragmatic) model that makes sense in the long run.

Choose high-quality automatable tools

Bad organizational structure has a clear relation to poor software quality. As nice as it would have been, good tooling on its own won't save you from poor leadership, poor requirements, poor code, poor tests, or even a bad day at work. What they *will* do, though, is make sure that whatever your conditions are they will be honored in a better way. In some sports, better tools mean it's easier to reach better results. Like other tools, a good test tool is reliable, easy to use, fast, unobtrusive, and provides valid and clear feedback.

First: How can you test your software?

Conventional testing is, to be fair, easy enough that a mediocre developer can build a relatively good testing framework in a matter of a few days, given that the software is of a typical nature.

You can ask yourself: Is your software possible to reach with an API or other technical interface? Do you have source code access (for example, it's your own software)? Is it only possible to test your software through a user interface – why?

Information

Even in the case of something like SAP business software, it does actually provide APIs to test functionality. However, the low general testing competence among people working with such software sometimes sparks false notions of the software not being able to test in a more technically-accurate, automatable fashion. Resist heavily when someone says it is not possible to automate testing!

Open source tooling or purchased?

Testing *isn't* about being a unique snowflake, about any type of exceptionalism, or about esoteric tools. In fact, these aspects are what I believe is off-putting to regular developers, when it comes to starting to do testing. Testing needs to be idiot-proof. And for this requirement we have lots of frameworks, both of the classic variety (Jasmine, Mocha,

Chai), the evolved school (Jest), minimalist options (Ava, the Node-native test runner), and when you bring in other testing types there are dozens of others names that whiz by. There is no lack of good tools in the open-source space. An additional pro of such tooling is that there is a community around the tool, its use, and best practices. Commercial tooling seldom has such factors in play for them.

Unless there is something exceptional going on, selling this type of tool at a premium is a fool's errand. Use open-source tools if at all possible.

In-context tests

More often than not, tests need to run in the same context as the code. That is to say, tests are part of the codebase. Separating test code from application code, for example in their own repositories, will make it incredibly inconvenient to test the functionality. I am hard-pressed to see how you could do any kind of white box testing (using the actual source code) if you were to do it this way.

Always prefer code-adjacent tools, testing the source code rather than “on top of” the running system.

Don't fall in love with your test tooling

As much as you might come to love your shiny tools, remember that technology progresses, time moves on, and trends shift. I am still seeing a lot of testers use “ancient” tools that also happen to use languages and runtimes that cannot be supported by modern pipelines (don't want/cannot mix Java 8 with Node 18, for example).

As far as possible, it's good to work with tests that are longer-term or even universal. In that regard, a bash script running `curl` will stand the test of time better than using a legacy test tool running a deprecated runtime will. While I haven't been in a situation like this, I do know that many businesses face significant problems when their tests become unmaintainable.

Writing good tests

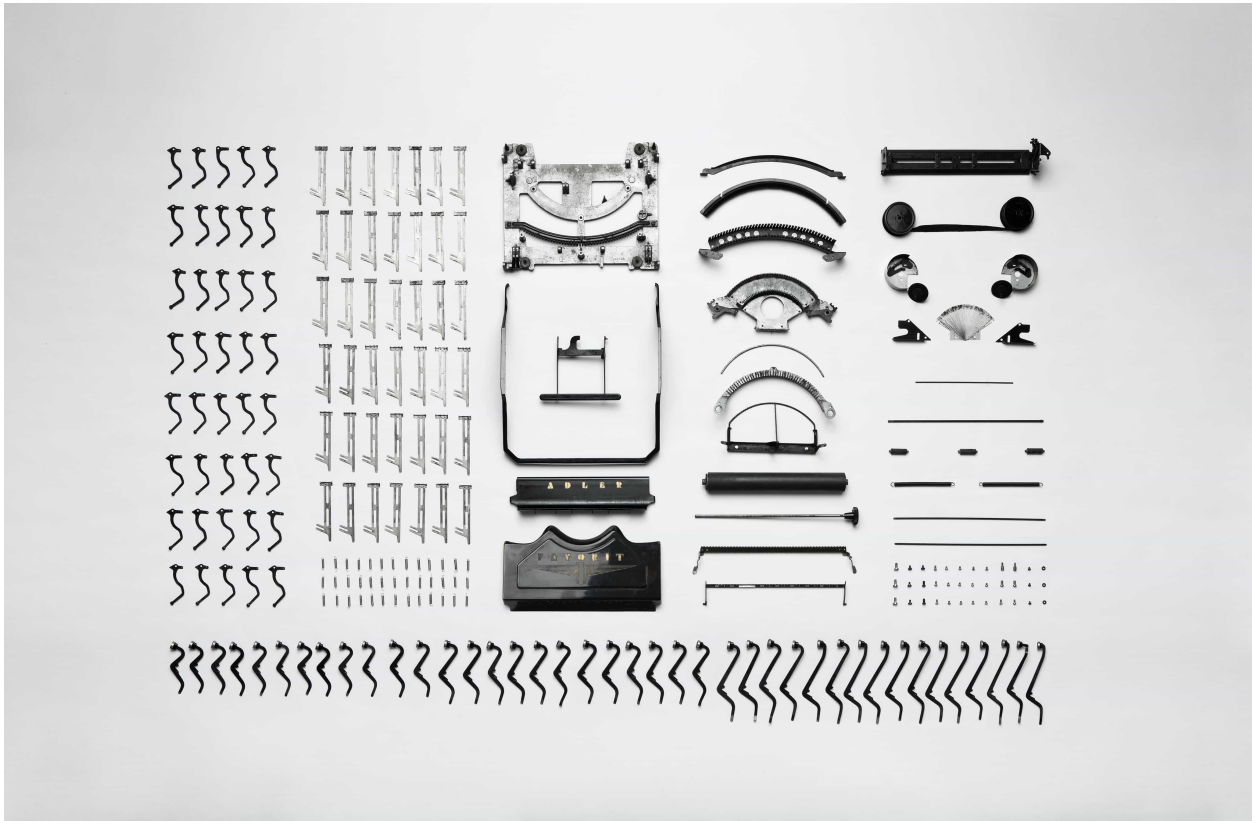


Figure 1.16: Good tests, like good tools, aren't magic; they are well-composed and well-designed.

People love debating what percentage of which type of tests to write, but it's a distraction. **Nearly zero teams write expressive tests that establish clear boundaries, run quickly & reliably, and only fail for useful reasons. Focus on that instead.**

— [Justin Searls](#)

Good tests and good code share a lot of similarities. Well, no surprise given that they both *are code*! If you're a half-decent developer this page should help fill in some of the gaps and questions you might have but take note that if you are already adept at writing semantic and clean code, then that should carry over really well to your tests too.

The FIRST principles of testing

One of the classic books on software engineering is Robert C. Martin's book [Clean Code](#). A well-known mnemonic, FIRST, came from the book which holds that a test should abide by the following principles:

1. **Fast.**
2. **Independent.**
3. **Repeatable.**
4. **Single assert per test.**
5. **Thorough.** Cover both positive and negative flows. There should be no unhandled exceptions. All failure conditions should be known and tested.

Information

See more related advice on TDD Manifesto's [A Clean Test](#).

This is pure rockstar magic and captures the essentials of good tests. If you closed the tab here and now, then already those principles should cover quite a lot. Crystal clear as they might seem, don't be surprised if you see tests that are far removed from this advice. Like chess, it's easy to learn but hard to master.

Good tests come easily given you already write good code. Sometimes, of course, some things are less easy to test, but don't accept brittleness. Even an ugly test, if needed, *should always work as expected*.

Programming in today's environment is less restricted by technology than by our cognitive and communicative abilities. Anything we cannot speak of in clear terms will always end up in sub-standard implementation (code and tests).

The one principle I'd argue that you can be more flexible with is the "single assert" principle, as it's not impossible you will want to assign several minor assertions as a single logical assertion. No one's going to arrest you, and just make sure you don't mess about with that everywhere but use it tactically.

Use the Arrange Act Assert (3 A's) format

The closest thing we have to a standard in (unit) testing is the [arrange-act-assert pattern](#). This is what you've seen since the very first demonstration:

```
const expected = 5; // Arrange
const result = add(2, 3); // Act

if (result === expected) console.log("Test passed!"); // Assert
else console.log("Test failed!");
```

Following this convention makes your tests readable and logically sound. It's quite possible that you'll have the odd occasion where there is more (messy?) setup that needs to be done, but as long as you follow this pattern and extract any bulky setup code from your test then you're pretty well set.

Always strive for “dumb tests”. The less complicated, the better.

Test for behavior, not for implementation

Google writes about a scenario where we replace the current implementation of a calculator class with a new one. What to do with the current, working, tests?

None of the existing tests should need to change since you only changed the code's implementation, but its user-facing behavior didn't change. In most cases, tests should focus on testing your code's public API, and your code's implementation details shouldn't need to be exposed to tests.

— Google Testing Blog: [Testing on the Toilet: Test Behavior, Not Implementation](#)

There's a transactional kind of beauty in a good test because we don't really care *how* the code did what it did, just that it did what we expected.

This is about having eyes on the *only* important thing: the behavior or response you get, not anything else. The test is a consumer of something (a function, method, class, service, system, whatever) and receives something. Can you evaluate the behavior? Can you see if it's correct? Yes? Good. You don't need more than that. Especially in a serverless

world where it's all messages and HTTP, that should come as a very convenient way of modeling your thinking.

Anecdotally I still hear about QA testers precisely testing for implementation...

Mock as little as possible and avoid specific tools for it as far as possible

Something I've also learned while working with people over the years is that there is sometimes a bigger reliance on tooling and frameworks to handle details like mocking. I find this to add complexity and dependencies for something that ideally should be able to be done in a manual and lightweight manner. There's a lot of reading in the area of [test doubles \(mocks, stubs, fakes, spies\)](#), particularly on mocks and the risks and issues that come with them.

Primarily I would advise creating “local” mock/fake implementations that you inject into your code to approximate the expected behavior. Prefer using composition to direct how you can test, over mocking.

There will be some more commentary on this later in this chapter.

Should I have one test per class/function/whatever?

Some smart people would say no.

Information

This article is really good and interesting, having a conversational type of format on [test contra-variance](#) and what you gain by not using the convention of a test file per class.

I would say: Write tests from the widest unit tests (use cases) first and then fill in additional tests (in separate files, of course) for each class/function/method that has code branches that are not covered by such wider “use case unit tests”. Don't forget to always

do negative testing as well!

Given that you probably have the possibility to throw some errors in most classes/functions then it's a realistic assumption that you will end up with individual test files for most of your code base. In those files, you should test the difference or delta, i.e. the "missing parts", not everything from scratch, once again. Using this approach means that you don't do excessive and redundant testing, but that you actually fill in bit-by-bit the missing code branches. In total, you will quickly and expediently move to better and better coverage.

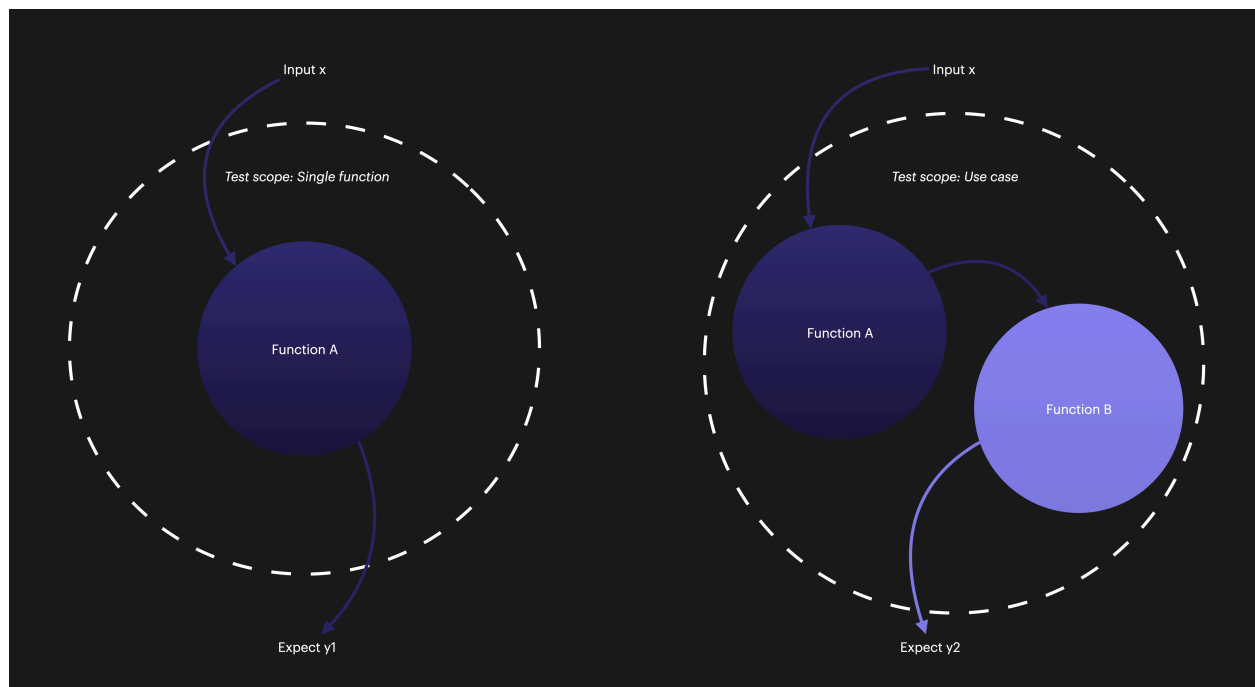


Figure 1.17: Save yourself effort and redundancy by writing tests for the outer scope of your system first.

The *farther out* we are (e.g. testing from the use-cases layer) the coarser-grained our understanding of the system is. That's why I always "fill in the gaps" where branch coverage is not full (or nearly full) in tests that exercise relevant functionality. Typically this would be something like this:

- Tests for use cases -> Behavior focused (general; almost like a unit test-flavored system test)
- Tests for functions (such as infrastructure utils) -> Behavior-focused (typical unit test scope)

- Tests for classes (public methods; refactor private methods to utility functions that can be individually tested, if relevant) -> Behavior focused

What is the test coverage to go for?

As we've seen in the Robert C. Martin quote previously, 100% should be the goal, whether or not we can reach it. Sometimes it's not practical or possible to get 100%, but anything significantly less than full coverage should make you ask what you are trying to get out of the testing in the first place.

Remember that even 100% test coverage does not rule out all logical, meaningful scenarios you might want to verify. So if I were a worse (but rather realistic) person maybe I'd set up the maxim:

100% test coverage is not the end. It's where we begin.

Sorry mate, if you expected another answer.

Information

While the following link concerns [Jest](#), I find that [this article is good enough to recommend anyway as it gives solid advice on how to actually reach full test coverage](#) and what to do about situations like “untestable” code lines. Highly recommended!

Cultivate a gut feeling for how much testing is “enough”

Even Kent Beck, the TDD and Extreme Programming guy himself, says [this](#).

Here's how my thinking (and feeling!) goes:

- Always document usage! Not documenting at least examples is inexcusable. Give API examples and anything needed for someone to test what you built, no matter how small. And how about you? Will you remember in 12 months, if you look at the code then? Nope. You won't.

- In very simple demos I don't do any tests at all. (There, I wrote it black-on-white). If the scope is very slim, internal or otherwise "safe", then just skip them.
- When you do test, test the things that can actually break, not every theoretical combination, or things you know will work (hairy proposition, but again, don't overdo testing). Your code should ideally behave predictably, which means test cases should not be unending.
- In small production projects, I might just do a small Node (or 'ts-node') script that runs functions/classes and verifies them on a functional level. No need for a big framework.
- If APIs are needed, I'll do kind of integration tests with a mock. There's a lot of nice tooling for this. From using real APIs with super basic [curl](#) to [node-fetch](#) or [superagent](#) to [mocking APIs with \(for example\) Jest](#), to [Mock Service Worker](#) or contract testing using [Quicktype](#)-generated schemas through [Pact](#).

Scoping the test

An important part of a test is to be precise; testing too much or too little is detrimental.



Figure 1.18: Like threading a needle, scoping a test might be picky but it's one and done once it's threaded.

Remember one of the basic rules of good testing: We should aim to only make a single assertion per test. In other words, tests need to be *precise*. One obvious way to gain precision is by minimizing the surface area of our tests. We can call the activity of deciding on the boundaries of a test as ***scoping our test***.

Scoping is not some exhaustive, esoteric process, it's simply about knowing “what does my test cover logically?” (not to be confused with test coverage!).

Terminology time:

- **Test:** The individual test and assertion you want to make.

- **Test suite:** A “family” or related logical collection of tests. For example, given an ATM software scenario, you might want to have a suite of tests for all positive/-working cases of withdrawing funds with credit cards.
- **Test scope:** The explicit or implicit “range” (scope) of what a test is responsible for in terms of covering and verifying truthfully.

Example with AVA

Without further ado, let’s look at how this might look in a unit test:

```
test("It should get slots", async (t) => {  
  const slots = await GetSlotsUseCase(dependencies);  
  t.deepEqual(slots, [{ something: "abc" }, { something: "xyz" }]);  
});
```

This particular example is from the [DDD example project](#) and uses [AVA](#), a very lean tool for unit testing. I like it a lot, as it is an unencumbered experience with fewer options and bells and whistles and it’s typically faster too when compared to a big unit testing framework like Jest.

The test is very clear and to the point. The use case is the “outermost shell” of our application in terms of unit tests, so quite a bit of code will be run under the hood once we run this test.

Information

AVA is somewhat less feature-rich compared to many frameworks, something I am not seeing as a problem in the majority of projects I work with. Do give it a try!

Scoping in AVA is just a question of placing all the relevant, related tests in a file (single or multiple files, as you like to organize them). Name the file what the test is scoped to, such as `GetSlots.test.ts`. That way, we can easily isolate and run selected tests/files if we want to test any certain features (or classes, functions...) of the application.

Information

Because my own work often requires only a relatively small number of tests, this does not become a major hassle, but sure, some test files can be a bit long if we collect a lot in a single file.

I always write both “positive” (happy flow) and “negative” (unhappy flow) tests and I tend to keep them in the same file. Their location is marked by a multiline comment such as:

```
/**
 * POSITIVE TESTS
 */
```

If there would ever come a time in which further segmentation would be needed, I’d consider breaking positive and negative tests into their own files, but I’d probably rather go for a fuller testing framework if that’d be the case.

Example with Jest

Another example is the following test from [Figmagic](#). This time we are using Jest, one of the most well-known and well-used tools in the Node/JavaScript/TypeScript world at the time of writing. It’s more full-featured than AVA for sure, and it also has a very nice describe construct that we can use.

```
describe("Failure cases", () => {
  test("It should throw an error if no argument is provided", () => {
    expect(() => {
      // @ts-ignore
      convertHexToRgba();
    }).toThrow();
  });

  test("It should throw an error if missing a single parameter", () => {
    expect(() => {
      // @ts-ignore
```

```
        convertHexToRgba(1, 1, 1);
    }).toThrow();
  });

describe("Success cases", () => {
  test("It should correctly return a CSS standard RGBA string", () => {
    expect(convertHexToRgba("#33ff00")).toBe(`rgba(51, 255, 0, 1)`);
  });
});
```

The describe feature is common among most unit testing frameworks, and it helps you to better express in the actual test file any scopes, such as failure/success cases, or to package tests by features within these blocks. **In Jest (and most such frameworks) you can use the describe feature as a way to express the scope of the test.** This makes it even easier to read, browse, and grok the tests, especially when there becomes many of them, and they might deal with unrelated use cases.

So there you have it. Scoping should be pretty clear to you by now, given that it boils down to other typical software engineering principles, such as being semantic, clearly organized, and so on.

Use the mechanisms you have available to you but don't forget to also write the shortest and most well-defined tests you can!

Prefer fast and reliable: On unit vs integration testing

Tests create a feedback loop that informs the developer whether the product is working or not. The ideal feedback loop has several properties:

- **It's fast.** No developer wants to wait hours or days to find out if their change works. Sometimes the change does not work - nobody is perfect - and the feedback loop needs to run multiple times. A faster feedback loop leads to faster fixes. If the loop is fast enough, developers may even run tests before checking in a change.
- **It's reliable.** No developer wants to spend hours debugging a test, only to find out it was a flaky test. Flaky tests reduce the developer's trust in the test, and as a result flaky tests are often ignored, even when they find real product issues.
- **It isolates failures.** To fix a bug, developers need to find the specific lines of code causing the bug. When a product contains millions of lines of codes, and the bug could be anywhere, it's like trying to find a needle in a haystack.

So how do we create that ideal feedback loop? By thinking smaller, not larger. [...]

With end-to-end tests, you have to wait: first for the entire product to be built, then for it to be deployed, and finally for all end-to-end tests to run. When the tests do run, flaky tests tend to be a fact of life. And even if a test finds a bug, that bug could be anywhere in the product.

Although end-to-end tests do a better job of simulating real user scenarios, this advantage quickly becomes outweighed by all the disadvantages of the end-to-end feedback loop.

— [Google Testing Blog: Just Say No to More End-to-End Tests](#)

In this chapter, we will relate unit tests and integration tests and I will argue for preferring unit tests over integration tests.

OK, so what are some properties of a unit test?

- A unit test is anything that does not reach out to other services, systems, or I/O.

- A unit test can test *any scope* within the same system: A complete use case or a particular error in a specific function. The consequence is that a unit test *is not tied 1:1 to a given class or function unless that's what you want*. They can be as exact as you want and need.
- To gain high levels of test coverage, you will have to write all wide (“acceptance” or use case) tests as well as fine-grained function/class-oriented tests.
- There should be no reason to mock a lot at all (if anything) since unit tests do not use other systems. It’s advisable to mock as little as possible since doing so adds complexity, frailty, and a decoupling of the test/mock and the actual implementation making it less confidence-building.

And integration tests...

- An integration test is anything that actually does use real services, systems, or I/O.
- Integration tests are “black boxed” higher-level tests that test for expected behavior, but do not add detailed feedback on its execution or “under the hood” operations.
- To be clear, integration tests use real infrastructure whereas unit tests do not require any at all.
- Question your need to test on infrastructure and drive towards unit tests until it doesn’t make any sense.
- Use real infrastructure if you need to actually see if something works.
- Side effects of integration tests may or may not be persisted, but they must always be controlled.

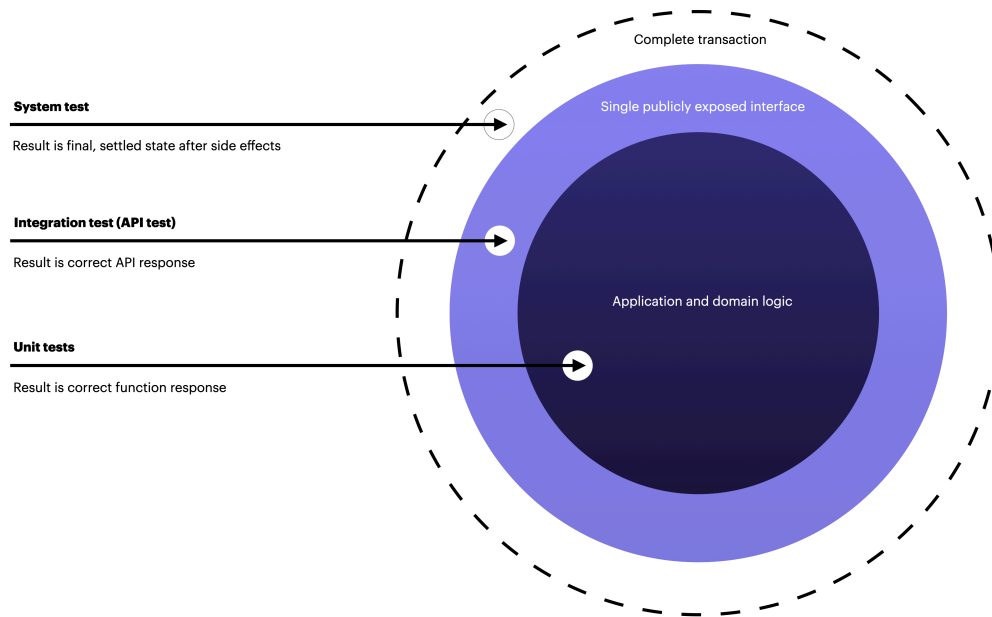


Figure 1.19: The further out we are, the more is in the scope of a test, yet, the less we know about what happens on the inside. Also, the further out we are from the code, the more expensive and slow it is to run our test.

Understand and question your goals with testing on infrastructure

What are your goals with integration testing, is it to use infrastructure to gain some confidence you are looking for? A part of your testing strategy and/or model should be to determine what types of tests and proportions you aim for. It seems that it's getting increasingly popular with integration testing in such models; however, you will remember I only attributed a small part to it, myself.

Some of the benefits of testing with actual infrastructure include:

- That it's closer to the “actual” performance and state.
- Feedback from such testing should be of a higher quality, being more truthful.
- It can smoke out errors in integration logic.
- It makes it possible to understand if our configuration is incorrect (such as not being able to send a message on an event bus because we don't have proper access rights

to it).

In fact, many of the above issues can be caught already with good unit tests and solid static analysis and types. What cannot be caught can be, to a large extent, substituted. How many integration errors are not actually your application-side errors?

With this said, *there is a need* for some integration testing, but try to push it to those points of your software where you can't meaningfully catch them with unit tests. I've found the best way to use integration tests as a small selection of very precise "acupuncture needles" to see that things hold together like expected, closer in spirit to smoke tests.

You need to think of *your goals* of testing with infrastructure, given that it's heavier, more expensive and time-consuming, and more of a hassle to test. Question and pressure-test your assumptions. Do you already have good unit test coverage? Do they cover the steps that produce data for outbound integrations? Are you validating, sanitizing, and cleanly handling input data?

As a developer (or architect) you should always aim for [supple design](#), and in keeping with [Occam's Razor](#), prefer the simplest answer. **In other words, prefer unit tests.** Even tech giants like Microsoft have been through this battle and have reaped significant benefits as a result:

A Microsoft team decided to replace their legacy test suites with modern, DevOps unit tests and a shift-left process. The team tracked progress across triweekly sprints, as shown in the following graph. The graph covers sprints 78-120, which represents 42 sprints over 126 weeks, or about two and half years of effort.

The team started at 27K legacy tests in sprint 78, and reached zero legacy tests at S120. A set of L0 and L1 unit tests replaced most of the old functional tests. New L2 tests replaced some of the tests, and many of the old tests were deleted.

— [Microsoft: Shift testing left with unit tests](#)

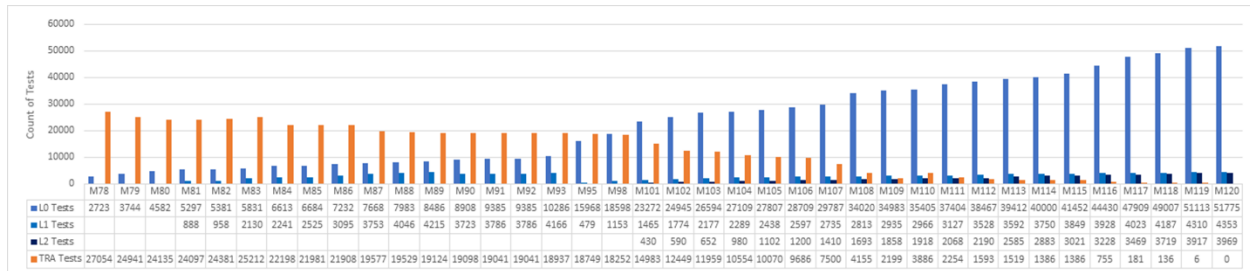


Figure 1.20: From <https://learn.microsoft.com/en-us/devops/develop/shift-left-make-testing-fast-reliable>

The Microsoft team drove down test time and increased the reliability of tests over the two years it took to migrate their significant collection of non-unit tests. Notice how few non-unit tests persisted once their transformation phase was over?

Information

Overall, you'll find plenty of advice about maximizing the use of unit tests in the literature as well as in good online sources. For more, see the Reference and resources section at the end of this book.

Boil down the surface area of your tests — I am sure you will see that integration tests are less needed than you thought they were at first.

Handle and mock side effects

Concerns around side effects, such as polluting a production database with test data or calling rate-limited and pay-by-use third-party services come up pretty swiftly in conversations around testing and test practices. All of these are valid concerns. The real question isn't "can we test this thing?", it's "how do we minimize unintended side effects?".

A *side effect* is that something somewhere happened or changed. An **intended side effect** can be that a function should write a file to disk. An **unintended side effect** could be that calling a function also sends an event to a hardcoded event bus residing in a production environment; this event may not be part of a test's scope, but it is part of the deeper functionality of the system under test. In other words, it's "needed" for our functionality, but is not something our test cares about. This is solid and sound advice, given that [we don't want to test I/O](#) and our [test scope should be limited to their correct boundaries](#).

Information

Some further reading on the subject includes:

- [Wikipedia: Side effect \(computer science\)](#)
- [What is a "side effect?"](#)
- [The Not-So-Scary Guide to Functional Programming](#)

We will touch on a whole spectrum of ways to handle mocks and side effects:

- Letting the code be test environment aware
- Abstractions and dependency injection (Test doubles)
- Consumer side mocks
- Provider side mocks
- Ephemeral (temporary) environments
- Tagged data

Temporary environments

A spin on the “test environment” idea would be to have a temporary environment and make sure your code is hooked up in a way that uses only infrastructure linked to that particular one. This way, when the tests are over, no data is retained and there is no leakage outside of the containment area. However, this will be slower, and depending on your level of determination, you should probably keep this environment only for your service.

While this sounds like an easy solution, you will have to model your code to use environment parity which might not be the way you have resolved relations, paths, and URLs before.

Tagged data

This solution uses the regular production or shared environment to run in, but you tag data (messages, stored items, etc.) with some type of marker or information that your systems will know means it’s test data and that it can be removed. Also, you should ensure such data is never read back by any other system.

This solution, again, sounds pretty easy to implement but will require *knowledge* of this whole mechanism elsewhere, creating undue risk.

Now for the better options.

Mocking

In object-oriented programming, mock objects are simulated objects that mimic the behaviour of real objects in controlled ways [...]

— Wikipedia: [Mock object](#)

This seems like a good idea (and is, to some extent), but you’ll have to be careful with mocks. Mocking may create very real coupling issues and harm later refactoring. As Philippe Bourgau writes in [Careless Mocking Considered Harmful](#):

The test initialization code was getting longer and longer, as it included a lot of mock setup. This made the tests more complex and less readable. It also made them unreliable, as it was not rare for all my unit tests to pass while the system was not working. I was taking the habit of running my end to end test more and more often. I was also losing a lot of time maintaining the mock setup code in line with the real classes. Mocks also tricked me into the bad practice of keeping a 1 to 1 mapping between code and test files. That again increased my maintenance burden when moving code from one file to another.

It reached a point where I could not take it anymore.

A nod to his title, we don't want to be careless.

Let's say you have built a pizza delivery service and you want to test the functionality for receiving an order. You have a dependency in your code on both some persistence infrastructure (a database) and messaging infrastructure (a pub/sub topic).

Are you now required to deploy this whole shebang to some test environment and then run integration tests on it to see if it works? No, you don't.

Given that the infrastructure is something built by someone else (for example your cloud hosting company or if it's some infra that another team manages) then **it is effectively out of scope for your tests**. But how do you tell your code this?

Dependency injection

We can use dependency injection (to pass in for example our database/message bus instance into the higher-level object) and/or differentiate between [abstractions and concretions](#) to enable a composable, classic way to handle this. Our controller or handler functionality could therefore be tested with a faked or stubbed version of our persistence infrastructure, and outward there would be no change. As far as possible, this is what I would recommend you do. You approximate the expected behavior and since the boundary of the test does not concern the database (etc.) then you are well within your remit to set the boundary here.

```
interface Database {  
    store(pizza: string): void;  
}
```

```
class LocalDatabase implements Database {
    store(pizza: string) {
        console.log(`Storing order for a ${pizza}`)
    }
}

interface MessageBus {
    send(message: string): void;
}

class LocalMessageBus implements MessageBus {
    send(message: string) {
        console.log(`Sending message: ${message}`)
    }
}

class PizzaOrderService {
    private readonly database: Database;
    private readonly messageBus: MessageBus;

    constructor(database: Database, messageBus: MessageBus) {
        this.database = database;
        this.messageBus = messageBus;
    }

    public order(pizza: string) {
        this.database.store(pizza);
        this.messageBus.send(`Order received for a ${pizza}`);
    }
}

const database = new LocalDatabase();
const messageBus = new LocalMessageBus();

// Injecting dependencies, service doesn't care if it's a mock or real
const pizzaOrderService = new PizzaOrderService(database, messageBus);

// Coded implementation isn't changed in any way
pizzaOrderService.order('Margherita');
```

That's a pretty conventional and completely frameworkless way to deal with that problem. This method could be dubbed the *consumer side mock*, since you had to take the time on

your end to create this solution, despite the dependencies being built by someone else.

Provider-side mocks

This isn't something I have any real experience with, but anecdotally I know of dependencies/services that do provide mocks *to you as a user* that you can use. Regardless of the level of fantasy here, that's definitely an interesting idea but don't expect to see it used a lot.

Being “test-aware” and leaking test details to code

What if you want to test some implementation logic on the code that wraps the infrastructure? One way I deal with this, and this isn't very orthodox, is to actually check whether this is running in a test or not.

```
class RealMessageBus implements MessageBus {
  send(message: string) {
    // Do lots of things here that we actually want to test, then... (↔
    ↔ indicative example)
    if (process.env.NODE_ENV !== 'test') messageBusDependency.↔
    ↔ emitMessage(`The actual message that goes to a real piece of ↔
    ↔ infrastructure`);
  }
}
```

While this specific solution is particular to Node, the general idea holds. I've found it very common, despite good clean code and separation of it, that we might want to test stuff on an implementation, except of course we don't want it to persist anything or start creating side effects.

You'll also, in the case of the database mock/fake, want to return some data back.

I've never had this do something it shouldn't but I also know that we are stretching the rules and principles a bit here. Given the method's effectiveness, the big drawback is that we leak some of the test structure into your code. If you can accept this trade-off, then it's a very easy and pretty safe way to really test *all of it* minus the bad bits (side effects).

Libraries when mocking is actually useful

It pains me to use a mocking library, but if you have something that actually works, it's worth using. For me, the clearest use of library-provided mocking is for networking and overall just getting a truthful sense of how a piece of third-party infrastructure works. I'm very hesitant to use it for more detailed work, rather it's mostly to ensure I leak less of the testing logic to my implementations.

A library that I've had good use of is [MSW](#) with its network mocking capabilities (fetching data), so one can effectively test network interactions (responses/requests) in a very simple way because these can be more painful and boilerplate'y to mock.

While *it is a mocking library*, I've found [aws-sdk-mock](#) to be a reliable and powerful way to mock out AWS infrastructure.

You should be able to mock manually or automatically in most testing frameworks. If you use Jest, there is [jest.mock\(\)](#).

Test environments

what if... the best “staging area” was just running in prod, but for corp users only? or for your free tier only? or for an opt-in set of users only?

— [Charity Majors](#)

Get ready for what might be some controversial points, if not for you, then maybe for your team or organization!

A commonly held orthodoxy is that you need a test environment – an environment that acts as a stand-in for running your tests, so that any spill-over effects from your testing, and potential failures, won’t affect your actual, real users. Testing environments are supposed to root problems that we can’t see locally. It’s a solid idea, but one I find wherever I’ve been, to be fundamentally dysfunctional for some of the same reasons why separating developers and QA is a misleading idea.

Let us start by unpacking three underlying assumptions: *impact*, *location*, and *longevity*.

What is the impact of a caught, or uncaught, issue?

Not every bug or issue is world-breaking. Do you have clear and known ways to measure and prioritize the scope of an issue?

Finding bugs in testing is a good thing, but not having any issues at all is of course even better. By aggressively pushing left on tests, making developers write them to a high coverage and quality bar, running them often, and generally being mature about our testing, we are mitigating the *escape rate* of issues that others will experience.

However, not every issue will or can be caught up-front, because the total surface of your solution is not only the code you wrote. Shifting left on tests should reduce, but cannot totally remove, the surface area of potential issues. Using a test environment offers us a framework to detect these lesser-known issues. Similarly to system tests and integration tests, however, using a full test environment for the class of issues we already know about makes no sense, adding only complexity, cost, and unnecessary procedure.

The impact and class of an uncaught issue should help guide somewhat whether a test environment, in any form, is needed.

Where do we run the tests? Splitting either by hardware or software

The testing environment concept is often highly tied up with a notion of being a separate environment in terms of hardware, meaning the code under test is deployed to some other piece of hardware than where it will run for any real users. Hence, we might call this a *hardware-segregated environment*. The location is different from where your real code will run.

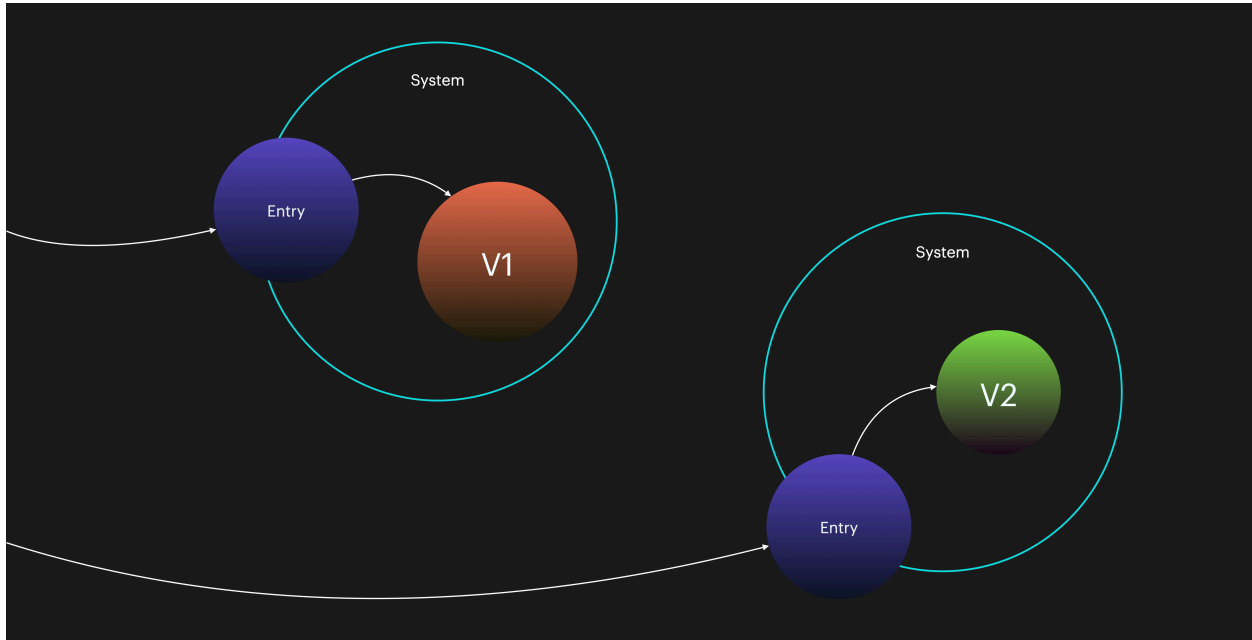


Figure 1.21: Typical hardware split. Each URL goes to a completely different system and set of circumstances.

This is what ultimately ends up being your `staging.org.com` and `prod.org.com` fully-qualified DNS names, and is a notion grounded in older times when doing redirects or traffic shifting was either not possible or it was hard.

Information

Some tooling, like Cloud Run, [can handle such things without any additional networking configuration](#).

Another option would be to run tests on the exact same hardware but distinguish the testing code by software means, for example by letting [feature flags](#) and/or in combination

with identifying HTTP headers and [branching by abstraction](#) in the code decide that you will run against the new, testable code.

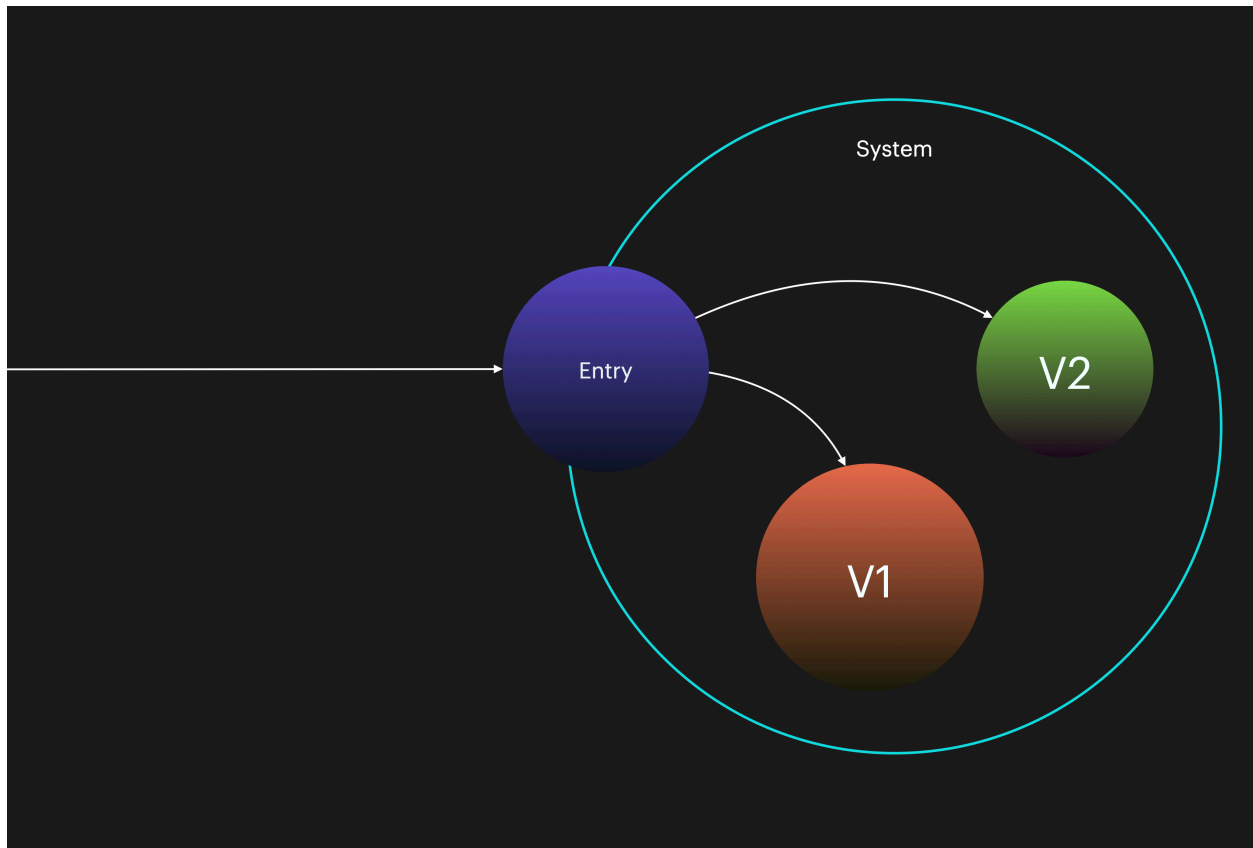


Figure 1.22: The application is deployed on a single piece of hardware and dynamically uses relevant code paths.

This is now a *software-segregated* test environment instead. In this reality, will only therefore only use the `prod.org.com` DNS name.

Information

Ideas for how to get the correct identity and feature set:

- Explicitly passed `Authorization` header and/or OAuth token, etc.
- Custom headers, like `X-Client-Version`.
- Presence of automatic edge-side headers and metadata; see for example [Cloudflare](#)

What do you think you are gaining from testing on separate infrastructure?

Given that it's neither more truthful, correct, faster, cheaper, easier, or better scaling, then why do we still keep reverting to this notion of the separate test environment?

One technical reason might be **infrastructure performance reasons**. Given your context this may be valid: Say you run a very small database instance for your production traffic (which is overall light) but have aggressive testing protocols. Hammering the database could therefore realistically worsen the user experience. It's impossible for anyone, let alone me writing an e-book, to say what is the *right thing for you to do* but I can point out a few factors you should consider:

- **Prioritize quality:** What if your testing efforts would emphasize quality over quantity? There is no correct number for tests, of course. Also, if we rebalance the types of tests we want to conduct to use more unit tests, then as a consequence of that, we also want to mock using real infrastructure, lightening the load on the infra.
- **Balance effort better:** What is the balance between actual use and the rigor of testing? In the above scenario, there was low traffic but an inordinate amount of heavy testing traffic. The balance doesn't seem right given the effort of testing something that is only lightly used.
- **Infra is under-provisioned:** The infrastructural setup might need to improve given that you can't even take a slight bit of increased load without it causing heart palpitations. Consider auto-scaling alternatives or even serverless options for future improvements.

So far we've seen that it's not impossible to put the performance argument to rest. In your common non-MANGA scenario it seems to rest on a broken strategy, rather than being a valid argument.

There could also be **compliance reasons** for why you would rather not test on the same infrastructure. It might be well worth discussing with your auditors if they are not OK with same-infra testing if you fulfill the same separation of data access and other such factors they might require of you. I've been in such conversations and auditors will typically open up for this if there are other guarantees in place. Don't mix up the auditor's job with yours! Target the wanted outcomes rather than being technology-centric.

Don't lose sight of the explosion in complexity when you run traditional hardware environment splitting:

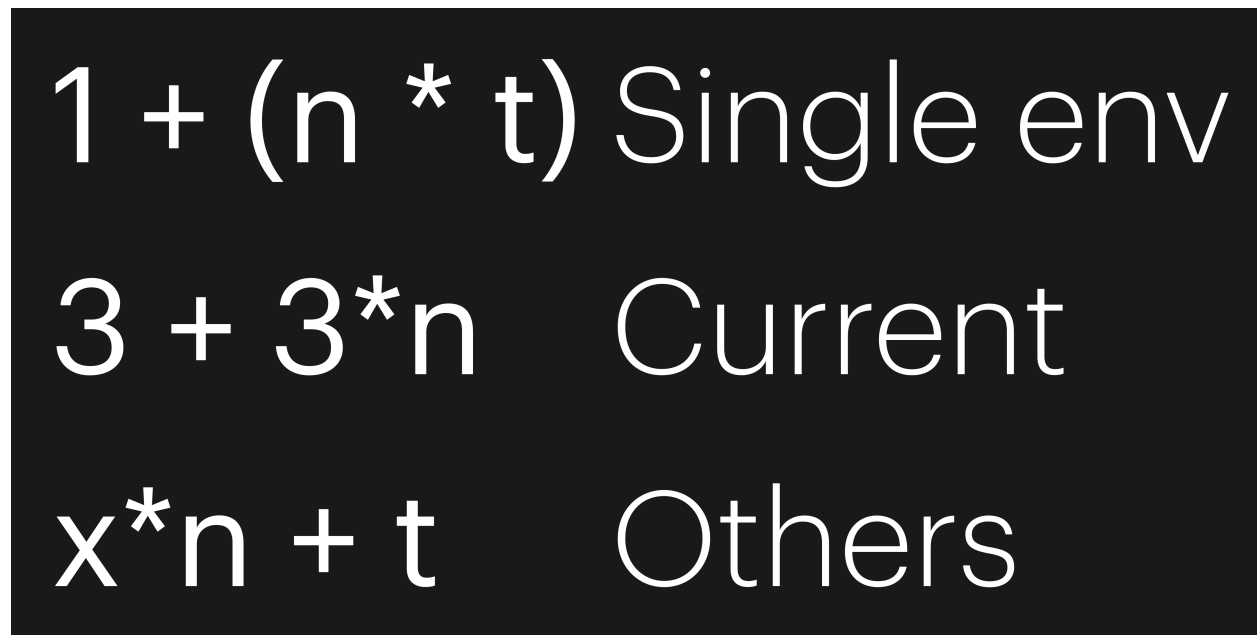

$$\begin{array}{ll} 1 + (n * t) & \text{Single env} \\ 3 + 3 * n & \text{Current} \\ x * n + t & \text{Others} \end{array}$$

Figure 1.23: Yeah, I was never the math guy, but that would be my take on a formula. The “Current” refers to the current norm of DEV, STAGING, and PROD. The worst thing? You still have no control over anything outside of your own system!

Running a single, or at least fewer permanent environments, is a really good idea.

Use infrastructure-as-code to keep environments exactly as configured and as similar to each other as possible

Another major testing problem that many developers face, is having flaky environments: something that worked in X no longer works similarly in Y. It's not uncommon that this is especially common in IT landscapes where we lack automation for deploying and maintaining the infrastructure.

Terraform, Serverless Framework, CDK, and many other tools allow specifying, configuring, and deploying infrastructure with such automation capabilities. As far as possible, migrate or at least start using tools like these to deliver infrastructure “on tap”, without drift, reducing the number of those situations. These tools make it much easier to ensure that an environment is strictly configured the same as another environment.

Regardless of your stance on software or hardware-segregated code, using IAC will make your life easier and give meaningful improvements on the stability of any environment, testing or otherwise.

Set up unique test environments for each system

Instead of sharing an environment, have a test environment that is unique to each system, minimizing the blast radius of issues, which better separates systems, as well as communicates that each system has its own lifecycle rather than some implied shared lifecycle. Allowing another system (or client/customer) to stall development and releases is anathema to agile development.

Benefit from the upsides of using ephemeral environments

We have started to look at the *location* aspect of environments—now we will turn to the *longevity* aspect.

Information

A **long-lived shared environment** in this context means a classic *long-lived* environment, typically something like DEV, STAGING, QA, TEST, and PROD used by all products that share that lifecycle. **Ephemeral** (temporary) in this context means a short-lived environment. It may be *actually short-lived* in which case all of the environment's infrastructure is created and destroyed as a direct consequence of, for example, testing.

It may also be handled in a way in which the testing infrastructure is uniquely owned and retained over time (though never shared across products!) but is never exposed for integrated co-dependent testing as is typical for static testing environments.

Information

We will assume IAC (infrastructure-as-code) tools like CDK, Terraform, or Pulumi are being used in both scenarios. We will, likewise, assume that two different team setups may be handling the respective solution (see the table).

Below, I will outline some of the properties of long-lived vs short-lived environments and how they stack up against each other.

Effect	Long-lived shared non-production environment	Ephemeral non-production environment
Ease of deployment	-	-
For Ops/Infra/Platform teams	More complex	N/A
For application/product teams	Somewhat complex	Easy
Ease of managing environments (drift, configuration etc.)		
For Ops/Infra/Platform teams	More complex	N/A
For application/product teams	Low to medium complexity	Easy
Ease of use		
For application/product teams	Easy, segmented on the DNS (URL) level	Easy, segmented by dynamic factor (headers, identity, etc.)
Ease of testing	Easy	Easy
Confidence of testing	Variable	High
Opportunities		
Supports canary releases?	Yes	Yes
Supports fast release cadence?	Yes	Yes
Enables independent deployability?	Yes, but can be hijacked	Yes
Offers more lifecycles than the fixed, provided ones?	No, not without special treatment	Yes
Enables high agility?	Not if dependent on Ops/Infra/Platform	Yes
Risks		
Inadvertent side effects if mismatching System A and System B environments?	Yes, may occur	Yes, may occur

Figure 1.24: Comparison table

One of the reasons you'll see that my assessment shows ephemeral environments being

“better” and easier to work with, is because they have **lower natural complexity**. Why lower complexity?

- You don’t have to park multiple applications and sets of data and/or other aspects in the same logical area.
- You ensure that no one can try to logically map or match environments and lifecycles of applications onto each other.
- A smaller surface area means less risk.
- Sharing an environment creates significant risk.
- The logical overhead of managing multiple applications in a given environment is often unacceptably high.

Consider using only short-lived environments. You are not stuck with DEV, STAGING↔↔, PROD, QA, or any of these staples. Short-lived infrastructure means faster and safer tests and reaching PROD in a better state.

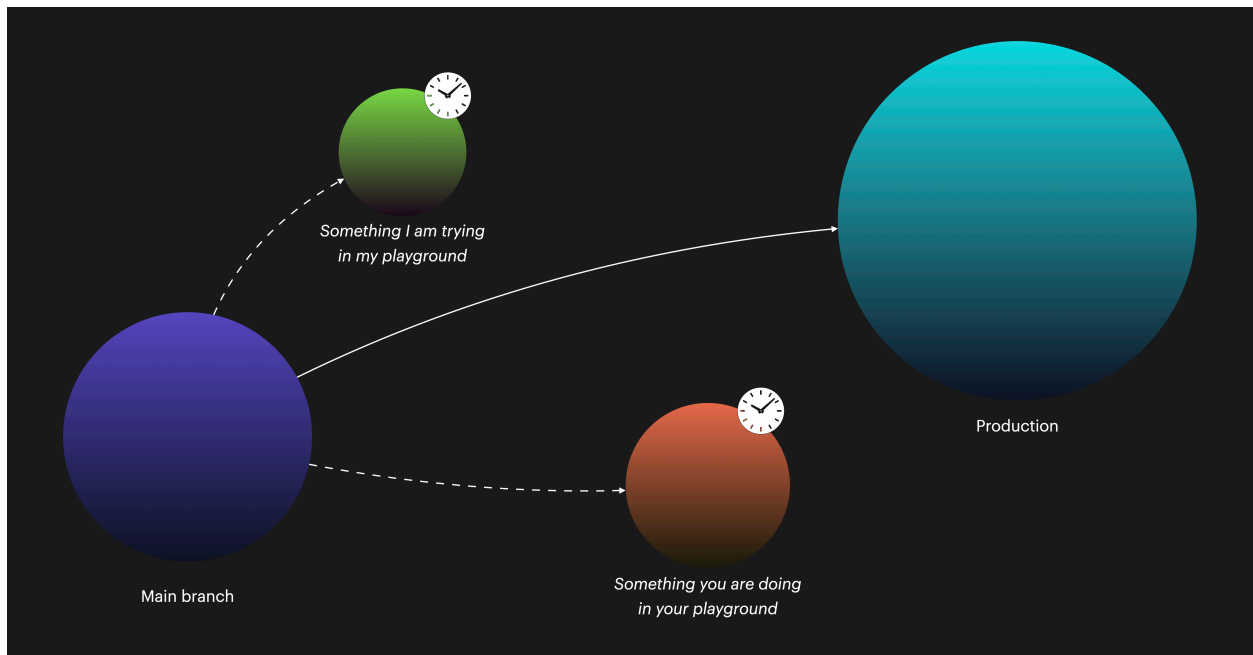


Figure 1.25: Environment lifecycle

In closing

I personally will no longer vouch for a DEV > STAGING > PROD setup (well, unless there is a good reason!), having seen this fail too many times. It's a poor model that is too hard, too expensive, and too broad to be of good use. It also very unhappily forces application segregation to the hardware level, which while mitigated to a great degree by serverless, is still a problem in terms of complexity. Using that type of setup also adds a lot of baggage to applications, as they now need to handle switching between environments. Maybe most depressingly, it sends the signal that a modern IT landscape—with many systems, possibly many microservices, all with their own DevOps teams and their own lifecycles—can have a static parity and stable interaction by means of their deployment regime. This is just not the case in a complex environment!

So if you ask me for the simplest, quickest, truest way to handle a test environment: Use short-lived environments where *you only test the things that can be meaningfully caught and reproduced in such an environment*. It must not substitute for the lower-level unit tests or any other known problem space.

Testing in production

We hear “testing in production” and think lack of caution, carelessness, poor engineering.

But in reality, every deploy is a test. In production. **Every user doing something to your site is a test in production.** Increasing scale and changing traffic patterns are a test. In production.

— Charity Majors, *Testing in Production: Why You Should Never Stop Doing It*

It’s actually a really intuitive notion. Given that tests are performed on something “known” and often in isolation (despite any misinformed ideas about permanent staging/test/QA environments), they aren’t very good at exposing complex issues. Tests are most often intended to prove that a known behavior performs as expected.

But testing, and monitoring, are also about learning about our issues—things we might not know about. In the last chapter, we looked at approaches for selecting a test environment strategy. It should have been clear that I’ve tried to push you in the direction of more often pushing code, being better tested in isolation, into your actual production environment rather than using pre-production environments.

Given that tests are something that verifies expected functionality, then these run at a single point in time, which is ideally several times between my code being written up until it hits the CI environment and is tested there. **Most of the conventional test types cover the known-known parts of our problem space.** But there is a lot more that could happen...

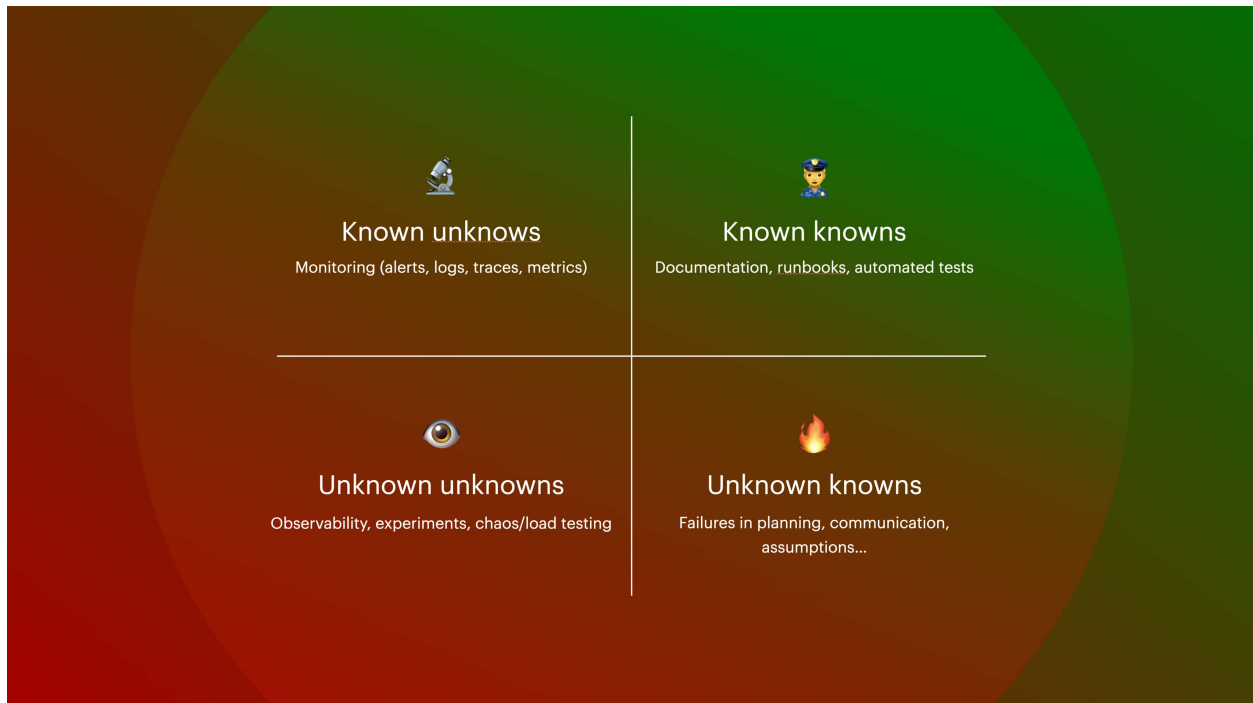


Figure 1.26: How developers can apply correct types of tools and methods for various problem spaces in testing and quality.

Conventional, “local” tests such as unit tests and (inbound, self-targeting) integration tests, are run close to the code and emit errors or reports that typically a developer will need to decode. The benefit is that these results are exact and actionable. The drawback is that *they are not run after* the deployment itself, while the code is in production, facing actual traffic and running alongside any number of other services.

A problem is not necessarily always of a class that can be logically caught, it can also be cascading failures, infrastructural failures, or a third party that has updated its behavior in a problematic way since you last deployed. Some problems can’t be picked up by testing, testing at the point of deployment, and certainly not only when this is done seldom (a few times per month or even more rarely). To test in production, a part of the secret sauce is to monitor or otherwise have an updated, informed view of the current state of our system.

Remember the temporal divergence between testing and monitoring, and how observability relates to them:

- **Testing** verifies known behaviors at the *point of testing*.

- **Monitoring** gives you eyeballs on the continued state of your software.
- **Observability** is the property that describes whether or not we can know what happens in your code.

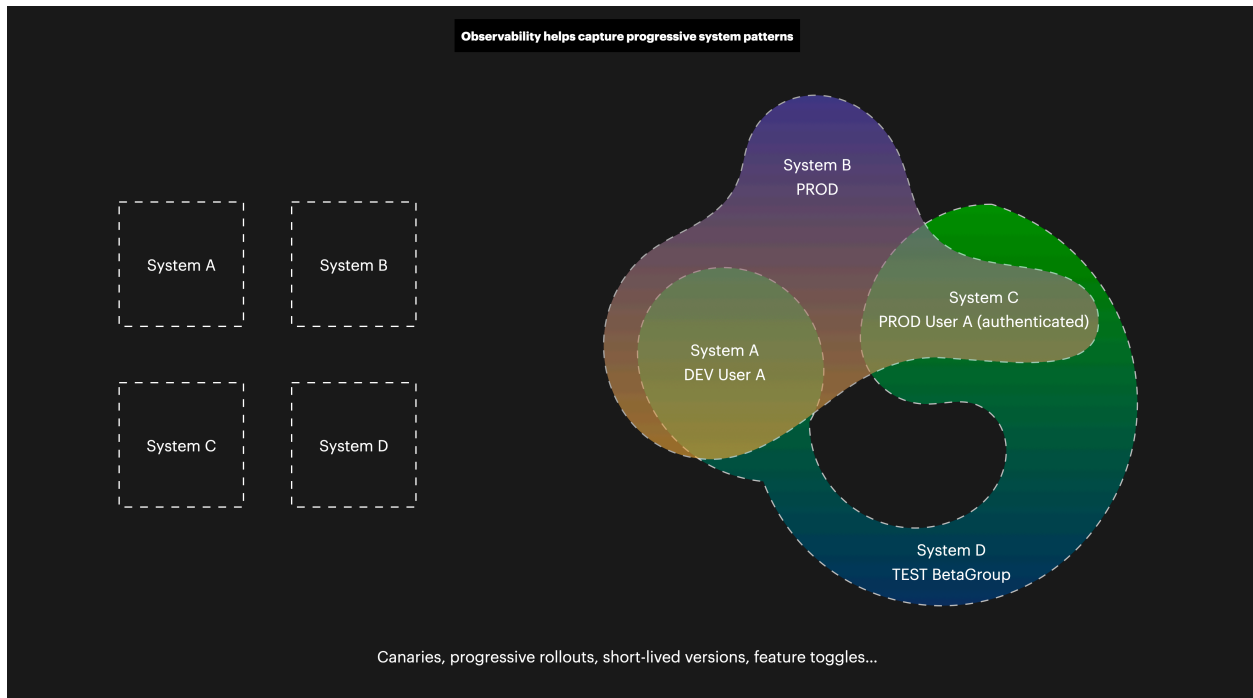


Figure 1.27: Life in a distributed landscape is not like those square boxes; rather, it's a lot more like on the right. We need to make our systems observable if we are to understand how they truly behave—something we can't forget just because we are writing tests before the fact.

One way to understand the role of testing in production is to accept that we need better ways to understand our failures, also from the temporal perspective. And like with Agile overall, shortening the feedback loop is a great way to start that journey. Doing production testing in a considered manner is smart, as instead of *ignoring* all the rich things that happen with your users in your systems, you can do something productive with this information.

Everything fails, all the time.

— Werner Vogels, Amazon CTO

Every problem or failure has a cost. Make the most of that payment! Cherish the opportunity.

If we've managed to create an agile, fast-learning, left-shifted testing culture, and we've decomplexified our testing and architecture, then learning from production should not be a problem. It's more of a cultural change once you are at that point. The fear many seem to have is that production is messy, irrational, illogical, and to be feared. Given that you prune your garden, as it were, the number of those odd things in production should decrease and you can gradually control it better and make positive changes in a faster, less painful cycle.

Information

A related option, discussed later, that provides a different value proposition is the *synthetic testing* approach. In this approach, instead of simply verifying functionality here and now (in isolation), we do it continuously, sometimes upwards of once every minute or even more frequently, using robot/synthetic traffic. Typically this is done with some type of tooling, rather than being just a basic cron-job. Some off-the-shelf options include conventional monitoring like [Datadog](#), dedicated SaaS like [Checkly](#), or even a provided hyper-scaler option like [AWS CloudWatch Synthetics](#). And because these tests no longer just “check” whether the deterministic functionality is correct or not, we are actually not really talking about just “tests” any longer. I'd recommend this type of tooling primarily if you have very low or sporadic traffic, as you can catch the same issues with real production traffic.

Charity Majors, one of the big proponents of testing in production, writes how there are multiple actionable paths for developers once they've grown accustomed to what production should look like:

Everyone should know what a normally-operating system looks like, and how to debug basic problems. This should not be a rare occurrence.

If you test in production, it won't be. I'm talking about things like “does this have a memory leak?” Maybe run it as a canary on five hosts overnight and see. “does this functionality work as planned?” At some point, just ship it with a feature flag so only certain users can exercise it. Stuff like that. Ship it and see. [...]

You're shipping code every day and causing self-inflicted damage on the reg-

ular, and you can't tell what it's doing before, during or after. It's not the breaking shit that's the problem: you can break things safely, it's the second part that's not ok. Your bigger problem can be addressed by:

- Canarying. Automated canarying. Automated canarying in graduated levels with automatic promotion. *Multiple canaries in simultaneous flight!*
- Making your deploys more automated and robust, and faster in general (5 min upper bound is good)
- Making rollbacks wicked fast and reliable.
- Instrumentation, observability, early warning signs for staged canaries. End to end health checks of key endpoints.
- Choosing good defaults. Feature flags. Developer tooling.
- Educating, sharing best practices, standardizing practices, making the easy/fast way the right way.
- Taking as much code and as many backend components as possible out of the critical path. Limiting the blast radius of any given user or change.
- Exploring production, verifying that the expected changes are what actually happened, understanding what normal looks like.

These things are all a *great* use of your time. Do those things.

— Charity Majors, [*Testing in Production: Why You Should Never Stop Doing It*](#)

With that long quote, I also want to mention that quality is never just the engineer's job.

This approach, and its heavy dependence on proactive monitoring connected to any failure/bad states, are not just technical means—they are sociotechnical in nature. Just buying a tool or wiring your code to be 100% observable, but lacking procedure and culture, won't solve your problems. Sorry.

Test data management

What would a test be without some input data and expected outputs? Not of much use, that's what!

We can divide test data into two very big overall categories:

- **Test data local to the test**, either *inlined* with tests (such as `const input = 'Hello world!'`) or as *co-located files* (the previous example, but in a separate file).
- **Test data remote to the test**, which means any test data that is separately stored, handled, and otherwise managed in a lifecycle that does not necessarily follow that of the code base.

To really mix up things as well we can start thinking of the **malleability of shared test data**. Does someone own/control the data? Does someone update, change, or otherwise manipulate the test data set? Does that carry implications for other users and/or systems—does it make other systems break in functionality and/or testing? Yep, a lot of questions come in when we allow mutating any shared test data. Suddenly we need a **test data management process**. To be frank, that's probably not a good sign. If we can solve the needs in easier ways, then we should.

Continuing, test data can be:

- **Static test data**, which implies there are files and/or data that have been created at some point and are left mostly untouched. This is probably the most common scenario.
- **Dynamic test data**, which is generated programmatically for each test run according to some specification.

Based on my experience and what I can assume from my own interactions with others and the great internet, I can only assume that quite a few folks (and organizations) are relying on remote, static test data. Of these two, **you should make all efforts to make test data a local concern**.

Our baseline goals should be that test data is:

- Always trustworthy.
- Always present, that is, co-located in the code base and not in a remote source.
- Additional test data should only ever be needed in a minimum number of cases, if at all.
- If it is shared data, the test data management process needs to be clearly documented and the process followed; It is also unacceptable that shared data should break anything at all, so such aspects need to be refactored away.

Information

For more, similar, advice I recommend reading Google's article [DevOps tech: Test data management](#).

What about duplication and/or sharing of test data?

Duplication of test data should be entirely acceptable if that's ever needed.

But, I would much rather, however, point to the logical follow-up question: “If you need to duplicate and/or share test data, are you testing the same system across multiple teams?”. The answer should be a resounding “no”, which should similarly close down the conversation as it doesn't make any sense.

If you are testing the same things across multiple teams, that should be a clear indication of leaky [bounded contexts](#) and definitions of responsibility. As per [Domain Driven Design](#) and most reasoning today, you should never put testing someone else's system into your hands. If there is a clear division of responsibility—i.e. you cover your own systems—then there should be no (wide, at least) need for externalizing test data.

Advice for test data management

Creating readily available test data is part of the definition-of-done

As part of your way of working, Agile or otherwise, let test data creation be a clear part of the definition of done (DOD). If the DOD has this expressly stated and everyone understands how to create usable test data, then you are likely on a good way to not require any external test data at all.

Information

Of course, an approach such as test-driven development (TDD) will automatically fulfill this goal, and moving over to TDD could be another path to investigate.

Collecting initial test data

To test something, not only do you need to know what it is that you test, you also need to have some expected input and output data. When you build it yourself (i.e. “first party” testing), then test data should be trivial to come up with because the thing that is tested **is the same system**.

My commentary here will instead, therefore, deal with handling test data from third parties. A very convenient way to collect data when, for example, integrating with a third-party service, is to **actually use the API** or service itself.

Information

Here are some examples of API documentation. They are all good but differ slightly in presentation and approach.

- [GitHub REST API](#)
- [Figma API](#)

- [Stripe API](#)
- [Better APIs Workshop example](#)

Using API documentation, schemas, playgrounds, or other documents/tools you get a clear and functional idea of what needs to drive a given endpoint or feature, and what to expect of it.

Success

I've found this to be a good basis for setting up an [Insomnia Design Document](#) (similar to [Postman's request collections](#)) with these integration points together with any notes, links, and actual, working examples I might need. These design documents/request collections can then also be documented in README files or similar, and/or exported and kept with the source code. Having done this, not only have you documented the usage of a system or API, you've enabled the team to have exactly the same insights as well!

The input and output data can then be stored as actual test data together with the source code to drive unit tests (and any other tests). I normally store the data as JSON files because that's what the input/output is nearly 100% of the time in a modern API. Obviously, if you have other needs, have it your way.

Doing this at the beginning of a project or at the start of integration work greatly increases the momentum, because the sometimes tedious work of correctly accessing the third party gets dealt with using minimal means ([cURL](#), a REST client such as Insomnia, or whatever tool floats your boat) before committing to writing any code. So when you start writing the code for real, at least the payloads—and therefore some of the tests—will be clear *prior to there even being any code* at all! Some call it magic, I call it the lazy man's logic.

So, is this request collection deserving to be dubbed a collection of “tests”? Well, somewhat, but not conclusively, no. First and foremost it's a convenience and developer tool for the extended lifecycle of an application. As a mere request, it's at best a very crude and primitive test. The true value of the design document/request collections is to be an easy-to-access way of troubleshooting and enabling the use of any relevant third par-

ties *in an emergency or similar situations*. **Don't institutionalize the use of these request collections as any type of real tests if you can avoid it.**

Colocate tests and code

The corollary from the “Test environments” page, as well as from the start of the page, is that tests and code need to be co-located to support an optimal workflow.

If you are currently relying on external test data, consider if it would be logically possible to take any needed data into your own code base and cut the ties with the external source.

Prefer unit tests

This advice is really about finding the least hard way to do something. And...

Unit tests are the dumbest and most basic tests you can write. If you know that your functionality can be covered with these, then that's the ideal solution. Even if you *think you can't do it* try again and see if you change your mind.

Unit tests also have the benefit of being clearer in terms of test data. You can usually easily have test data inlined in the tests, or as local files.

Depend on as little data as possible

Continuing the crusade here. Now that you've removed external sources, you are using unit tests for all (or the vast majority) of your test cases, you can start simply relying on as little data as you can. Do you really need someone else to prepare the test data? Why? Is the system not deterministic, or is the team's understanding of it simply not correct?

This advice works even without any of those potentially dramatic changes. At the end of the day, dependence on *anything* creates a chokepoint in your process. Always be careful about dependencies, new or existing. History should not be your precedent as you improve the testing conditions.

Don't use sensitive/private/production data for testing

From a legal perspective, this is the only advice that you *absolutely have to comply with*.

Instead of using real data for your tests, opt for *fake* or *synthetic* data. This can be done as easily as:

- Using production data and cleaning it from any sensitive fields, for example by masking them or dropping them altogether. Do this once or if needed, create a script or program that can export clean data from a trusted resource.
- [Generating fake data with custom scripts](#) or with an open-source tool like [Faker](#).
- Using a commercial service like [Gretel](#), [Hazy](#), or [Mostly.ai](#).

Success

Of all these three I highly recommend the second one—generating fake data— as it is easiest, free, available off-the-shelf, and works perfectly with the overall approach around unit test-centricity.

The most important thing here is that more often than not, the *shape of the data* is a lot more important than the exact contents. If you are dependent on certain contents that might imply a problematic API or interface. If it's simply that certain fields need to follow certain conventions, that part is easily fixed with any of the above approaches.

Generate test data dummies

This is a direct follow-up on the previous point. Any actual production of fake or synthetic data should be done in a mechanical, automated, predictable way.

Using a tool like Faker, it's ideal to dynamically create test data for a test as you need it. This brings several benefits:

- No reason to save and check in (commit) test data

- Allows more confidently exercising the code as neither you nor the implementation can optimize for a specific dataset
- Lower logical complexity

The primary drawback is that if the volume is very big then doing this dynamically might be more taxing on the test hardware and may push up the test's final run time considerably.

Given that's not the case, the dynamic test option is excellent for configurable cases on a higher level, i.e. for example for integration tests.

Information

Related, but different, approaches using dynamic data are possible with [mutation tests](#) and [fuzz testing](#).

Use QuickType to generate SDKs to validate against

Lastly, you can use tools like [QuickType](#) to create models (code bindings/SDKs/packages) that you can directly validate inputs against. I've used this approach in [contract testing](#) scenarios and similar, and it is even the glue driving my [TripleCheck](#) projects. Doing it this way you can effectively remove the conventional usage for test data as it's generated into a testable model.

This approach is a more particular technique that you might not need for the majority of cases, but keep it in mind!

Testing serverless isn't that different

Testing serverless isn't different. Mostly, at least.

There seems to be a widespread notion that testing serverless applications, or microservices in general, is very different from testing other applications. While it's true that there is a lot more surface area and integration area in a microservice-dominated technical landscape when seen service-by-service, the differences shouldn't be that big for actual testing. At the very least they shouldn't necessitate a great deal of stress!

A web application written in Node, using a Fastify server will not be leagues different from the same application broken into several Lambda functions with an API Gateway exposing them.

How serverless (and the cloud in general) clarifies responsibility

Overall responsibility

One of the clear differences between serverless and most other paradigms is that serverless services (like DynamoDB, Lambda, and Firebase) are managed services; sometimes effectively being fully managed.

Information

See [this article](#) by Google's Priyanka Vergadia for more on *serverless* versus *fully managed*.

In short, this means that the provider (i.e. AWS, Google, Microsoft, or whomever it may be) takes care of the majority or all of the management, such as making sure the service is available, in a useful state, securing some of it, and generally automating the heavy lifting involved in dealing with computer systems. Your benefit is that a lot of that headache is dealt with. By leveraging serverless—which does have a slightly higher cost premium compared to unmanaged “raw” hardware—we can actually win back most of the maintenance and operations costs we would have to deal with otherwise. We can also generally trust the providers to provide a more secure and well-performing bottom line than if we

did all of the work ourselves.

Serverless is, as I am writing this in 2023, one of the current pinnacles of truly being “cloud native”. We offload all of the hard specialist bits to someone else and can interact with resources dealing with, for example, compute and storage globally simply by using APIs. Using the cloud in any capacity, we commit to the [shared responsibility model](#) and how it divides activities expected of the provider and from us, the consumer. This model is used with minor differences in all (big) clouds. This all leads to us having to cater to a lot fewer concerns. For example, the security model of the cloud dramatically changes the old “castle-and-moat” models.

Ultimately, we are abstracting hardware into APIs, that is...software! But our business software itself has not really changed all that much.

Logical responsibility

Without going into a big “microservices vs monoliths”, or “which is best” argument—in which the only real, but boring, answer is “it depends”—the single clearest difference is that microservices, well, at least ought to do less than a classic all-in-one monolith.

If you are new to serverless functions, have no doubt in your heart that you *can* certainly create a [monolithic Lambda](#), or worse, a [distributed monolith](#). All too many have walked that path: some oblivious to the negatives, and some who swear to never use serverless functions again. Stack Overflow and Medium are full of these folks.

How big a Lambda should be is something you will have to read more about elsewhere.

Information

Some good sources to make an informed decision on that would be:

- [Learning Domain-Driven Design](#) by Vlad Khononov (2021)
- [Building Microservices](#) (2nd edition) by Sam Newman
- [Identify microservice boundaries](#) in the Azure Architecture Center
- The chapter [Domain driven Lambdas with a use case approach](#) from my

It's very typical that a **service** (the logical composition and mass; the thing-as-a-whole) involves multiple **microservices** (the executable sub-part of that entirety), meaning a given deployment may entail several functions being changed.

My own thinking and accumulated wisdom lead me to recommend choosing either of these approaches:

- **Workflow-oriented**, meaning all functions are strictly contained logically. Think of this as a function that does only a single, unambiguous thing. This works well for many simple use cases and for piecemeal code. More often than not, this will suffice, especially for typical event-driven flows. In this approach, no function overlaps (meaningfully) functionally or logically with another.
- **Entity-oriented**, (which taxonomically *may* be misleading) in which we may be building more typical, complex systems. I call it this because we tend to operate on things such as entities and we require orchestration of classes and pieces of code that may be reused between functions. Some call this the “fat Lambda”. I think this is a very good compromise between traditional programming and fending off a too-chatty, minimalistic function landscape.

Before [Lambdas with function URLs](#), the only way to make a Lambda function become publicly accessible would be to host it behind an API Gateway. With this said, AWS API Gateway (or whatever similar capability your cloud offers) is—and will most likely remain for many years—the most general-purpose way to expose functions. One major benefit is that by using an API Gateway, it becomes very clear how your functions map to the exposed URLs. With a good software architecture, you can easily draw a direct line between the endpoint's path names their logical functionalities, and their respective function code. That itself could be informed by the use cases and/or requirements for the system.

Regardless of choice, in any non-trivial situation, practically no single microservice will exercise all code paths/branches, conditions, lines, or statements. I would say that in the microservices world, this is a lot easier and more intuitive to understand than with a monolith because it's so ingrained in the very notion of microservices. Monoliths can

become daunting as they tend to not offer as expressive interfaces and system boundaries (since they have fewer of them!).

The intellectual labor (and complexity) increases somewhat with microservices. All in all, as Yan Cui has written, serverless and microservices may ostensibly feel to be more complicated than a monolith but in fact, that is not correct—microservices are truer as they do not hide these details quite as much as classical monoliths do.

Impact of microservices and serverless on testing

Don't test the “cloud” or its side effects

The “cloud” is a third party, and as you likely remember, one does not test third-party code. Care about your business logic and test that instead. As with any other testing, don't test for side effects (“did the SQS get my message?”). No difference, compared to any other scenario.

Don't emulate the “cloud”, instead use actual infrastructure or mock if needed

While I respect that we want the “feel” of a local development environment, there are good and less good ways to deal with this.

Instead of something like [LocalStack](#), which only complicates your workflow and has to keep pace with all the changes in the cloud, use something like the lightweight [Serverless Offline plugin](#) to get the API and function parts working (so you can run against them), and if needed, use [aws-sdk-mock](#) (or a similar capability) to mock rather than to emulate other functionality, like queues and storage.

Information

As I already wrote in the chapter on handling and mocking side effects, always prefer composition and your own local mock implementations over heavy-handed, potentially problematic mocking if possible.

Also, there's nothing saying you can't deploy your own infrastructure to test and work against. I highly advise you to do this if possible—this is a big change against the traditional world in which deploying infra was only permitted by the basement wizards.

Microservices are APIs, so test the APIs why don't you

Request/reply, streams, or events... Whatever your input format, all of these can be tested, ideally with test data you have collected, using their correct formats. For TypeScript, [there are also types you can use](#). Prefer unit tests (business logic) over integration tests (infrastructure).

Overall this is not very different from any other unit testing and integration/API testing:

- Collect test data for the relevant input type (since they have different shapes).
- Decouple the handler and the use case from each other.
- Make sure your code wrangles the input into a well-known, well-defined Data Transfer Object.
- Write unit tests against your use cases (infrastructure-independent) with the Data Transfer Object.
- Write integration/API tests against anything that is relevant and does not simply duplicate your business logic tests.

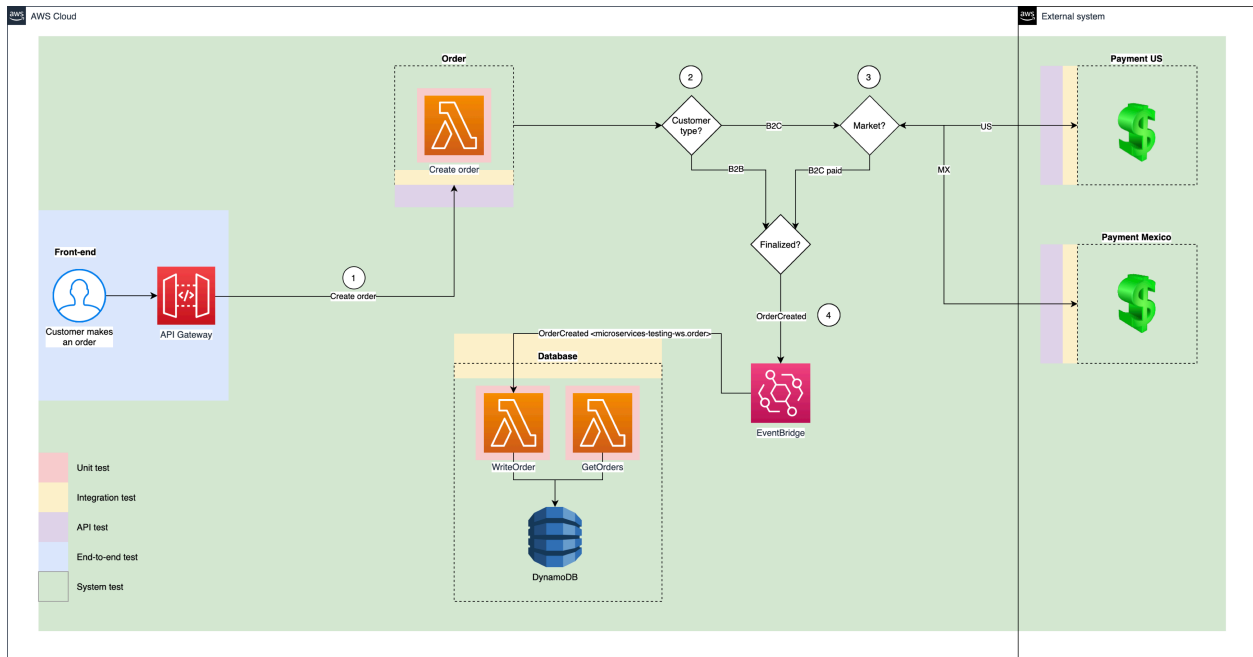


Figure 1.28: All of the green areas can be individually developed, tested, and deployed. Make the most of that fact!

In the above diagram, there is no need to check that *all parts* are working, at least for the test scope of each individual service (mint-green box). Ensuring the business logic is working, that our interactions with mocks work, that types are correctly used, and that data wrangling (incl. validation, sanitization) is done will go a long way in the scope of a specific service.

In closing

Some notes for you to take away from this:

- Testing with microservices and serverless means having clear boundaries.
- Stay true to the microservices mentality: Keep your test boundaries equally tight and clear.
- Only test *your* code and services, not infrastructure or third parties.
- The internal logic of a microservice is tested with unit tests.

- Each microservice is exposed through an API: That's the actual "integration testing" part! Only test through the API if there is good reason to believe there are classes of issues you can rectify through such tests.

Testing considered harmful

The majority of material in this book, and definitely in the “Understanding modern test automation practices”, should already give you fairly clear advice on what *to actually do*. This page collects some of the things you *shouldn’t* do, inverting the perspective for a moment.

Over-reliance on QA and/or “over the wall” testing

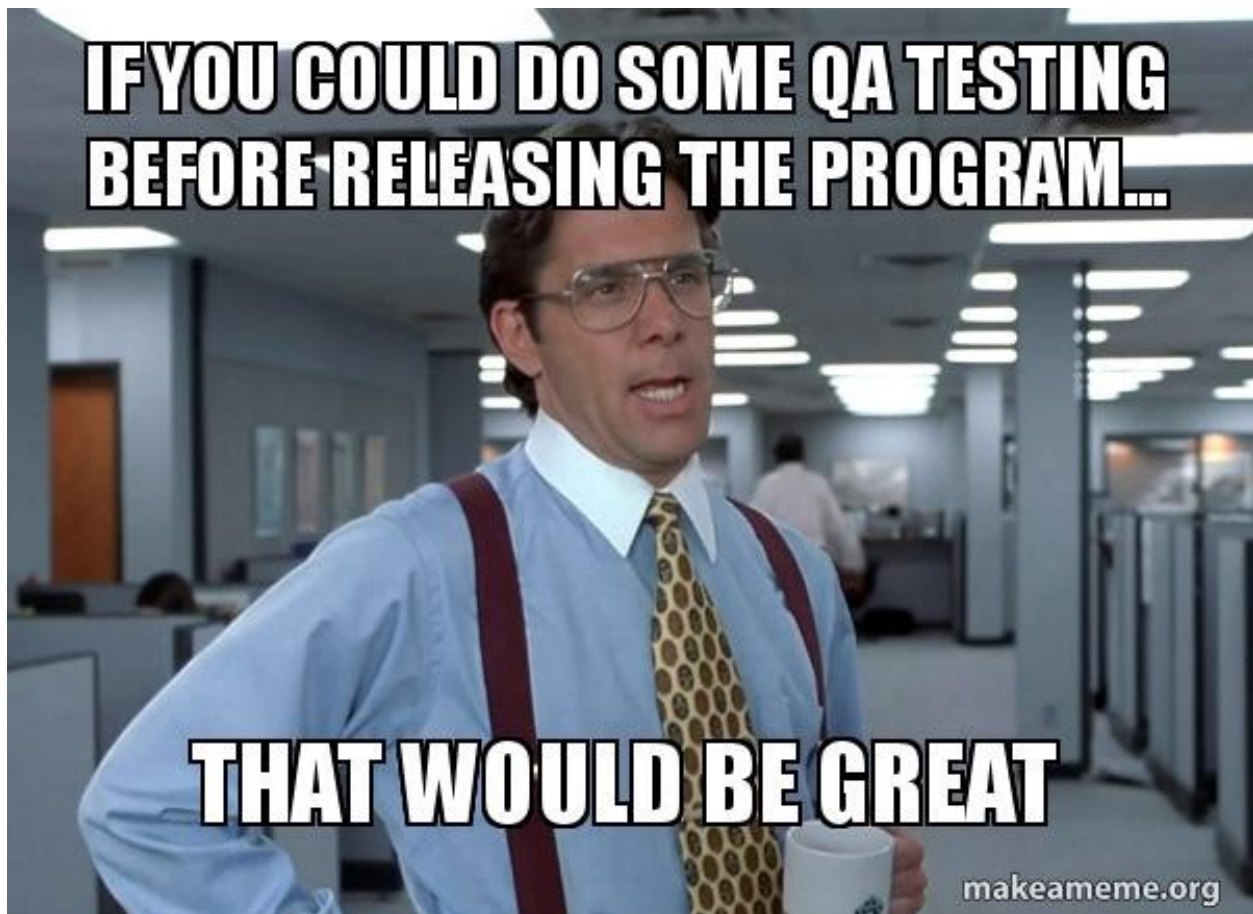


Figure 1.29: Yeah no.

This scenario is one of the reasons I even took the time to write this book. Don’t do this.

Software engineering is a sociotechnical practice – comprised of both social and technical concerns that we need to address throughout our work. While a QA culture fulfills

part of the technical requirements we might have, it doesn't fill in the organizational and cultural values we try to reach with ownership of code and quality *within the team*, insistently automating such testing, and making testing an activity that is closely related to the writing of the code itself.

Success

Solution You will probably want to introduce developer-led testing both on the technical level (knowing how it works), the team level (that *we do it together*), as well as on the planning level (making sure testing is a part of definition-of-done and that you as a team take that responsibility). Perhaps you can also let the QA people you might have worked with either teach your team some tricks of the trade, or they could learn from your work if they are more of manual testers today.

TDD misinterpretations

Again, I'm not a TDD person myself, though I take a lot of what is said and written as very good guidance and let that influence my own approach. I can easily see how both rabid TDDism or allergic reactions to TDD could haunt even a moderate discussion of moving from "over-the-wall-testing" to more automation. Don't let a developer-led approach to testing be conflated with TDD, which is in the big picture just another approach to getting to the same goal.

Success

Solution Clarify the strategy and approach that you will take in clear terms so everyone knows what to expect in reality in terms of ways of working, tooling, and procedure.

Too many tests (i.e. redundant tests)

Redundant tests mean that we have several tests covering the same assertion.

At some level, there is sometimes no way to completely get rid of redundancy. For example, in a test suite with 5 individual passing tests, all will likely assert that the class or function itself will run (in the meaning of “start”) without crashing. In reality, you have five tests that cover some of the same lines, branches, and statements. They might indeed all do *something* specific as well, but in terms of ensuring that the hooking-and-wiring is set up, it would be technically correct to call them redundant. Maybe we can call this *functional redundancy*.

Real and painful redundancy however happens at different scales and scopes.

Information

On this theme, consider reading:

- [Are Your Unit Tests Getting Redundant? Here's How To Write Them Effectively](#) by Oliver Spryn
- [Test Suite and Test Redundancy: How Your Watchmen Have Turned Against You](#) by EuroSTAR Huddle

For example, in my project [Figmagic](#) (started in 2018) I have followed a classic convention in which each class and function gets its own tests. You will see, among others, that Angular enforces this old-school convention. At first blush, this makes sense, as there are not many scenarios in which something does not deserve *some degree* of testing. But the back side is that we are encouraged to write code that is modular without hierarchy—something that the onion/hexagonal/clean architectures all have tried to address. So, while this is not a “problem” per se, it is something that can lead to redundant tests. In my case with Figmagic, for example, there is quite a bit of unnecessary redundancy because of this approach.

Success

Solution Over time the testing evolved to something closer to what I think is “correct” today. In this newer approach **I will typically write the passing use case tests first of all, since these are the most important tests of all:** without these passing the expected behavior will not manifest itself. Individual functions may work, but the big picture will fail. That’s a bad thing. After these tests, we can investigate the detailed coverage and start smoking out the difference/delta we need to get to 100% coverage. Very often a big set of tests will have to be written, that deal with the failing states. Depending on the rigor of your software architecture you might have a wider surface to cover at the outer scopes (i.e. such as use cases) than on the private methods on an entity (because validation may have occurred in earlier stages). Writing tests that validate failure states on the use case level may be harder or less intuitive. Instead, we want that detail *on the actual class or function tests*. That’s where I would add those tests. So:

- Use cases test for all intended positive states.
- Use cases should test for all reasonable use case-level failure states.
- Individual tests (i.e. per class or function) handle primarily failing states (since positive states should be exercised with the coarser-grained tests coming from the use cases).
- Individual test files may contain additional tests for positive states if those are not easily or intuitively covered from the “outer” use case tests.

Conducting your testing this way ensures we have the right detail in the right locations. It may not come as easily as the more simplistic “just write all the tests in the respective file”, but it will certainly minimize redundancy, allow you to run tests where they matter, make you write *fewer* tests (the exact number of tests is less important than having *confident* coverage), and therefore leads to a more maintainable stable of tests.

Too few tests

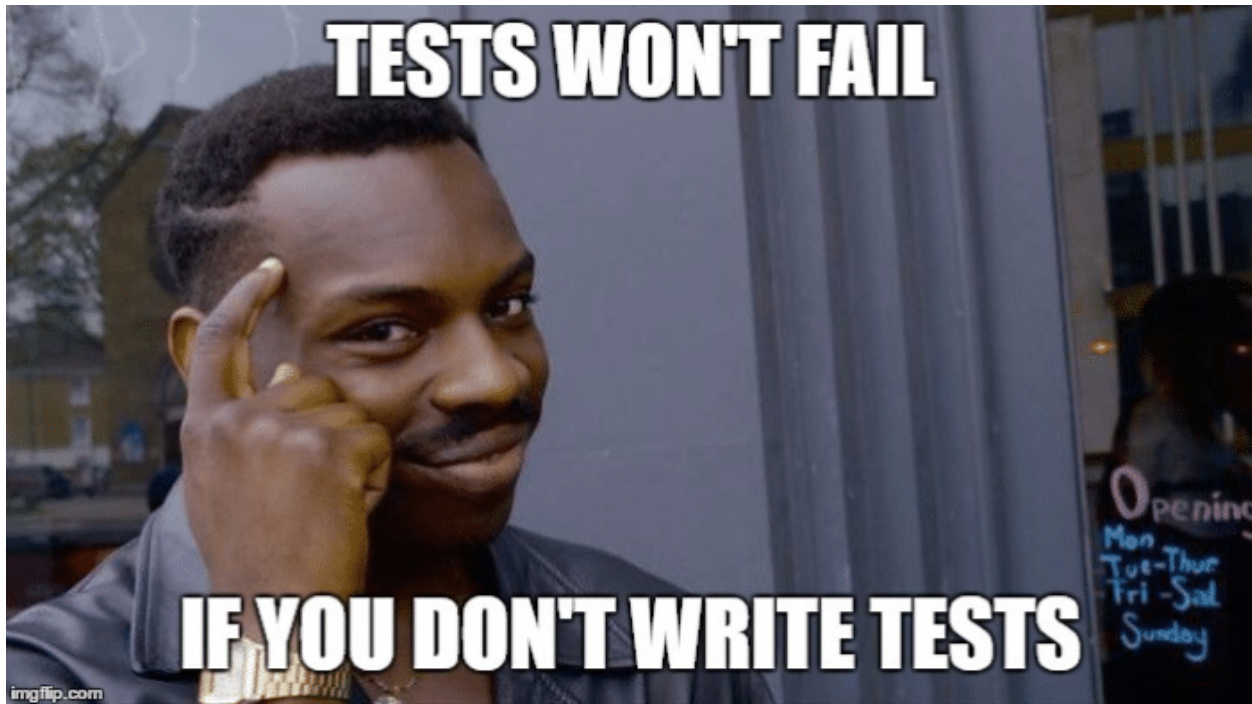


Figure 1.30: Unfortunately not just a meme

This one may seem like an oxymoron, as the very intent of this book is to give you practical guidance in writing tests and why you would want to do that. Still, it bears mentioning.

If you are building a basic, non-critical application and have a few (say in the range of a dozen) tests for parts you *know* are something you need to hold some control over, but your total coverage is, say 40%, then that's totally fine!

Just as stating an absolute limit or expectation on tests and test coverage is hard, so is saying when there are too few of them.

A heuristic is generally more applicable and easier to use. As I wrote on the page about "Confidence-based testing", confidence should be the driving aspect irrespective of any strategy. The more critical your workload is, the more you should also apply Reagan's "[trust but verify](#)" attitude. So, in your personal low-criticality app having no or few tests is probably entirely fine, if you feel that you can trust the app to work once it gets deployed. However, in a business environment in which the criticality element is significantly increased the confidence factor becomes a distributed concept that somehow has to be proven to more people than yourself.

Success

Solution In a professional context, the common expectation of code coverage is often for a value over 80%. Beware that code coverage is never *for you*! It's a level of guarantee (for what it's worth) for every stakeholder and user of the application.

Blind trust in code coverage

Similarly, code coverage can be misleading. Code coverage can be high but misunderstood, such as only looking at line coverage instead of branch coverage or statement coverage which are both more powerful and meaningful factors.

It's also not that hard to fake coverage by not asserting anything, or even more mischievously, having high coverage but not failing CI builds based on such failures.

Code coverage and testing overall is just one angle of the much larger code quality spectrum.

Success

Solution The first thing to do is to ensure that code coverage is accurately reported and that test infrastructure (CI, test scripts, etc) is working as intended so that we can trust the numbers at all. What I am after is not some kind of mechanical failure on part of the tools but rather to ensure tool integrity so no developer has manipulated them as mentioned above. Secondly, it's worth having a richer strategy for quality that extends beyond (unit?) testing as is implied here. Consider using tooling such as SonarCloud, DeepSource, Codiga, and similar to run deeper and more exhaustive static code analysis.

Brittle and flaky tests

This is what you end up with if you cannot depend on your tests running or working as intended every time that you start them. Unbelievable as it may sound to someone who hasn't done a fair share of testing in their career, this is quite common it seems, and is one of the first things you should address in there is already test automation but a low trust in the tests.

Personally, I tend to see these tests having one or more of the following properties:

- Async in synchronous contexts
- I/O dependence
- Shared environment conflict (constants, process environment variables...)

Success

Solution Locate the brittle tests and analyze the root cause of the problem. Address it conventionally by code. If it's one of the above, consider building test utility functions like `setEnv()` and `clearEnv()` that are run in lifecycle methods like `beforeEach()` and `afterEach()`. Generally, you also really don't want to test too much I/O stuff either. While it *can work* it may be painful and not add meaningful confidence to your tests if you really aren't interested in the exact details of the I/O (which you probably aren't).

Be overly framework-dependent

Testing tools should be very low on your list of things that are highly customized. Have a transactional, Platonic relationship with your tools—not a love affair. Once you need to cross over to other tools, let it be possible to do so in a very short time (in some hours, not in days or weeks).

Success

Solution Use or migrate to lightweight tooling that uses nomenclature, patterns, and standards that come with less baggage. Consider options like [AVA](#) or the [Node 18-native test runner](#).

Running tests in practice

In this grand section, you'll explore a round-up of some common testing types, a comparative analysis of what they do and when they matter, and real-life examples of their usage.

Static code analysis

Information

Surface area Code **Confidence level** High, in the local code scope **Granularity** Very high precision but does not say much about the “big picture” **Pros**

- Cheap (typically)
- Fast
- Mostly unobtrusive
- Relatively uncontroversial
- Helps streamline and enforce conventions

Cons

- Does not test logic, just the line-by-line syntax

Success

When to do this type of testing? Always.

Static code analysis is a total game-changer for writing better, more readable code. It is a wide category of tools—some are commercial and run only within third-party products (such as [DeepSource](#) or [CodeScene](#)) and others are IDE-centric and open-source, like [Checkov](#). Most commercial tools will bundle open-source packages into a more comprehensive bundle.

Information

Note that the commercial model does not preclude something running (or not) in an IDE!

The really big use case for a static code analysis tool is to offload menial, tedious work. Candidate activities for this category include:

- “Linting” code, finding problematic patterns such as security issues.
- Standardizing code formatting, such as spacing, tabs, line wrapping...
- Checking files for misconfiguration or problems.

By having configuration files in the repository your tooling will pick up the rules from them and apply them, meaning that in a team the same rules will apply to everyone. For the question of when to run them, I recommend:

- Apply the tools on each save; this works well with [ESLint](#) and [Prettier](#), for example. This allows you to receive instant feedback as you work.
- Run the tools (and any other testing) in a pre-commit hook, for example with [husky](#), so that you won’t miss anything before committing the code.
- Run the tools as part of your CI/CD pipeline as part of your baseline tests.

Information

See [here for an example](#) of what a husky pre-commit file might look like.

Using ESLint and Prettier in the IDE

Probably the most common tooling for a TypeScript developer will be ESLint and Prettier. I will show you how I typically set them up for a project.

ESLint is a “pluggable linting utility” that offers rich functionality to detect issues and fix problems with your code in a deeply configurable way.

Prettier is, on the other hand, an “opinionated code formatter”. There is a good article by its team called [Why Prettier?](#) that does a good job of explaining the pains of code style

guides and why Prettier is a good and painless way to get most of the benefits without the brutal work that it takes to get to such style guides. The most significant thing about Prettier is that it also reformats any code (on save) so you literally never have to go in and manually edit things to follow the conventions.

Installing and configuring

We will first install the required packages as developer dependencies. I will assume you already have TypeScript installed as a developer dependency. Run:

```
npm install prettier eslint eslint-config-prettier eslint-plugin-↵  
↵ prettier @typescript-eslint/eslint-plugin @typescript-eslint/↵  
↵ parser --save-dev
```

Information

Read more about the other packages here:

- [eslint-config-prettier](#)
- [eslint-plugin-prettier](#)
- [@typescript-eslint/eslint-plugin](#)
- [@typescript-eslint/parser](#)

It's not a very lean list, I'll give you that, but it's a bit of a headache to always get these tools to work together as expected without some extra plumbing.

Now to create a configuration file, `.eslintrc.json` in the root of your project:

```
{  
  "plugins": ["@typescript-eslint"],  
  "extends": ["eslint:recommended", "plugin:@typescript-eslint/↵  
↵ recommended", "prettier"],  
  "parser": "@typescript-eslint/parser",  
  "parserOptions": {
```



```
    "ecmaVersion": 2022,  
    "sourceType": "module"  
  },  
  "rules": {  
    "@typescript-eslint/no-explicit-any": ["off"],  
    "@typescript-eslint/ban-ts-comment": ["warn"],  
    "complexity": ["warn", { "max": 9 }],  
    "no-async-promise-executor": ["off"],  
    "no-prototype-builtins": ["off"],  
    "no-useless-escape": ["off"]  
  },  
  "env": {  
    "node": true,  
    "es2022": true  
  },  
  "ignorePatterns": ["node_modules/**"]  
}
```

Here we are using `@typescript-eslint` as a plugin so we can use ESLint with TypeScript. We set the parser to a modern ECMA version and extend the ruleset of prettier itself and the recommended subsets of two of our plugins. This all gives us a stable foundation to work with.

Next, we set a few rules of our own: We disallow warnings for the any type (so we *can* use it when/if needed) and we turn down the `ban-ts-comment` rule to only being a warning (as we can make good use of that in negative tests when we want to test dumb inputs). Then we set a cyclomatic complexity threshold at 9 so we get warnings for anything that might look like a mess.

Finally, we set the environment to understand that we are in a backend/Node context.

Those are the basics of ESLint in TS!

The same story goes for the Prettier config—but a lot easier—at `.prettierrc`:

```
{  
  "useTabs": false,  
  "printWidth": 100,  
  "tabWidth": 2,  
  "singleQuote": true,  
  "trailingComma": "none"  
}
```

This should all be quite apparent in how we want Prettier to work its magic.

Warning

You may need to restart your editor for changes to be picked up.

Demoing some of the basic functionality

With the tooling in place, you can now try things like:

```
const message = "Hello world!";
```

Which upon saving now magically turns into:

```
const message = 'Hello world!';
```

This is Prettier in action.

Or how about very long, ugly lines like:

```
function willBeBrokenIntoShorterLines(param1: string, param2: string, ↵  
    ↵ param3: string, param4: string, param5: string, param6: string) {  
    console.log(param1, param2, param3, param4, param5, param6);  
}
```

Which now become broken up into more readable, shorter lines:

```
function willBeBrokenIntoShorterLines(  
    param1: string,  
    param2: string,  
    param3: string,  
    param4: string,
```

```
    param5: string,  
    param6: string  
  ) {  
    console.log(param1, param2, param3, param4, param5, param6);  
  }
```

Let's look at the cyclomatic complexity check provided by ESLint.

```
function overlyComplex() {  
  if (1 > 2) console.log('');  
  if (1 > 3) console.log('');  
  
  if (2 > 1) console.log('');  
  if (2 > 3) console.log('');  
  
  if (3 > 1) console.log('');  
  if (3 > 2) console.log('');  
  
  if (3 > 1 && 2 < 3) console.log('');  
  if (2 > 1 && 1 < 3) console.log('');  
}
```

This will now trigger a warning: Function `'overlyComplex'` has a complexity of 11.↩↪
↩↪ Maximum allowed is 9. You might want to set a different limit than 9, and you might want to use actual errors rather than warnings. I've used the value 9 myself after seeing that CodeScene has gone with that specific number, and those are some pretty smart people.

The tools displayed here are boring, which is a good thing. They make your life, and those reading and working on your code, a little better and that goes a *long* way.

As a tool during CI

As mentioned previously, you can typically run this category of tools in CI as well.

However, some *are only available to run as a CI integration* and aren't available as an of-line/IDE tool. This may be for a variety of reasons, including that the analysis stage may not be something you can do in a few seconds. As an example of this, it is typical

that dynamic code analysis may even take hours to complete.

A central benefit of a tool external to the IDE is to get a consolidated surface to see code quality metrics, statistics, states, and trends. This makes even more sense when you have a bigger engineering organization.

Some examples of commercially provided tools include (in alphabetic order):

- [Codacy](#)
- [CodeScene](#)
- [Codiga](#)
- [DeepSource](#)
- [Embold](#)
- [SonarCloud](#)

Information

The tools in the above list all offer generous free (or open-source-focused) plans, so go ahead and try a few of them. Some of them are possible to run in the IDE or with local analysis as well.

There are certainly *a lot* more tools, and because code quality is a very wide umbrella you will find all kinds of varieties of tools that for example may focus more on delivery and issue tracking ([LinearB](#)) or security ([Veracode](#)). The above, however, does represent more developer-centric products with a clear focus on the actual code-level details.

Integrating a tool, like those mentioned above, is usually not a very taxing thing if they provide ready-to-use integrations for your version control system (GitHub, BitBucket, GitLab...). After integrating, my recommendation would be to run the testing tool on any commits or pull requests as early as possible.

There's no reason to go into more detail on that here as it will depend entirely on the tool and version control system. The important thing is to actually run the tools if you have them!

Unit testing

Information

Surface area Any individual (or combined, such as a higher-level function calling

other) classes and/or functions **Confidence level** Very high at the logical level

Granularity Can be any combination of assembled, local code **Pros**

- Cheap
- Fast
- Easy to write
- Helps enforce good code and structure
- Provides very good logical proof of functionality
- In TDD or TDD-inspired approaches unit testing will be done prior to code meaning you always know what you are dealing with

Cons

- Covers only the application layer and not other layers

Success

When to do this type of testing? Always.

The beauty of unit tests is that they are really malleable: We can make them wider or more narrow in scope as needed, and they are fast and “cheap” to run.

I am not making the claim that unit tests are the only good tests; I am stating that they are incredibly useful and great confidence providers in well-built systems. Kent Beck recently paraphrased Tim Ottinger on the qualities of good units tests, which are:

- **Isolated** — Unit tests are completely isolated from each other, creating their own test fixtures from scratch each time. (Note that I’m not saying these are the only useful tests, just that if tests aren’t isolated you’re going to have a hard time making the case that they are “unit tests”.)
- **Composable** — Follows from isolation.
- **Deterministic** — Should be. Code that uses a clock or random numbers should be passed those values by their unit tests.
- **Specific** — If a unit test fails, the code of interest should be short.
- **Behavioral** — If the behavior changes accidentally, a unit test should fail.
- **Structure-insensitive** — This can be a challenge for unit tests. Too much mocking, especially strict mocking, is a structure sensitivity nightmare.
- **Fast** — Yep.
- **Writable** — Good interface design makes writing unit tests easier. Alternatively, difficult-to-write unit tests are the canary in the bad interface coal mine.
- **Readable** — It can be challenge to write readable unit tests, because you are seeing so little context compared to the whole system.
- **Automated** — Yep.
- **Predictive** — Unit tests passing likely gives little confidence that the whole system is working. Unit tests failing should give great confidence that the whole system is **not** working.
- **Inspiring** — A frequently-run unit test suite gives great confidence the programming is progressing. Sometimes I run my unit tests 2 or 3 times, just because it feels good (and they’re wicked fast).

— Kent Beck, *Desirable Unit Tests: A Test Desiderata Perspective*

Considering the limits of unit tests

How about any backside to these tests? As always, yes, there are backside too. Perhaps the most pressing problems are related to:

- Unit tests may not be able to cover all branches or cases.

- The lack of verification of external components, such as databases.
- Writing tests for systems with unclear and/or poor system interactions and bounded contexts.

Information

With this said, I am a firm believer that sometimes people are just being either actively ignorant, not aware of modern software development practices (including staples like version control), or that these folks are more philosophers than pragmatic practitioners.

In reality, we can address, to a meaningful degree, most of the hard parts here, given you have some software engineering chops.

Covering most cases

Modern test tools use some type of visual output that will give you a correct understanding of what lines/statements/branches of the code were running. Good code is written in a way in which it's possible to test it, i.e. the code is “dumb” and provides clear ways to interact with it and finishes with clear outputs.

With this said, indeed it's possible to get 100% test coverage without trying “all” cases. In all honesty, it would be impossible to cover “all” cases of anything other than trivial systems for combinatorial reasons: Given a single boolean condition, you would need two tests at a minimum. Given more complex inputs will significantly raise that test count. What you want to ensure is that you cover all code paths in the *expected* “success” and “failure” modes. And as any good software engineer knows, you look for patterns and generalizations rather than matching on exact inputs.

For highly complex cases you can potentially make things easier by introducing early-stage validations and other logic to look for anything out of the ordinary, such as too-long strings, use of unexpected characters, too many or too few items in an array, etc. Suddenly you can now build (and test) such checks independently from the higher-level use cases you are better off concentrating on.

Write “simple” and testable code—it will serve you well.

Verifying the “big picture”

It is true that we don’t want to “test” infrastructure or external components in unit tests. This is not an actual problem, per se, but a necessary constraint. We can, and should, certainly use other types of testing to verify this functionality. In practice, however, this isn’t typically a problem: Mocks or similar will be fine enough, and if you can arrive at a solution in which you can use minimalistic mocks (remember the SOLID principles!) then you know that *within the boundary of this system*, everything is working.

There’s also a concept being used called *component testing*. I’ll only speak to my own best professional opinion but I am also not hearing a lot of discussion around component testing; ironically it’s mostly testers and QA people who have raised this notion in my own vicinity.

The point of a component test is to strike a middle ground between unit tests and the more cumbersome integration tests, which require some actual infrastructure to be up and running. It’s a great idea, but even better is that you can (as I have done in this book) simply erase the component test from your mind, as nothing says that unit test can only test individual, “small” functions—in fact, they can absolutely test entire use cases, effectively testing from a functionally higher level.

Use layers in your source code (following for example [Clean Architecture](#)) to make it easy to run tests on both the “higher” and “wider” level (entire functional use cases) as well as on the “lower” and “narrower” level of individual functions (for example ensuring that correct errors are thrown on invalid inputs).

Testing with unclear boundaries

This of course brings us to building or evolving toward systems with clear interactions and bounded systems. This is a necessary piece of our work, because frankly, why should you test something if you don’t know it’s yours to test? For some reason, a lot of people accept this as some kind of inevitable fact, but it really isn’t. You need to be crystal clear about what goes into your box and what you need to trust, that comes from somebody else. At a minimum, they should provide artifacts like API specifications, service level

objectives, support, or whatever it takes for you to cut your ties on the implementation level. You wouldn't test an AWS API, so why would you test another team's things? It's wise to remember that many mistake testing for observability, a very different property and set of tools that will better assist you in understanding if something went *wrong* rather than *prove that you are doing it right*.

Draw a hard line between “what you own” and what you don’t and consider increased observability efforts for third-party failures rather than being delusional about being able to “test away” things that don't work in systems you don't manage or have first-hand access to.

Information

While there are more articles on this, a quick glance at [this section](#) on Wikipedia should give you a hint that there are further limitations as well.

Examples

Positive tests

Sometimes a unit test can be made incredibly short and succinct. This is a test from [Figmagic](#), written in Jest.

```
// Imports etc. above
test('It should set a negative "quantity" to 0 and return a value based↔
    ↔ on the two defaults for "quantity" and "scale"', () => {
    expect(roundColorValue(-4.2)).toBe(0);
});
```

If you can, keep your tests that short! No reason to write messy tests.

But, yeah, granted, the function itself was really terse and exact. How about we look at something that's a bit messy, just for kicks? Here's one of the tests for reserving a time slot in [my DDD example project](#). This is represented by testing at the outer-most layer, the use case. Note: This uses AVA.

```

import test from 'ava';

import { ReserveSlotUseCase } from '../../../src/application/usecases/↵
↵ ReserveSlotUseCase';
import { setupDependencies } from '../../../src/infrastructure/utils/↵
↵ setupDependencies';

import metadataConfig from '../../../testdata/metadataConfig.json';
import { correctSlots } from '../../../testdata/TestDatabase';

import { setEnv, cleanEnv } from '../../../testUtils';

setEnv();
const dependencies = setupDependencies(metadataConfig as any, true, ↵
↵ correctSlots);

test.beforeEach(() => setEnv());
test.afterEach(() => cleanEnv());

/**
 * POSITIVE TESTS
 */

test('It should reserve a slot', async (t) => {
  // Arrange
  process.env.SECURITY_API_ENDPOINT_GENERATE =
    'https://RANDOM.execute-api.eu-north-1.amazonaws.com/shared/↵
↵ generateCode';
  const expected = 'RESERVED';
  const slotId = '0128c8d4-cb2f-460d-8f49-c5587ebbf83a';

  const slot = correctSlots.filter((slot: any) => slot.slotId === ↵
↵ slotId)[0];

  // Act
  const response = await ReserveSlotUseCase(dependencies, { ...slot });
  t.is(response.code.length, 8); // Assert!

  const slots = await dependencies.repository.loadSlots();
  const updatedSlot = slots.filter((slot: any) => slot.slotId === ↵
↵ slotId)[0];
  const { slotStatus } = updatedSlot;

  // Assert

```

```
t.is(slotStatus, expected);
});
```

It's not my most beautiful test, which is part of the reason I demonstrate it. I wouldn't call it bad, but we absolutely see that sometimes things do get more complicated. A benefit here is, as mentioned several times before, that we exercise *a lot* of our code here, so the test has a very high value for our confidence.

I'm doing two assertions, despite common wisdom. This is because I don't otherwise ensure we have a correct-looking response code, thus packaging that check in here.

There is also dependency injection happening here, in order to remove the real database. Also, the `correctSlots` is a set of test data.

All of this is to ensure that the logical functionality remains stable, rather than testing databases or other infra.

Negative tests

This is cleaner and focuses on ensuring we can throw the right type of error if we have a slot with an incorrect status.

```
/**
 * NEGATIVE TESTS
 */

test(
  'It should throw a ReservationConditionsNotMetError if attempting to ↵
  ↵ reserve a slot with non-OPEN status',
  async (t) => {
    // Arrange
    const expected = '8613678f-2bbe-4fdd-93b8-97f65674c94f';

    // Act
    const slot = correctSlots.filter((slot: any) => slot.slotId === ↵
    ↵ expected)[0];

    const error = await t.throwsAsync(async () => {
      await ReserveSlotUseCase(dependencies, slot);
    });
```

```

    // Assert
    t.is(error?.name, 'ReservationConditionsNotMetError');
  }
);

```

Here our use case structure and arrange-act-assert composition shines bright.

Mocking API dependencies (HTTP calls) with MSW

It tends to be messy when we have to deal with HTTP calls. Given these are actual network requests, it's not always quite as easy as just writing a mock object to deal with this (indeed, it could be, but hang on). [MSW](#) offers a good way to handle these scenarios.

Again, from the [DDD example project](#).

```

import { PathParams, rest, RestRequest } from 'msw';

const SECURITY_API_ENDPOINT_GENERATE =
  'https://RANDOM.execute-api.eu-north-1.amazonaws.com/shared/↵
  ↵ generateCode';

const MsgSetMswInterceptedDataFromApi = (url: string) =>
  `[MSW] - Mocking intercepted fetch request data from API: ${url}`;

const logInterceptedRequest = (req: RestRequest<any, PathParams>) =>
  console.log(MsgSetMswInterceptedDataFromApi(req.url.href));

export const handlers = [
  rest.post(`${SECURITY_API_ENDPOINT_GENERATE}`, (req, res, ctx) => {
    logInterceptedRequest(req);
    return res(ctx.status(200), ctx.text('1234'));
  }),
  rest.post(`${SECURITY_API_ENDPOINT_GENERATE}-fail`, (req, res, ctx) ↵
    ↵ => {
    logInterceptedRequest(req);
    return res(ctx.status(400), ctx.json({}));
  })
];

```

Above, you see how we intercept the POST request to both one of our API endpoints, as well as a faked endpoint ending with `-fail`. For the first one, we respond with a status 200 and some garbage text, and for the other one, we send a status 400 (user issue) and an empty object.

In closing

Unit tests are super versatile and once you get the hang of dealing with some of the “boundary” issues, like HTTP calls and mocks, you should be on a good track in your testing prowess.

Smoke testing

Information

Surface area Facade of your deployed service **Confidence level** Medium **Granularity** Low **Pros**

- Cheap and simple
- Very close to “the real deal”

Cons

- Side effects from using an actual system
- Errors aren’t necessarily directly debuggable

Success

When to do this type of testing? Always, as needed.

[Wikipedia](#) explains smoke testing as

preliminary testing to reveal simple failures severe enough to, for example, reject a prospective software release.

[Microsoft’s Engineering Playbook](#) expands on this:

Smoke tests **cover only the most critical application path, and should not be used to actually test the application’s behavior, keeping execution time and complexity to minimum.** The tests can be formed of a subset of the application’s integration or e2e tests, and they cover as much of the functionality with as little depth as required.

The golden rule of a good smoke test is that it saves time on validating that the application is acceptable to a stage where better, more thorough testing will begin. [...]

Smoke testing is a low-effort, high-impact step to ship more reliable software. It should be considered **amongst the first stages to implement** when planning continuously integrated and delivered systems.

Smoke tests are a rough type of test that assesses whether or not something is working as expected. Therefore, smoke testing is really nothing more than a thin slice of integration tests. *Smokes* are ideally short-running and technically simple.

Information

I would therefore recommend that smoke tests are broad, functionally-oriented tests rather than (pointing to the Wikipedia article) on any *unit size* in scale—prefer other, faster, and simpler tests for such cases.

Personally, I've not worked with any large number of smoke tests. The way I've approached them, as compared to integration/API tests, is that smoke tests are very simple and make only basic assertions whereas integration/API tests can be more granular if needed. The technology itself is in both cases the simplest possible that makes sense. For *smokes* the simplest would be something like `curl` because the nature of the assertion is decidedly primitive.

Here's an example of a callable Bash (or shell) script testing an online API:

```
#!/bin/bash
set -e

# RUN SMOKE TEST
# Expects input parameter with an URL
# Checks for status code 200

URL=$1
STATUS_RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" $URL)

if [ "$STATUS_RESPONSE" == 200 ]
then
```

```
    echo Worked just fine && exit 0
else
    echo Yep that broke something && exit 1
fi
```

Calling it would look something like:

```
bash smoketest.sh https://www.acmecorp.xyz
```

Remind yourself of the fact that these only attempt to give you a very broad sense of whether or not something worked or not!

You still need to handle side effects

The problem with using any “real” system is always that you will get side effects so any advice on implementing your system in a way that intelligently takes care not to propagate test runs to production systems is applicable here too.

API (integration) testing

Information

Surface area Facade of an actual, deployed service or API (your service or some external service) **Confidence level** Medium; good enough if you trust there are no adverse side effects or similar **Granularity** Low, you know it works and does what should do for you as a consumer **Pros**

- Probably the most intuitive and easy way to actually test something

Cons

- Side effects from using an actual system

Success

When to do this type of testing? Your services: Any time you are building something that provides an API, then consider a low degree of these tests to verify complete upfront functionality. External services: Always mock these, if for no other reason than because your unit tests will be able to do any outside calls without actually creating side effects.

Let it be known that the classic notion of integration testing—testing several software components at the same time—is practically useless. That could just be covered by a wider unit test.

Information

A related notion is the one of [sociable vs solitary tests](#).

For the intents and purposes of this chapter, we will think of integration testing and API testing as essentially the same: We test something “from the outside”, whether that’s an

HTTP API or something else matters less.

Because we are going to use an actual system we don't really have to do more than simply use it! It wouldn't *not* be testing if you'd use Postman or Insomnia on every deployment to call an API with some prepackaged payloads, however, we'd of course not have it automated either. So let's discard that idea even if it's valid to some extent in factual terms.

Running tool-centric tests

What you can do, however, with Postman and Insomnia is to use their CLI counterparts to run the tests during CI:

- [Postman, using newman](#)
- [Insomnia, using inso](#)

I don't have any deeper experience with these, but it's definitely possible and makes sense if you already use such API clients to store payloads.

Moving on to our DIY solution...

Writing our own integration testing tool

To do a bit of lightweight integration testing we need two things:

- Some kind of evaluator engine.
- A set of assertions to test.

For the evaluator, we'll make life a little easier by using [Ajv](#) to perform JSON validation on our behalf, which could otherwise easily escalate into a major pain. While `fetch` is nowadays native in Node 18 and upwards, for compatibility reasons we will also pull in [node-fetch](#) to handle requests.

Danger

There are breaking changes between versions 2 and 3 of `node-fetch`. If you are using Webpack to bundle your application, consider using the most recent 2-series version of `node-fetch` or you may face bundling errors.

While this isn't strictly zero-dependency, it's lightweight enough and allows us the required flexibility to test *actual* APIs with payloads or assertions that we define without surrendering to specific testing frameworks.

Let's browse the code.

```
// import fetch from 'node-fetch'; ---> Only needed when using less ↔
↔ than Node 18
import Ajv from 'ajv';

import { assertions } from './assertions';

const INTEGRATION_ENDPOINT = 'https://my-web-api.acmecorp.xyz';
const AUTH_TOKEN = 'something-here';

async function runIntegrationTests() {
  if (!INTEGRATION_ENDPOINT) throw new Error('Missing ↔
  ↔ INTEGRATION_ENDPOINT!');
  let testsFailed = false;

  const tests = assertions.map(async (assertion: any) => {
    return new Promise(async (resolve, reject) => {
      const { name, payload, schema, expected } = assertion;
      const { method, path, headers, body, urlParams } = payload;

      // Use auth header if needed
      headers.Authorization = AUTH_TOKEN;

      console.log(`Running integration test: "${name}"`);

      const response = await fetchData(
        `${INTEGRATION_ENDPOINT}/${path}`,
        headers,
        method,
        body,
        urlParams
      );
    });
  });
}
```

```

    );
    if (!response) throw new Error('No response!');

    /**
     * If there is an Ajv matching schema use that to check,
     * else use an exact comparison to check.
     */
    const isMatch = schema
      ? test(schema, response)
      : JSON.stringify(response) === JSON.stringify(expected);
    if (isMatch) resolve(true);
    else {
      testsFailed = true;
      reject({ name, response });
    }
  });
});

Promise.all(tests)
  .catch((error) => error)
  .then((result) => {
    if (testsFailed) {
      console.log(
        `Failed integration test: "${result.name}" --> ${JSON.↵
↵ stringify(result.response)}`
      );
      process.exit(1);
    } else {
      console.log('Passed all integration tests');
    }
  });
}

/**
 * @description Wrapper for fetching data.
 */
async function fetchData(
  url: string,
  headers: Record<string, any>,
  method: 'POST' | 'PATCH' | 'GET',
  body: any,
  urlParams: Record<string, any>
): Promise<any> {
  /**
   * If we have `urlParams` (which we can infer meaning that it's a GET↵

```

```

    ↪ case), then
    * manually spread these (known) properties first into a full URL.
    *
    * Else just use it as-is.
    */
const fetchUrl = urlParams
  ? `${url}${getParamsString(urlParams)}`
  : url;

const response = await fetch(fetchUrl, {
  headers,
  body: body ? JSON.stringify(body) : undefined,
  method
});

// If this is OK and status 204 ("No content") then we can safely ↪
    ↪ return
if (response.ok && response.status === 204) return 'OK';

const text = await response.text();

// Return text or JSON depending on what it actually was
try {
  const data = JSON.parse(text);
  return data;
} catch (error) {
  return text;
}
}

const escapeString = (value: any) => {
  if (typeof value === 'string') return value;
  return JSON.stringify(value).replace(/\\s/g, '%20').replace(/"/gi, '\\↪
    ↪ ');
};

const getParamsString = (urlParams: Record<string, any>) =>
  Object.entries(urlParams).reduce(
    (previousValue: [string, any], currentValue: any[], index: number):↪
    ↪ any => {
      let paramValue = index === 1 ? `?` : `${previousValue}&`;

      // On the first run this will include the "zeroth" value
      if (index === 1) {
        const [key, value] = previousValue;

```

```

        paramValue += `${key}=${escapeString(value)}&`;
    }

    const [key, value] = currentValue;
    paramValue += `${key}=${escapeString(escapeString(value))}`;

    return paramValue;
}
);

/**
 * @description Run a test by validating a schema with Ajv.
 */
function test(schema: any, data: any): boolean {
    const isArray = Array.isArray(data);
    if (isArray) data = data[0]; // Use the first item in an array if ↔
    ↔ this is one

    const ajv = new Ajv();
    const validate = ajv.compile(schema);
    const isValid = validate(data);

    return isValid;
}

runIntegrationTests();

```

Information

Feel free to copy the code, it's open source on my GitHub Gist.

The critical parts are:

- `runIntegrationTests()` which orchestrates the overall functionality.
- `fetchData()` does just that.
- `test()` uses `ajv` to compile and validate the provided expected schema with what we got

Note that in the above code we also check if the response is simply “OK” in which case it is accepted as a match (for cases in which you may not actually receive any content, such as with [status code 204](#)).

The assertions are in a custom, though fairly flexible, format in which we can set our test names and payloads for any HTTP method.

```
export const assertions = [
  {
    name: 'It should DO SOMETHING',
    payload: {
      method: 'POST',
      path: 'DoSomething',
      headers: {
        'X-Client-Version': 1
      },
      body: {
        userName: 'Sam Person',
        actions: [
          {
            Id: '2n022yd',
            ActionType: 'CONFIRMED'
          }
        ]
      }
    },
    schema: {
      type: 'object',
      properties: {
        systemId: { type: 'string' }
      },
      required: ['systemId'],
      additionalProperties: false
    }
  },
  {
    name: 'It should UPDATE SOMETHING',
    payload: {
      method: 'PATCH',
      path: 'UpdateSomething',
      body: {
        id: 'abc123',
        newValue: 'qwerty',
      }
    }
  }
]
```

```
    },
    expected: 'OK'
  },
  {
    name: 'It should GET SOMETHING',
    payload: {
      method: 'GET',
      path: 'GetSomething',
      urlParams: {
        systemId: 'something',
        user: 'something'
      }
    },
    schema: {
      type: 'object',
      properties: {
        id: { type: 'string' },
        version: { type: 'number' },
        hasDoneSomething: { type: 'boolean' }
      },
      required: ['id', 'version', 'hasDoneSomething'],
      additionalProperties: false
    }
  }
];
```

Using a body means we pass that actual body in a POST request, while `urlParams` are used if we need parameters passed in the URL rather than as a request body.

Lastly, the schema is your everyday JSON Schema to validate with `ajv`.

Are you integration testing someone else's thing?

Doing API (integration) testing in CI of *other's* services does not make logical sense, as this could lead to cases in which your deployment failed because of a failure *elsewhere*. And if you are thinking that “well, I mean if the *other* thing ain't working then there's no real use for me to deploy, right?” then the answer is clear: That's plain mixing it up in the wrong way.

Remember that a test is always of *some unit*, *performing something at that specific moment*. It is

not a guarantee for correct functionality at any later point, and especially not if other factors are more fluid, such as completely external dependencies and APIs, and that includes any non-application layer (i.e. network, etc.).

In closing

Hopefully, this demystifies integration/API testing a bit for you, and that you see why I argue so much more passionately for unit tests.

Synthetic testing

Information

Surface area Facade of your deployed service **Confidence level** Medium **Granularity** Low **Pros**

- Roughly the same benefits as smoke testing
- Relatively easy to automate today with decent tooling
- Given correct thresholds and alerts this will give the only (so far) solution that sees the solution performing as expected over a longer timespan than conventional testing

Cons

- Roughly the same drawbacks as smoke testing
- Not well-understood
- Not standardized or strong conventions in place
- You should design your system to accept synthetic input without creating adverse side effects unless you really do want them

Success

When to do this type of testing? If you are very interested in knowing the continued functionality of something, and if there is considerable reason to believe the functionality is on a “sliding plane” of working or not, then you should consider this type of testing. Also, if you don’t have consistent traffic or need to fake some traffic, then this might be a good option for you.

Synthetic testing sounds really cool but it’s, as always, not quite so exciting. It’s no more than plain old smoke testing with some small but important differences.

Synthetic testing and smoke testing are somewhat similar in that they are typically very basic assertions of an actually exposed system. Both types of testing can easily be done on a machine that you control or from a third party that provides such testing as a service. They are equally poor at addressing debuggable aspects as their respective responses will be no more or less than status codes or similar. Their commonality extends to both using “real systems” so you should be able to trust that both types of tests give you a fair sense of confidence.

The relative problems around synthetic testing seem to me to be mainly conceptual, rather than technical.

Where a smoke test is done *at the time of testing* it cannot guarantee the continued functionality of a system. Using a scheduled synthetic test we can actually continuously verify it! Once again, let’s turn to [Microsoft’s Engineering Playbook](#) for some wisdom on how things change when we add a time aspect to the testing pyramid model:

However, as more organizations today provide highly-available (99.9+ SLA) products, they find that the nature of long-lived distributed applications, which typically rely on several hardware and software components, is to fail. Frequent releases (sometimes multiple times per day) of various components of the system can create further instability. This rapid rate of change to the production environment tends to make testing during CI/CD stages not hermetic and actually not representative of the end user experience and how the production system actually behaves.

For such systems, the ambition of service engineering teams is to reduce to a minimum the time it takes to fix errors, or the [MTTR - Mean Time To Repair](#). It is a continuous effort, performed on the live/production system. Synthetic Monitors can be used to detect the following issues:

- **Availability** - Is the system or specific region available.
- **Transactions and customer journeys** - Known good requests should work, while known bad requests should error.
- **Performance** - How fast are actions and is that performance maintained through high loads and through version releases.
- **3rd Party components** - Cloud or software components used by the system may fail.

— Microsoft: Code With Engineering Playbook, [Synthetic Monitoring Tests](#)

By probing our applications with what seems to be actual, human traffic we can do more than just see “it worked” and start seeing it in more realistic, elaborate circumstances, such as for complex user flows. This is especially interesting in case you have an application in which there is low or even no real traffic, or if you have bursty traffic and need to verify things work in a low-traffic period.

Failures still need to be caught in your observability tool and applications, as ever, need to be built to a standard that affords rich and concise observability, i.e. making it possible for engineers to understand what went wrong, when, where, and why.

Success

Now would be a good time to also remind ourselves of how testing and observability are not the same thing but need to co-exist if we are to deliver applications at any acceptable modern level. If the probe fails we should expect that our tooling to alert our team of this unexpected, negative fallout.

But, why probe applications that have steady traffic? The perhaps most enticing reason would be that it should be easier to detect a direct relation between a failing test (that you control) and the error that occurred, which may be logically harder to pinpoint from an actual user error. Your mileage may vary, of course. If you are not making troubleshooting *easier* with these tests then you have more fundamental problems to address.

Information

Read more about synthetic testing at:

- [Google SRE Book: Testing for Reliability](#)
- [Azure DevOps: Shift right to test in production](#)
- [Martin Fowler: Synthetic Monitoring](#)

If you are interested in setting up synthetic monitoring I would recommend a nice pay-as-you-go option such as [Checkly](#) or your cloud-native offerings such as [AWS CloudWatch](#)

Synthetics. These both provide either basic checks (“pings”) for basic system-level interactions or actual UI end-to-end testing for anything where user flows need to be tested.

Information

I was using AWS’ synthetic canaries some time ago and [wrote an “improved” variant of how it does things](#) as I hit limits on how efficiently it would crawl complex websites. It may still offer some value to you if you get any such issues.

Comparing to RUM (real user monitoring)

This is a dead giveaway when you just weigh “real user” and “synthetic” against each other.

But really, let Mozilla explain:

Synthetic is well suited for catching regressions during development life cycles, especially with [network throttling](#). It is fairly easy, inexpensive, and great for spot-checking performance during development as an effective way to measure the effect of code changes, but it doesn’t reflect what real users are experiencing and provides only a narrow view of performance.

RUM, on the other hand, provides real metrics from real users using the site or application. While this is more expensive and likely less convenient, it provides vital user experience data.

— [Mozilla: Performance Monitoring: RUM vs synthetic monitoring](#)

An example of a relatively lightweight RUM product could be [Cloudflare Web Analytics](#) or you could look at bigger commercial products like those in [Datadog](#), [New Relic](#), and [Akamai](#). Your exact choice will most likely depend heavily on what other network and/or monitoring tools you have in place.

In closing

There are interesting use cases for this type of testing, but you should probably first ensure that your team has tightened the bolts of other tests before progressing to this type, also given its tad more academic value.

Load testing

Information

Surface area Facade of an actual, deployed service or API **Confidence level** Medium, given a more “fuzzy” test this will be somewhat more confidence-building than a straight API test **Granularity** Low, you know it works and does what should do for you as a consumer **Pros**

- Good tests will give you confidence that the system (and any behind it) can stand up to more traffic; you can also start doing more fuzzy testing to ensure problematic inputs work as expected

Cons

- The intensity of the load test means that you have to be even more careful that there are no adverse side effects created as a result

Success

When to do this type of testing? Typically when you want to pressure test an application. If you cannot confidently run an API test for your service, then you should not even consider this class of test.

The overall category of “load testing” comes in many sizes but generally, the same taste, as evidenced by load testing tool [k6](#)’s introduction to various test types:

Smoke Test’s role is to verify that your system can handle minimal load, without any problems.

Load Test is primarily concerned with assessing the performance of your system in terms of concurrent users or requests per second.

Stress Test and **Spike testing** are concerned with assessing the limits of your system and stability under extreme conditions.

Soak Test tells you about reliability and performance of your system over an extended period of time.

The important thing to understand is that each test can be performed with the same test script. You can write one script and perform all the above tests with it. The only thing that changes is the test configuration, the logic stays the same.

Different test types will teach you different things about your system and give you the insight needed to understand or optimize performance.

— k6

For the purposes of this book, load tests follow naturally from where smoke and synthetic tests left off, with us this time cranking up the volume of traffic and generally moving to more potent tooling.

Information

For Node.js, some of the more popular open-source tooling in this area—besides k6—are [Artillery](#) and [Gatling](#).

A k6 script for an example made for the [Better APIs book and workshop](#) looks like below. Here we are attempting to get random variants of users and API versions and see that they will all work without throwing weird errors at us. The volume of traffic will be 10 virtual users looping on the same endpoint for 5 seconds.

```
import http from 'k6/http';
import { sleep } from 'k6';

// Setup
const endpoint = 'https://RANDOM.execute-api.REGION.amazonaws.com/↵
↵ shared/fakeUser'; // TODO: EDIT THIS TO YOUR ENDPOINT
const randomUser = getRandomUser();
const randomVersion = getRandomVersion();

/**
 * @description Helper to get one of the users.
 */
export function getRandomUser() {
```



```

const getRandomInt = (min, max) => {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min) + min); //The maximum↔
  ↪ is exclusive and the minimum is inclusive
};

const users = [
  'erroruser@company.com',
  'legacyuser@company.com',
  'standarduser@company.com',
  'betauser@company.com',
  'qauser@company.com',
  'devuser@company.com',
  'devnewfeatureuser@company.com'
];

return getRandomInt(0, users.length - 1);
}

/**
 * @description Helper to get a random version (or none)
 */
export function getRandomVersion() {
  const random = Math.round(Math.random() * 100);
  if (random <= 50) return 1;
  if (random > 50 && random <= 100) return 2;
}

/**
 * K6 configuration
 */
export const options = {
  vus: 10,
  duration: '5s'
};

const params = {
  headers: {
    Authorization: randomUser,
    'X-Client-Version': randomVersion
  }
};

```

```
export default function () {  
  http.get(endpoint, params);  
  sleep(1);  
}
```

Success

Could you do all of the smoke testing and API testing with k6? Yes, that could absolutely work, especially since they provide competent tools for setting thresholds and such for results. However, I like to use the ideal, but most minimal, tool needed to do the job so that's why I show you examples in line with that belief.

The tiny load provided here fulfills mostly a demonstrational purpose. For a real load test, you might instead consider using a [scenario](#) to add an increasing amount of load for a few minutes.

Tools like these are brilliant as they are easy to use and script and work well in CI as well.

Information

Limits on network load Every machine has natural restrictions on the amount of network load they can generate. For *very high* loads you may need to use a distributed solution for load generation. Some examples include:

- Running a fan-out of load generation on Lambda;
- Running tests from a high-powered virtual machine (since they typically get higher network speeds as well); or
- Using the SaaS versions of k6 or Artillery or similar.

As always, know *why* you are conducting a test. What load limits do you have, or what hypothesis are you having around this? Also, keep in mind any elasticity you might have in your infrastructural setup; for example, it seems relatively awkward to test on a server-

less compute platform that has automatic scaling, unless you have a clear hypothesis that the platform can't handle *spikes*, for example.

System testing

Information

Surface area Theoretically the entirety of a system **Confidence level** High **Granularity** Low-medium **Pros**

- Cheaper and faster than the UI-based approach
- Possibility to slice and scale as needed

Cons

- Similar drawbacks to UI end-to-end testing
- Most likely the slowest test of any category
- Requires a fully deployed and functional system or set of systems
- Will create side effects
- Requires writing custom logic as there is no conventional or easy tool-based way to exercise a system (or set of systems) in the precise manner you will want

Success

When to do this type of testing? Consider doing this only if you have these circumstances:

1. There is a significant “entry” component that exercises multiple “deep” or unknown backend systems so you know precisely that the required systems are in scope.
2. You have a very clear end state for success or failure, ideally without relying on privileged access to databases or similar for verification.
3. You are able to use a stable pre-production environment:

1. Without contaminating environments with side effects (generated test data);

Information

The nomenclature regarding system vs end-to-end tests can be confusing. A meaningful distinction would be that, commonly, system testing is for the entirety of a single system, whereas end-to-end testing could encompass multiple systems. Also, it's at least implicit in many contexts that end-to-end tests are primarily driven through visual means, such as through the user interface. However, I don't necessarily agree that it's a hard and fast rule that a system test could not interact with more than a single system. Instead, the bigger problem is: Where is the ownership (and code!) located for a huge test (or set of tests) that is not restricted to a clean-cut, perfectly isolated service?

System testing seems to go into the bucket of tests that people hope will cover everything: “We need to ensure all of it works together!”. While I personally see a lot more traction regarding UI end-to-end testing, this one can be useful to have some mental construct for.

We can immediately assess that this is a **functionally oriented** test type—we care about our particular use cases and that they work as expected. As opposed to other lightweight tests, this test will indeed use actual infrastructure. The test will most likely not happen in a production environment, but absolutely could if you want it to.

I'd recommend seeing the system test as a “fully functional”, API-driven backend-oriented way of testing expectations. Focus on raw functionality, sort of like an extended integration test. For example, in a system that has four components that are sequentially called and end with data in a database, you could call the main entry point and verify that data ended up correctly stored.

It's easy to hate a type of type that's as extreme as this one, but there are *some* good sides:

- It enables a high degree of confidence.

- It will (hopefully) function exactly as expected.
- Having one or more system tests can help codify the *totality* of a use case (in a bizarre sense improving technical documentation!).
- By automating this/these tests and crafting them well, you can indirectly improve your stance on disaster recovery and full-environment recovery.

A partial example of a system test

In this example from a [microservices testing workshop](#) I ran, we have an order system that we want to verify completely, from the external-facing API Gateway to fully formed orders residing in the order database.

The solution will be to use the exposed endpoint, calling it with fake data, and then verifying that the exact number of items exist and look as expected. Even in this extreme test, we do not need to check things like if the AWS libraries work if an event bus got a message, or if it will be raining tomorrow—we *know* that it works because *any failure* along the way will mean we are unable to correctly retrieve the data.

```
import { verifyData } from './verifyData';
import { createOptions, callService, createDummyOrder, sleep, ↵
    ↵ generateTestId } from './utils';
import { getEndpointConfig } from './config';

/**
 * Environment
 * Update this to your values
 */
// API Gateway ID
const ID = 'abcde12345'; // {{UNIQUE_ID}}
// AWS region
const REGION = 'eu-north-1';
// API Gateway stage
const ENV = 'dev';

/**
 * Test configuration
 * Modify if you want, or just leave it be
 */
// ID for this specific test run
```

```

const TEST_ID = generateTestId();
// Time to wait until first verification
const WAIT_TIME = 5000;
// Time to wait until running next iteration in call-loop
const LOOP_WAIT_TIME = 150;
// How many test orders to produce
const TEST_COUNT = 25;
// Test can be an empty object, in which case defaults will be used
const TEST_CONFIG = {};

/**
 * @description Full system test, starting with a number of test orders↵
 * ↵ and verify presence in order database at the end.
 */
async function SystemTestController(testCount = 1, customerData = {}) {
  console.log('Generated test ID is', TEST_ID, '\n');

  try {
    /**
     * Get endpoints for our environment
     */
    const { createOrderServiceEndpoint, getOrdersServiceEndpoint } = ↵
    ↵ getEndpointConfig(
      ID,
      REGION,
      ENV
    );

    /**
     * Loop-call service
     */
    for (let i = 1; i <= testCount; i++) {
      const order = createDummyOrder({ ...customerData, testId: TEST_ID↵
      ↵ });
      const resp = await callService(createOrderServiceEndpoint, ↵
      ↵ createOptions('POST', order));
      await sleep(LOOP_WAIT_TIME);
      console.log(i, order, resp, '\n');
    }

    /**
     * Wait for events to have settled a bit
     */
    await sleep(WAIT_TIME);
  }
}

```

```

/**
 * Get data
 */
const dbData = await callService(
  getOrdersServiceEndpoint,
  createOptions('POST', {
    testId: TEST_ID
  })
);

console.log('Count of items is', dbData.Count);

/**
 * Verify
 */
verifyData(dbData, testCount, TEST_ID);
} catch (error) {
  throw new Error(error);
}
}

// Run tests
SystemTestController(TEST_COUNT, TEST_CONFIG);

```

Data is tagged as test data and the application is rigged to never read any of the generated data.

We use a [test data builder](#) pattern, calling `createDummyOrder()` to create valid order data, and utilizing the [Faker library](#) to set parts of our data randomly and dynamically.

```

/**
 * @description Create (test) order
 */
export function createDummyOrder(customerData) {
  let {
    name,
    email,
    phone,
    street,
    city,
    customerType,
    market,
    products,

```



```

    totalPrice,
    orgNumber,
    testId
  } = customerData;

  const MARKET_US_QUOTA = 50; // US vs MX quota
  const CUSTOMER_TYPE_B2B_QUOTA = 20; // B2B vs B2C quota

  market = market
    ? market
    : (() => {
      const chance = Math.round(Math.random() * 100);
      if (chance <= MARKET_US_QUOTA) return 'US';
      return 'MX';
    })();

  customerType = customerType
    ? customerType
    : (() => {
      // Mexico is B2C-only
      if (market === 'MX') return 'B2C';

      const chance = Math.round(Math.random() * 100);
      if (chance <= CUSTOMER_TYPE_B2B_QUOTA) return 'B2B';
      return 'B2C';
    })();

  if (market === 'US') faker.locale = 'en_US';
  if (market === 'MX') faker.locale = 'es_MX';

  orgNumber = (() => {
    if (customerType === 'B2C') return 0;
    return orgNumber
      ? orgNumber
      : faker.random.number({
        min: 6,
        max: 20
      });
  })();

  const firstName = name ? name : faker.name.firstName();
  const lastName = name ? name : faker.name.lastName();
  email = email ? email : faker.internet.email();
  phone = phone ? phone : faker.phone.phoneNumber();
  street = street ? street : faker.address.streetAddress();

```

```
city = city ? city : faker.address.city();
products = products ? products : 'BB001,BA002';
totalPrice = totalPrice ? totalPrice : parseInt(Math.round(Math.↵
    ↵ random() * 1500) + '00');

const customer = {
  name: `${firstName} ${lastName}`,
  email,
  phone,
  street,
  city,
  customerType,
  market,
  products,
  totalPrice,
  orgNumber,
  testId
};

return JSON.parse(faker.fake(JSON.stringify(customer)));
}
```

As you see, the majority of the above code is simply the overall test controller function. For this particular system, it will create a configurable number of dummy orders, call the order service and pass in the data and cycle this loop for the number of times we need it to do the work.

Information

Dedicated functionality for verifying and creating data isn't part of the above snippet as these are entirely dependent on what exactly is required. For more on this, you can check out the full code in the [associated GitHub repository](#).

After having sent the data we will wait a short time to let the system settle and then call an endpoint that will retrieve our orders with the unique test ID. Our table structure is set up in a way in which items are possible to retrieve by this ID and by sharing it across multiple items for testing purposes we could theoretically do this test in a production environment without too much risk.

In closing

System tests are like taking a tank to a gunfight. Yes, it's powerful, and not quite that different from integration tests, but it comes at the pretty steep cost of *touching everything*. As far as possible, gain the needed confidence by other means. But if you do it: Make it as slick and lean as you can!

UI end-to-end testing

Information

Surface area Theoretically the entirety of a system, but most critically its UI layer
Confidence level Medium-high
Granularity Minimal
Pros

- Gives a user-oriented and functional understanding of whether or not flows and functionality work as intended
- Tooling is getting better so there are some tools today that are easy to use and work well

Cons

- Cumbersome to write, many steps needed to reproduce a user flow
- Will use actual hardware and software so there will be side effects
- Notoriously brittle tests
- Encourage incorrect test boundaries (i.e. testing things that change naturally over time)
- Expensive
- Long test times
- If something breaks or fails, there is effectively no good way to granularly understand *why* it broke
- Depends on all or most of a system to be deployed and functional
- Many older tools are particularly brittle and poor in DX

Success

When to do this type of testing? Consider doing this only if you have these circumstances:

1. There is a significant UI component that exercises multiple “deep” or unknown backend systems that you do not control.
2. You have a very clear end state for success or failure, ideally without relying on privileged access to databases or other infrastructure for verification.
3. Your team has more experience with UI-based testing than programmatic testing.
4. You are able to use a stable environment:
 1. Without contaminating environments with side effects (generated test data);
 2. Without requiring multiple teams to co-deploy and/or induce code freeze or environment freeze.

End-to-end testing (“E2E”) is the customer-facing , often graphical, part of the flow: In other words, just as a user will experience it. Focus therefore on user flow. That means setting up *automated* browser testing.

If you just came from the previous page about system testing, I explained that UI end-to-end testing and system testing share in common that they use the full scope and real infrastructure to assess functionality. Where system testing leaned toward using the extremes of the system landscape—such as the API as an entrypoint and the database as the final output—E2E will do much the same but through the same means as the user has to their disposal: That is, through the user interface.

While frontend-centric testing has not been given a lot of place in this book, it’s smart to keep in mind that the same overall model and guidance is valid here too:

- **Logic in your app should use unit tests.** It’s not uncommon to see front-end developers bake logic into hooks and other “harder to test” parts of their apps. As always, appropriate tooling here includes [Jest](#), [Ava](#), [React Testing Library](#) and even the [Node 18+ native test runner](#). Really: The dumber => the faster => the better, when it comes to this category.

- Use [snapshot testing](#) or [visual testing](#) for the visual components. Of course, *not* testing the visuals would never work in the front-end world! This is well-worth it, especially if you have a [good review process in place](#).
- **Keep E2E tests, integration tests etc. at a minimum, i.e. as per the “top of the pyramid”.** If you’ve already ensured that the logic is correct, that your visuals aren’t broken, and the required testing is done in the API or back-end layer, why would you need to do more? If you have a compelling answer, then by all means add a dash of these here, but don’t do it unless you know exactly what you are hoping to uncover.

E2E and its bad reputation

So why exactly is E2E problematic in terms of testing?

Firstly, it comes down to the key premise that a test should use the simplest form of verification possible without sacrificing confidence. I’ve spent quite some time in this book arguing how even quite complex phenomena can be tested accurately with “simple” testing types like unit tests. Testing closer to the code also gives you a better understanding of its failure modes.

Secondly, E2E tests are (in theory) no more than scripted manual user interactions. In fact, usually, the E2E testing tool will fire up a browser and quite literally reproduce the steps you’ve given it. Even if this is a computer doing the dirty work, you have all of the same waiting for the process to start, for things to load, for things to get ready, for performing interactions, and so on. Also, it’s not uncommon that tests become brittle as UIs can change a lot (*high volatility*), even without the functional flow or requirements changing at all. Tests that are ill-devised will break on these otherwise unproblematic changes.

On the flip side, when done well and selectively, they are a very close approximation of the actual, real user experience. Also, using modern tools like Codecept make this form of testing more stable than before and we can even run it “headless” to make it faster and less dependent on the visual elements (as there is no human in the loop, anyway!).

Importantly, from our test models, we need to recall and respect the highly selective approach to these tests, as well as not simply copy-pasting tests that are already confidently covered in better, easier, and cheaper tests.

Does it ever make sense to *start* with E2E tests? Well, given a situation in which there is a heavy—and maybe even troublesome—separation between a front-end team and the API or back-end team—and especially if the overall solution has no (or very low) test coverage—it may be an efficient and somewhat easy move for the front-end team to set up E2E testing. This could cover the key flows if the API is not documented or easily testable, and if there is no other surface area that can have deep, informed testing; Unit testing and any of the “cheaper” and more exact tests require you to have direct access to it. Without access, and certainly, if the code is in someone else’s responsibility and/or domain, then you might need these other mechanisms of ensuring confidence. In this situation, **we could think of end-to-end-testing being a transitional form of activity, rather than the ultimate end goal of any testing.**

Using Codecept to do UI testing

Traditionally [Selenium](#) has been a big name, but a lot has happened in recent times (once again), so consider [Codecept](#), [Playwright](#), [Cypress](#), or other newer tooling. For our own example, it’ll be using Codecept.

The following example is from my [microservices testing workshop](#) on GitHub.

We have a feature in the system which is called “Order items”. The first scenario we will use is named “Order single item as US B2C customer”. Give the following a read and see if you can figure out what’s going on!

```
Feature("Order items");

Scenario("Order single item as US B2C customer", ({ I }) => {
  I.amOnPage("http://127.0.0.1:8000");

  I.seeElement('//button[contains(., "300 $")]');
  I.click("300 $");
  I.see("1", "#cart-itemcount");
  I.click("#cartsymbol");

  I.fillField("#input-name", "Anders Andersson");
  I.fillField("#input-street", "Thisway 123");
  I.fillField("#input-city", "Göteborg");
  I.fillField("#input-email", "someone@nowhere.xyz");
  I.fillField("#input-phone", "123123123123");
```

```
I.click("Submit order");  
I.see("Thank you for your order!");  
I.seeInCurrentUrl("/success");  
});
```

As you see, the format is very close to what you would expect of [behavior-driven development \(BDD\)](#). That's definitely a nice thing as it makes for more concise and readable code. In this example, you've probably understood that we are acting on a locally running copy (telltale sign in the URL, 127.0.0.1:8000) and that our script is performing a sequence of activities:

- Finding a button element with a value, 300 \$
- Clicking the button
- Seeing the status of the cart inventory update
- Clicking the cart symbol
- *(we get moved to a second page)*
- Fill in several fields
- Click to submit the order
- Verifying the finished state by checking the confirmation message and final URL which has now changed

Once the script is running, the browser (headless or regular) will receive the programmed inputs and the test will break if something is not possible to do, or if the assertion is not met.

Good or bad demonstration?

Well, you might ask, is this a good or a bad test? I would argue that this one is serviceable but not more than that. We still have a number of highly volatile assertions—such as the exact contents of texts and the exact classes or IDs of fields—and we could fail on incredibly minor changes. Also, since modern frameworks and bundlers often have

auto-generation of CSS class names, they might not even be easily accessible or stable over time. While this can be solved as we did here, using IDs (which don't get replaced), I think it should be even clearer now for you how E2E tests dramatically decrease the stability of the underlying test compared to most of the other test types we've seen, already in a basic demonstration such as this one.

As far as I am concerned, the biggest thing we might have wanted to do in order to make the test better is to remove any assertions regarding exact text contents, as these are always more volatile than IDs.

Remaining scenarios

Next up, we'll add the two remaining scenarios we have in the application.

```
// Continuing on the first example
Scenario("Order single item as US B2B customer", ({ I }) => {
  I.amOnPage("http://127.0.0.1:8000");

  I.seeElement('//button[contains(., "150 $")]');
  I.click("150 $");
  I.see("1", "#cart-itemcount");
  I.click("#cartsymbol");

  I.selectOption("#customer-type", "B2B");
  I.fillField("#input-orgnumber", "123123123");

  I.fillField("#input-name", "Anders Andersson");
  I.fillField("#input-street", "Thisway 123");
  I.fillField("#input-city", "Göteborg");
  I.fillField("#input-email", "someone@nowhere.xyz");
  I.fillField("#input-phone", "123123123123");

  I.click("Submit order");
  I.see("Thank you for your order!");
  I.seeInCurrentUrl("/success");
});

Scenario("Order single item as MX B2C customer", ({ I }) => {
  I.amOnPage("http://127.0.0.1:8000");

  I.seeElement('//button[contains(., "150 $")]');
```

```
I.click("180 $");
I.see("1", "#cart-itemcount");
I.click("#cartsymbol");

I.selectOption("#country", "MX");

I.fillField("#input-name", "Anders Andersson");
I.fillField("#input-street", "Thisway 123");
I.fillField("#input-city", "Göteborg");
I.fillField("#input-email", "someone@nowhere.xyz");
I.fillField("#input-phone", "123123123123");

I.click("Submit order");
I.see("Thank you for your order!");
I.seeInCurrentUrl("/success");
});
```

Information

It's possible all of these three scenarios could somehow be more DRY and programmatic, as the majority of their steps are similar. At least they are pretty easy to follow.

Nevertheless, we can see the neatness that a well-functioning E2E test can exert, as well as providing a very good sense of the software doing what it should.

In closing

Given adequate DRYness, precise use of tests, selection of a good tool, and asserting stable (low-volatility) behaviors, then end-to-end testing can definitely work as an OK transitional test type, at least until we can push toward lower-level testing. Don't stop with this!

Contract testing

Information

Surface area The schema or a representation of a schema for one or more external systems **Confidence level** Variable **Granularity** Low, you know it *should* work **Pros**

- Significantly less risky than API/integration testing
- No side effects
- Fast

Cons

- Not well-understood
- Not standardized or strong conventions in place
- May add extra work
- If the contract is separately handled from the service's schema and/or implementation, there is a non-zero risk of these being different and thus leading to wrong results and/or functionality

Success

When to do this type of testing? Whenever there are significant external systems with considerable breaking changes and update cycles you cannot trust.

As we've seen, in a distributed technical landscape the piece that connects all of the services is the contract: The thing that expresses what a system expects and provides. Because of contract decoupling, we are free to independently evolve services and their APIs. How do you *test a contract*?

First, let's clear out what I mean by these different terms.

An **API specification**, in general, is an authoritative document that is both human-readable and machine-readable. We might for example use [OpenAPI](#) or [AsyncAPI](#) formats to document our API.

The actual API document (JSON or YAML file) that you write, we call your **API definition**. The definition acts as a **schema** and defines inputs and outputs, explaining how the API works and generally answering “what it can do”.

Information

I've borrowed this definition from [The Layers of the API Specifications, Definitions, and Schema Onion](#).

Now, when it's time to test against something, yes, you *can* test against the schema. But it won't cover all cases, and given the abstract nature of the definition/schema, it's more practical to think of it as a description rather than a technical artifact than is useful for testing.

Schemas are abstract, contracts are concrete.

— Matt Fellows, [Schemas are not contracts](#)

See his article for much more on this problem; in short, it's not quite a contract.

However, even just a collection of API requests/responses combined with the definition would be a starting point for a **contract**. A contract “[defines how two systems are able to communicate by agreeing on what interactions \(conversations\) can be sent between them and providing concrete examples to test the agreed behaviour](#)”. The bi-directional part, that both consumer (user) and provider (those who build the system), have an arrangement on the expectations towards each other is the semi-revolutionary feature here.

Examples

Let's say we have an API—let's call it Greeter—that returns Hello world! as its only output. We might *know* this fact, but how can we test this?

Our schema will be of typical OpenAPI 3 flavor:

```
openapi: 3.0.3
info:
  title: Greeter
  description: Greeter service
  version: 1.0.0
paths:
  /hello:
    post:
      summary: Greet person
      description: Greet person
      operationId: greet
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Greet'
components:
  schemas:
    Greet:
      type: string
      description: Returns 'Hello World {name}!'
```

In human-speak: Users can make a GET request to the path /hello and we will return Hello World! to them.

Using Ajv

For a really barebones approach, we could use [Ajv](#) for ensuring that crude expectations on a schema work. Below, we are passing in an object with an id key. This will work.

```
import Ajv from 'ajv';
```

```
const ajv = new Ajv();

// JSON Schema style schema we will test against
const schema = {
  type: 'object',
  description: 'Get user name by ID.',
  properties: {
    id: {
      type: 'string'
    }
  },
  required: ['id'],
  additionalProperties: false
};

// Our test input
const data = {
  name: 'Zelda'
};

// Test
const validate = ajv.compile(schema);
if (validate(data)) console.log('Success');
else console.log(validate.errors);
```

A solution for running this across teams could be to store schemas in some central location, pull these in CI, and run them against our test inputs (*expectations*). Obviously, here we only get a basic mechanism to compare payloads.

For the reasons mentioned above, and while this is a crude model (and yes, it works even across teams), it's not very sophisticated and we have to hand-crank pretty much everything.

Pact

The big dog in contract testing is [Pact](#), a piece of open-source software available for many languages and it also has a paid, managed version.

Pact also brings in the idea of a **broker**, a central location to store contracts, which starts to solve the problems around how we get to share expectations and contracts in the first

place.

Information

If not for anything else, then at least take some time to [read their good documentation](#), which also sells the good points about contract testing.

The following is from the documentation for [Pact JS](#). This uses Mocha for the test, and should be able to be replaced with Jest or something else if you want.

```
import { PactV3, MatchersV3 } from '@pact-foundation/pact';

// Create a 'pact' between the two applications in the integration we ↔
↔ are testing
const provider = new PactV3({
  dir: path.resolve(process.cwd(), 'pacts'),
  consumer: 'MyConsumer',
  provider: 'MyProvider',
});

// API Client that will fetch dogs from the Dog API
// This is the target of our Pact test
public getMeDogs = (from: string): AxiosPromise => {
  return axios.request({
    baseURL: this.url,
    params: { from },
    headers: { Accept: 'application/json' },
    method: 'GET',
    url: '/dogs',
  });
};

const dogExample = { dog: 1 };
const EXPECTED_BODY = MatchersV3.eachLike(dogExample);

describe('GET /dogs', () => {
  it('returns an HTTP 200 and a list of docs', () => {
    // Arrange: Setup our expected interactions
    //
    // We use Pact to mock out the backend API
    provider
```

```

    .given('I have a list of dogs')
    .uponReceiving('a request for all dogs with the builder pattern')
    .withRequest({
      method: 'GET',
      path: '/dogs',
      query: { from: 'today' },
      headers: { Accept: 'application/json' },
    })
    .willRespondWith({
      status: 200,
      headers: { 'Content-Type': 'application/json' },
      body: EXPECTED_BODY,
    });

    return provider.executeTest((mockserver) => {
      // Act: test our API client behaves correctly
      //
      // Note we configure the DogService API client dynamically to
      // point to the mock service Pact created for us, instead of
      // the real one
      dogService = new DogService(mockserver.url);
      const response = await dogService.getMeDogs('today')

      // Assert: check the result
      expect(response.data[0]).to.deep.eq(dogExample);
    });
  });
});

```

You’ll immediately note that we are back in regular “programmatic testing land” and that we have a mock concept in play. One of the reasons I’ve never grown fond of Pact is exactly because of the mock-heavy behavior—it’s always seemed oddly heavy-handed to me.

TripleCheck

The approach to contract testing I built a few years ago came from my frustrations with Pact; I call it [TripleCheck](#). Like Pact, it also provides a [broker](#) to store and retrieve our shared files.

This elaborates on the Ajv approach we looked at before.

Using the CLI to run tests, you can configure it to fetch all contracts and tests from the broker, get the latest distributed state, or decide to run completely off-the-grid from your local machine if you want.

The contracts are stored in an array where we can keep any number of them, and this is loaded into memory and tested when we run TripleCheck. Here we have the greeter↵↵ object with a 1.0.0 specifier and an object with the name key: Our actual contract. TripleCheck uses [QuickType](#) to infer this to be a required string input.

```
[
  {
    "greeter": {
      "1.0.0": {
        "name": "Somebody"
      }
    }
  }
]
```

The test that goes with the contract is pretty similar. Here we say we have a test called Greet person and the input is similar to that in the contract, so we already know this will work.

```
[
  {
    "greeter": {
      "1.0.0": [
        {
          "Greet person": {
            "name": "Zelda"
          }
        }
      ]
    }
  }
]
```

The drawbacks here are very similar to those with Ajv, and it's worth knowing that complex behaviors are logically not supported.

In closing

Contract testing is one of the more exciting ideas in testing right now, but it carries a complexity (infrastructural, cognitive, competence-wise) that is not worth taking unless you feel you've evolved to a position where your team can truly reap all the benefits.

Similar to when preparing for integration testing, there are auxiliary benefits you get by doing some of the boring preparatory work: Better documentation (and providing schemas to consumers), drawing better lines between “my system” and “other systems”, and pushing towards more unit tests. Do look at [Better APIs](#) for some other recommendations to drive API stability—despite contract testing being such a compelling notion, you shouldn't have to *only* resort to contract testing because other systems break for you. Evolve into this if needed, but start with the homework!

Continuous testing in CI

This is the last section before we move into some concrete testing scenarios. Here is where we will see how to push what you’ve learned about practical testing into the CI context and get ready to industrialize your work.

Google is great at development and I’m not going to be a stranger to a painfully long quote from [their own description](#) of [continuous testing](#):

The key to building quality into software is getting fast feedback on the impact of changes throughout the software delivery lifecycle. Traditionally, teams relied on manual testing and code inspection to verify systems’ correctness. These inspections and tests typically occurred in a separate phase after “dev complete.” This approach has the following drawbacks:

- Manual regression testing is time-consuming to execute and expensive to perform [...]
- Manual tests and inspections are not reliable [...]
- Once software is “dev complete”, developers have to wait a long time to get feedback on their changes [...]
- Long feedback cycles also make it harder for developers to learn how to build quality code, and under schedule pressure development teams can sometimes treat quality as “somebody else’s problem”.
- When developers aren’t responsible for testing their own code it’s hard for them to learn how to write testable code.
- For systems that evolve over time, keeping test documentation up to date requires considerable effort.

Instead, teams should:

- Perform all types of testing continuously throughout the software delivery lifecycle.
- Create and curate fast, reliable suites of automated tests which are run as part of your [continuous delivery pipelines](#).

Not only does this help teams build (and learn how to build) high quality software faster, DORA's research shows that it also drives improved software stability, reduced team burnout, and lower deployment pain.

— [Google](#)

It's always a bit iffy to do long quotes, but I definitely believe that the above encapsulates a lot of the reasons why you want to do proper Continuous Delivery and not the traditional, ill-performing delivery models.

Information

Consider also reading [Continuous Testing - Continuous Delivery](#), [Martin Fowler - Software Delivery Guide](#), or [Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation](#).

With newer conventions like [GitFlow](#) and [GitHub Flow](#), substantial deviations were made against the classical notion of Continuous Integration in which the trunk is always, continuously, integrated and ready to be deployed.

Information

See for example [Dave Farley's video providing some context on why GitFlow \(and GitHub Flow\) should be avoided](#).

I won't provide too much bias henceforth, but we will keep our example as lean as we can as I can't provide all answers for all integration models.

Scripting a CI pipeline to enable continuous testing


In the following example, we will use [GitHub Actions](#) to create a single pipeline that will take our code from commit to being deployed to production. Our theoretical application will be some sort of API.

The steps will be:

- Baseline activities
- Deploy a testing stack
- Test the stack
- Deploy the production stack

My tip is to always write in plain text—as above—what you want a pipeline to do *before* committing any work on it. Planning and thinking goes a long way!

GitHub Actions is a powerful and easy way to use CI/CD solution. You can set up your jobs pretty much just as you need them and you don't have to care about hardware as GitHub will provide all of that for you. What you need to do is to provide a description of your desired workflow—this is what we'll work on now.

You can have multiple workflows, meaning you can (for example) conveniently support a dedicated workflow for pull requests, something for main branch commits, and something for hotfixes. Again, here we will consider a single, monolithic pipeline called `main`  `.yaml`.

The script we will look at is nothing magical, which is a good thing. Remember that when you are running a workflow on a remote machine it is a “stateless” activity so things like dependencies you might have locally will typically not exist. It also won't persist state or carry over across jobs (unless you [upload/download](#) artifacts).

Enough said, time to check out the example.

```
on: [push]

jobs:
  #####
  # BUILD #
  #####

  build:
    runs-on: ubuntu-latest
    name: Build, package, and test
    steps:
```

```
- uses: actions/checkout@v4
- name: Install dependencies
  run: npm ci
- name: Lint
  run: |
    npx eslint './src/**/*.ts' --quiet --fix
    npx prettier ./src --check
- name: Compile
  run: npx tsc
- name: Package
  run: npx sls package
- name: Test
  run: npm test
```

```
gitleaks:
  needs: [build]
  runs-on: ubuntu-latest
  name: Gitleaks
  steps:
    - uses: actions/checkout@v4
      with:
        fetch-depth: 0
    - uses: gitleaks/gitleaks-action@v2
```

```
licenses:
  needs: [build]
  runs-on: ubuntu-latest
  name: License compliance
  steps:
    - uses: actions/checkout@v4
    - name: License compliance check
      uses: mikaelvesavuori/license-compliance-action@v1.0.3
```

```
trivy:
  needs: [build]
  runs-on: ubuntu-latest
  name: Trivy vulnerability scanner
  steps:
    - uses: actions/checkout@v4
    - uses: aquasecurity/trivy-action@master
      with:
        scan-type: 'fs'
        format: 'sarif'
        output: 'trivy-results.sarif'
        severity: 'CRITICAL'
```

```
checkov:
  needs: [build]
  name: Checkov
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - name: Checkov
      id: checkov
      uses: bridgecrewio/checkov-action@master
      with:
        directory: src/
        quiet: true

#####
# RELEASE #
#####

deploy-test:
  needs: [build, gitleaks, licenses, trivy, checkov]
  runs-on: ubuntu-latest
  name: Deploy to test
  steps:
    - uses: actions/checkout@v4
    - name: Configure AWS Credentials
      uses: aws-actions/configure-aws-credentials@v4
      with:
        role-to-assume: ${ secrets.AWS_ROLE_ARN_TEST }
        aws-region: eu-north-1
    - name: Deploy to test
      run: npm run deploy:test

live-tests:
  needs: [deploy-test]
  name: Run live tests
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - name: Run live tests
      run: |
        npm run deploy:test
        npm run test:smoke
        npm run test:integration
        npm run test:load
```

```
deploy-prod:
  needs: [live-tests]
  runs-on: ubuntu-latest
  name: Deploy to production
  steps:
    - uses: actions/checkout@v4
    - name: Configure AWS Credentials
      uses: aws-actions/configure-aws-credentials@v4
      with:
        role-to-assume: ${{ secrets.AWS_ROLE_ARN_PROD }}
        aws-region: eu-north-1
    - name: Deploy to production
      run: npm run deploy
```

The most crucial concepts to understand now are:

- **on:** What can trigger this workflow? In this case, it's a “push” to any branch.
- **jobs:** Sets of actions that can be run. The jobs align well with our previous list of desired activities.
- **steps:** Defines the individual things happening, such as running a CLI command. Most jobs use ready-made Actions (see below) but a few use regular Bash/shell commands mapping to scripts defined in our assumed `package.json` file of this project.
- **uses:** Allows the use and reuse of, for example, Marketplace-provided Actions. This constitutes the majority of activities.
- **secrets:** Makes it possible to use concealed values that we want to use but not be able to read back in clear text. Great for stuff like passwords, credentials, or other sensitive values.

Information

To learn more about the specifics of GitHub Actions, turn to [Understanding GitHub Actions](#). If you are interested in more about how the AWS credentials section works, [read this practical article by my colleague Elias Brange](#).

Our script as per above would cover most of the essentials when it comes to quality-checking an API. Provided that our testing practices were half-decent we can now feel entirely confident in deploying directly into production without any further manual intervention.

Need for speed: Pushing CI times

When we talk about CI/CD there is a widely known heuristic that we want testing and deployment to finish in no more than X number of minutes. You'll easily find numbers ranging from maybe 5 to 15 minutes. The exact number is not important—what matters is that you do what you can to ensure that you get the fastest possible feedback from testing your system.

Technical activities you can do include:

- **Add more/faster hardware.** However, this rarely increases performance in a meaningful way!
- **Migrate to faster deployment mechanisms.** If you are in the AWS world, you will know that CloudFormation can be a very slow experience. Migrating to other models, such as Cloudflare Workers might be a reasonable option for certain use cases; Changing could take you from minutes of deployment time to a few seconds. Of course, this is an extreme option, but keep it in mind when you are assessing architectures from a holistic, lifecycle perspective.
- **Don't wait for teardowns.** If you are using tools like Terraform, you can opt to not wait for the completion of removal/destruction tasks. This will save you quite some time if you are currently awaiting the results of such tasks in your pipeline.
- **Do less.** You can always choose to do less. Sometimes that makes sense. You can also look at a more elaborate deployment strategy, though I personally fear that could drive you from aiming for a simple and clean strategy which is a *better* goal.
- **Deploy to fewer stages.** Related to the “slow deployment mechanisms” point, it makes sense to deploy to the minimum number of stages needed. Naturally, if it takes 2 minutes to deploy to a given stage, it's a matter of $\text{<stage count>} * 120$ seconds—yet another good argument for a [single environment setup](#).

- **Remove long-running and/or slow tests.** This is the thing *you absolutely should do*, and a great driver for deleting and/or migrating tests from the system/UI level to unit tests. You should expect at least 1000 unit tests per minute to run. You'd be happy to have 10 end-to-end tests run in that space.

Breaking down a big script into smaller ones

When a pipeline becomes unwieldy or you want to support many teams composing their own pipelines from smaller pieces you can turn to [reusable pipelines](#).

Using these reusable pipelines we can start disassembling our script and provide, as here, the overall build step as a single component.

```
name: 'Build, package, and test an application'

on:
  workflow_call:

jobs:
  build:
    runs-on: ubuntu-latest
    name: Build, package, and test
    steps:
      - uses: actions/checkout@v4
      - name: Install dependencies
        run: npm ci
      - name: Lint
        run: |
          npx eslint './src/**/*.ts' --quiet --fix
          npx prettier ./src --check
      - name: Compile
        run: npx tsc
      - name: Package
        run: npx sls package
      - name: Test
        run: npm test
```

You'll need to make it available somehow. Let's assume you put these pipelines in a repository with such bits and bobs. Now, to use the above workflow from your calling (orchestrating) workflow, you can simply do:

```
name: 'Call reusable workflow'

on: push

jobs:
  build:
    uses: YOUR_USER/REPO_WITH_REUSABLE_WORKFLOWS/.github/workflows/↔
    ↪ build.resusable.yml@main
```

It's a nifty way to scale your CI/CD operations and to decouple your own work for that matter if you have a whole lot of repositories.

To summarize, without fully committing to a competent CI pipeline you won't be able to take advantage of all the benefits that modern software engineering and testing can provide you. I've heard horror stories of teams that attempt to set arbitrary limits of 80/20 test automation and use CI tooling that should have been considered useless even 10 years ago.

If you take anything from this book, then that should be how critical continuous integration (and testing) is for doing software engineering well.

Example testing scenarios

In this short final section, I will present three brief scenarios that hopefully won't be too far removed from your experiences and how we could approach testing for them.

Front-end application

Information

Scenario Our team is building a new web application (front-end) for a small client. We've been granted access to the previous iteration written in another language; it's good enough for us to understand generally what we need to do for the back-end integration. **Testable surface** Web front-end. **Code access** First-hand access to front-end code and any new integrations; no access to existing back-ends or APIs.

Success

Proposed approach Write key user flows as E2E tests. Separate front-end logic into unit-testable functions and write snapshot tests for UI components. Send integration errors to an APM service with alerts to our business stakeholders and our technical team.

This scenario will push nerves a bit because it will feel antithetical to much of what I've written in this book. Fear not, we will make the best of it!

In a front-end team, we tend to control a lot less of our circumstances. Still, as you've read before, we need to be crystal clear in our boundaries: I don't want to test *your* things! The same goes for you: There should be no need for you to test the packages, modules, or APIs *I've* provided you with.

Alas, the reality of our front-end application is that it is by nature a “dirtier” thing than a pristine back-end. Not because I like it less—not at all!—but simply because it has to cobble together and use an array of services and whatnot to work at all. In the back-end world, we can talk, something dismissively, about “integration”, yet no one talks in such illusive terms about setting up the Stripe API in a web application.

I believe we can sense a basic maxim at play:

You wouldn't test third party code (so don't).

Or in similar terms... “[You wouldn’t steal a car?](#)”?

Information

Yes, I am old enough to remember this clip being rolled in the pre-movie trailer section of VHS tapes. Remember those?

So with that long digression out of the way: Why would you test someone else’s code, even if it’s that written by the friendly team down the hall? Don’t.

Information

This example uses the [microservices testing workshop](#) codebase.

Start with cleaning up, then introduce unit tests and static code analysis

A great place to start is to ensure that the code is in ship shape.

If you are using a **pure vanilla stack** (and I do literally mean HTML, CSS, JS) then you will have a *very clean* baseline to work with. Of course, with a higher degree of flexibility comes a requirement for more mastery of the principles. You could consider for example rigorous [BEM naming conventions in CSS](#), [super-tidy JavaScript](#) (maybe even a functional programming style), a hexagonal/[Clean architecture](#) style (folders, etc)... Fewer guardrails give ultimate freedom; intoxicating but dangerous in the wrong hands! While it may sound archaic to some to revert back to these core basics, it’s actually not too dissimilar from other “rawer” approaches like [Svelte](#) or [Alpine.js](#). There’s definitely some kind of resurgence in this general area.

With this approach, I would argue it’s clearer how you write clean code and any requisite tests. Just write them! Always make sure your code and tests on the local level are watertight and of good quality.

In a framework-driven and more conventional setting, such as if you are using a freer and

less opinionated one like React, then you can stay vigilant on not polluting functions with business logic. Instead, separate complicated or complex functions into smaller, testable bits. Consider the easiest and rawest option as generally the best option: Functions over hooks, for example. There's tons of good reading and advice on this (and some bad), but I can recommend following [Nik Sumeiko on LinkedIn](#) as a start if you're a React developer.

Introduce static code analysis tools as well, such as ESLint and Prettier, while you are at it.

Visual snapshot tests might be relevant as well, depending on the volatility and complexity of our project.

Checking the full flow (or as much of it as possible)

Given that we control none of the backends our most reasonable hope, based on the likely composition of our team, is to add end-to-end tests. Integration tests and contract tests won't be of any real use to us here, unfortunately.

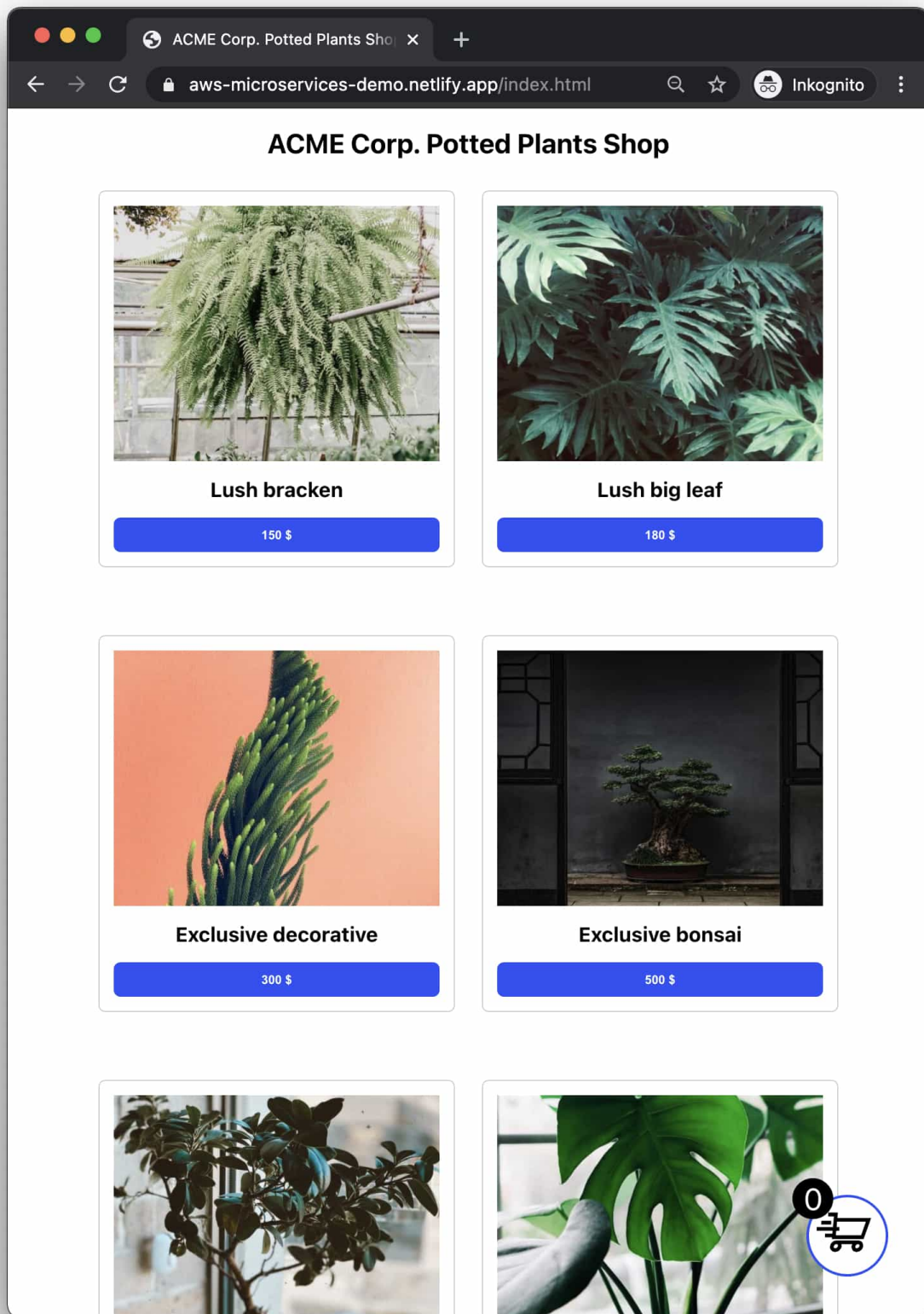


Figure 1.31: Our end-to-end flow could check that external features work as intended. Pure logic should already be covered in unit tests, as always.

Before doing so, we need to check if we have testing endpoints or are allowed to create test transactions so that we know whether or not it's OK that our flow actually goes all the way, or if we need to stop short of that.

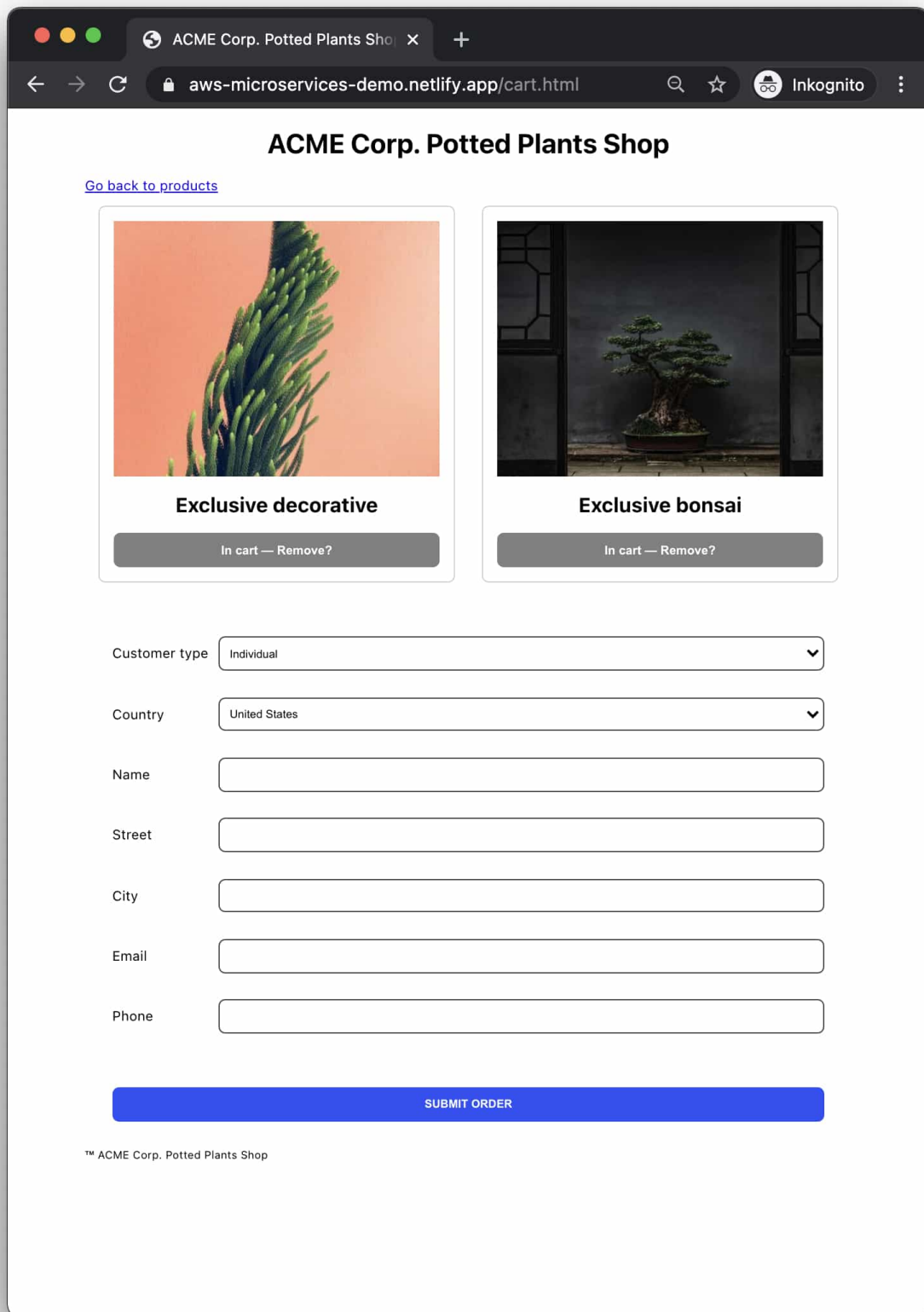


Figure 1.32: Is it OK to submit our test order? Before continuing we need to know what to do about side effects.

Errors need to go to some tool

It's advisable that all errors are passed to some tool, for example, an Application Performance Monitoring (APM) tool such as [Sentry](#). The less control we have, the more we should treasure any feedback we get—especially the negative ones, such as actual errors! This will help us know if integration errors occur for our customers since we (yet again) don't control any of the other systems ourselves.

Missing out on implementing this capability in our current circumstances will make us blind, so make sure to add one.

We could also combine some basic synthetic testing with our APM tool to feed data to us before we have a handle on the real user traffic.

Back-end relying on third parties

Information

Scenario You are brought in to do a minor clean-up for the order-taking part of an e-commerce solution. It's requested that you help train the original development team how to richly test their solution as well since their manual end-to-end process is impeding the delivery of new (well-tested!) features. **Testable surface** Two back-end solutions use one or more Lambda functions; one solution is publicly exposed, and the other is internal. Also, a message bus and a database. **Code access** First-hand access to all code and infrastructure. No access to third-party code but full access to their API documentation.

Success

Proposed approach Collaborate on the API, document it, and set up request validation. Write unit tests on the functional (use case) level for all of our functions. Write basic integration tests to all services backed by APIs; but only if that they produce side-effects that are easily handled or use QA/mock/fake endpoints.

In this scenario we come in as a senior, advising a team and their existing solution. While we are new to the context, we also are in a bit of a luxury position in that we can focus entirely on improving the quality and testing parts of the solution.

Information

This example uses the [microservices testing workshop](#) codebase.

The separation in the solution is already good and very precise. If this weren't the case, we might have wanted to first start cutting out the respective parts better. This time we didn't need to.

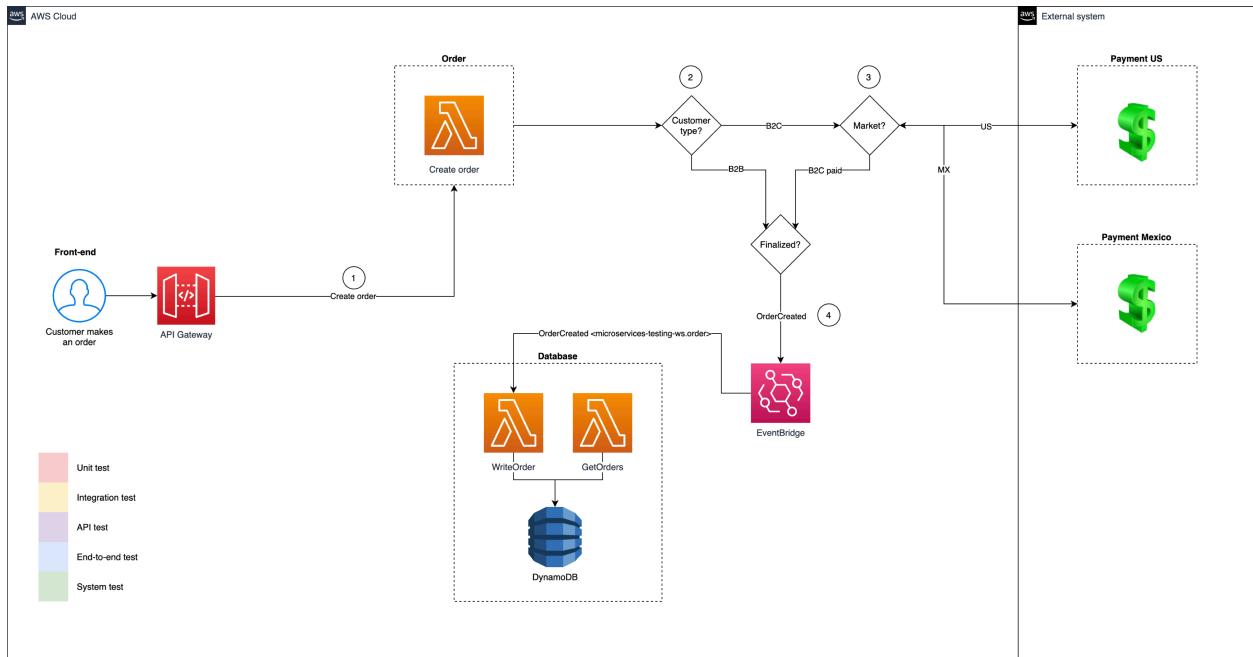


Figure 1.33: Very simple, but clean microservices architecture.

Improving the API

There's also a chance to talk to the team building the front end (given it's not us!) and define an API collaboratively. By this point, we *already have an API* but it might be worth checking in if we are both satisfied with it. We should spend some time creating an actual API schema and making it available, to fend off any avoidable misuse from our consumers. Also, let's slap on a validator on the API for good measure.

Next, we can move on to creating a Data Transfer Object wrangler function so that every time we actually operate on “business logic” we do it from a well-defined object and not just some indiscriminate blob passed into the API Gateway.

At this stage, there is already a massively reduced error surface.

Writing the unit tests

Now we can inspect the current state of unit testing. We should focus, given that the architecture itself is well-separated, to write use case-oriented unit tests. With the DTO wrangler written, it should be pretty easy to write these tests in a way we feel confident

about.

The majority of testing seems to concern how the order is correctly produced based on their market and customer type. These tests should be flawless. If the payment providers provide provider-side mocks, test APIs, or something else we can use, then great. If not, let's just mock them out based on what we know from their docs.

For the Database service, the testing should potentially be even simpler as it should now only be able to receive well-defined inputs. Also, they don't interact with nearly as many systems.

We will also create some testable functionality to create correctly-shaped objects for our persistence, so we can mock out DynamoDB while being safe in knowing the objects are always stored correctly. On top of that, we will add the semi-official types and maybe the AWS SDK mock too.

Integration tests

If it's possible to run integration tests against the third parties, we can do this now after the more pressing parts are handled. If not, then we'll simply stick with the mocks we have and make sure we have good monitoring and/or APM software in place to collect integration errors.

Serverless, distributed, event-driven application

Information

Scenario You are architecting a new serverless, microservices-oriented solution from scratch. It needs to support good development and testing practices.

Testable surface Multiple independent services consisting of any number of serverless functions and other cloud infrastructure components. **Code access** First-hand access to all code.

Success

Proposed approach Write rich, deep unit tests for every service and Lambda function. Produce and distribute all contracts (including event shapes) for services in a location all teams have access to. Produce and distribute diagrams (flows and sequences) of events in the same location. Create a repository of shared mock event data to facilitate testing of the independent services, or even consider full-on contract testing. Define a way to separate test data from production data either with separate infrastructure or by tagging test data as such. Consider a very lightweight system test that can generate a test order and can verify the existence of the final, settled data in the database; do not over-invest in this capability but think of it as a system-wide smoke test.

The present scenario is much more dynamic and complex than the other scenarios we've looked at. Also, this time we are entirely first-party and can afford to do some things we might not be able to do when there is a third party involved, such as running system tests. But in that detail is actually a very testable solution!

Information

This example uses the [ACME Corp. Potted Plants Web Shop: AWS microservices demo](#) codebase.

It's a well-considered architecture that uses messages on EventBridge to communicate completely asynchronously. The only synchronous actions that happen are from the customer side when they make an order or book a delivery time.

Define the API

Given the criticality of the API, driving all of those flows with only two commands, it's the first thing we should look at. What should the respective endpoints require in terms of input data? Too much and detail-oriented and we risk making a CRUDdy API; too little it might be too chatty or not able to drive all the flows. Let's talk to the team implementing the front end of this if it's another team.

Same as in the last chapter, we want documentation, request validation, and a good-looking API that people actually want to use.

This is less about testing than putting things in order.

Define events

The absolute majority of communication happens through events. We'll want to document and collaborate on the events. Ideally, every service is documented. AsyncAPI can do this for this type of event-driven service; in that case, we can describe our events there.

From the events, it would be easy to infer test data, if the owning team does not produce it themselves.

Share the events and test data in a way that is ideally automatable.

Version your events and don't break functionality unless you emit a later event version.

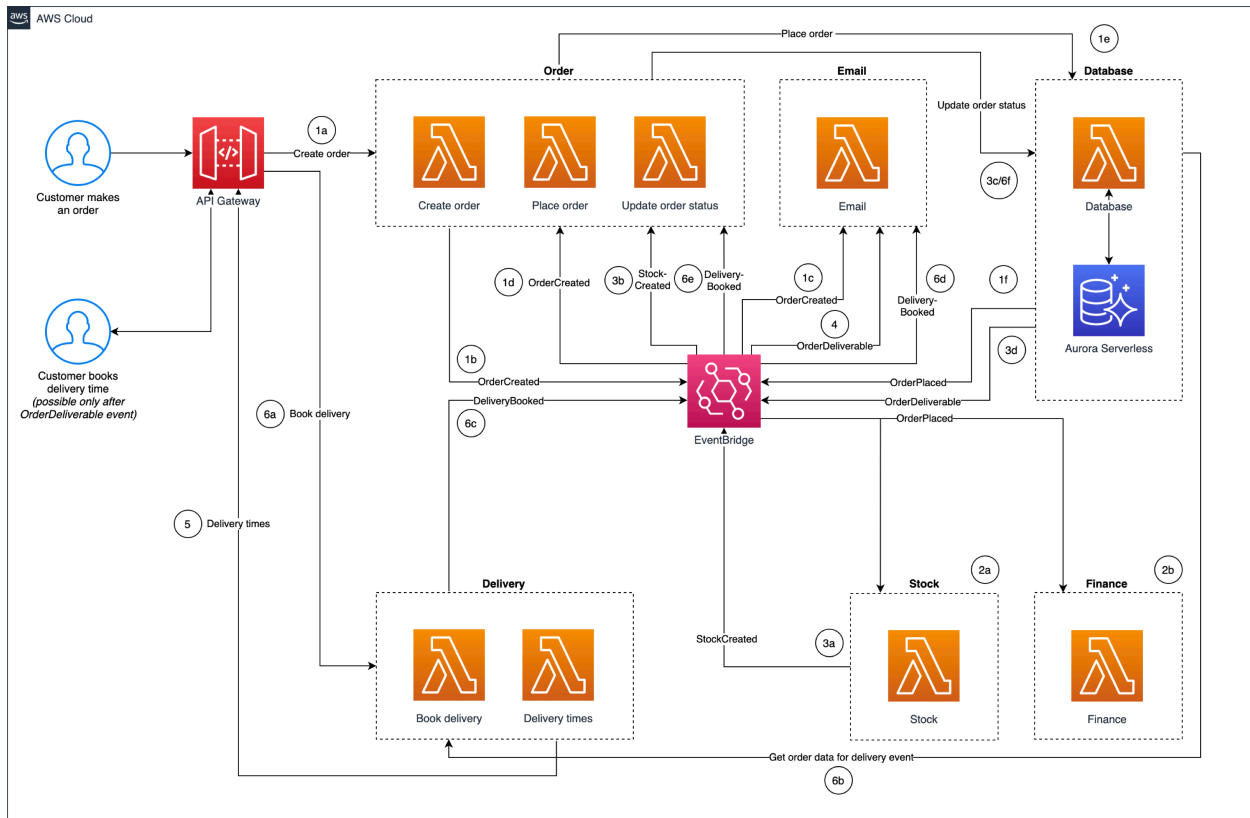


Figure 1.34: Six different services connected with lots of events, a public API Gateway, persistence... Oh my!

Unit test the interfaces

Once there is some kind of idea of what events will drive the respective services, we can create (or use the shared) test data and use it as input in our use case-oriented unit tests.

Because the services are small, doing only one or a few things, and because there is only persistence in a single service, then we should quite easily be able to write out tests without any mocking and other such activity other than in the Database service. We can see from the diagrams which events we need to act on, so there should be no surprises here.

Unit tests should, as I nag about a lot, form the vast majority of your strategy here. Especially as you have well-known inputs and interactions.

I don't actually think integration is that important in this scenario unless we start having real issues creating conditions for us to question that position. As the events are clearly

outlined and presented, and we have a pool of test data, then I find it strange if we would start seeing *change- or evolution-based problems* in integrating our services.

Ensuring fit with a system test

It's perhaps worth considering a system test that creates a small number of example orders and ensures they end up in the database. This would have been a risky proposition if we had third-party systems, but here we are completely self-contained.

Due to the high number of connecting parts, a system test may actually not be that dumb of an idea, and we could use it as a kind of elaborate smoke test during our development as well. Don't make the test too unwieldy or force it to assert every interaction—just check that the order was stored, because it is, then well, everything worked out, didn't it?

While not *strictly necessary* it could fill the role of a development tool (seeing it's all moving towards a working state) and a way to know that everything works in the future, too.

Information

Just don't forget to handle all of the test data you create!

In summary

The way I think of testing could be summarized as this:

- **Remember the five rules of testing:**
- [Write good, deterministic code.](#)
- **Good software architecture will make your code easier to test**, by being cleaner, better formatted, and layered in a way where you know what is important and what is less important.
- The key thing for testable code is already stated in the SOLID principles, more exactly the S—the **Single Responsibility Principle**. If a good test asserts one thing, and your *system under test* has a big bag of tricks and all kinds of wizardry, then how will you be able to write any good tests?
- **Make sure your code is observable, i.e. it's possible to know what went wrong with it.**
- **Nothing is untestable, but do remember that poor code is very hard to test.** Refactoring code to be “good” and testable is commendable and a real part of your job as a developer. Just do it, don't ask for permission.
- **Accept that tests are only a part of your software quality strategy.**
- **Use “testing in production”** as a way to observe and learn from failures in your actual production system.
- **Always write unit tests.** Never skip them.
- **The correct level of unit test coverage is 100% (overall).** See this as a definition of done. Deviations may be accepted when it comes to testing side effects (which we can't or won't).
- **Full coverage does not preclude that other types of issues may not occur.** However, in keeping with building qualitative, deterministic systems, then full coverage should be a very good indicator of overall stability.
- **Tests should be handled as code**, with the same semantics, standards, care, and maintenance, and they should be colocated with the code that it checks. Never divorce tests into their own repositories or contexts.

- **Catch as much as you can already in the IDE.** Enable testing locally, as a pre-commit hook, and in CI to ensure testing and quality are adhered to at every stage. Also, add stricter compilation options and IDE tooling to help you normalize the code to expectations so there is no environmental drift.
- **Push as much testing burden “to the left” as possible**, i.e. prioritize fast, lightweight tests (unit tests and sprinkles of strategic integration tests) over heavier, slower, frail tests like end-to-end tests.
- If you feel like TDD is too much for you, consider a variant in which you **simply write and run a test as the actual input for your development work**. While this is a marginally different approach, it is not quite as dogmatic. And you will have your test(s) written! Not even saying that you will have a much quicker feedback loop while developing.
- Testing models (pyramids etc.) are useful as a theoretical construct but they are just that: Theory. **Adapt what tests you use based on your actual use cases.** Remember that any useful models *will* discourage the use of system tests and end-to-end tests if at all possible. Prioritize easier, faster, stable test types over others to an extreme degree.
- **Optimize for a good build chain** with high-quality code analysis tools and stable testing tools.
- **If building APIs, always write an API schema.** If you have to go for a leaner approach, for example during early development, then write basic endpoints and example requests/responses in your README .md.
- **If you can, then always use request validation at the API level to offload some testing.** While this may sound like a security feature (as it ensures incoming input follows the right convention) or a quality add-on (making the API easier to consume) it also ensures that the surface area of your testing grows much smaller. You can focus on what is actually meant to happen.

References and resources

Sharing is caring, and this is what I could remember.

Books

Online books

- [Better APIs](#)

Physical books

- [Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests \(2009\)](#)
- [Kent Beck: Test-Driven Development: By Example \(2002\)](#)
- [Lisa Crispin, Janet Gregory: Agile Testing: A Practical Guide for Testers and Agile Teams \(2009\)](#)
- [Janet Gregory, Lisa Crispin: More Agile Testing: Learning Journeys for the Whole Team \(2014\)](#)
- [Kent Beck, Cynthia Andres: Extreme Programming Explained: Embrace Change \(1999\)](#)
- [Matt Weisfeld: The Object-Oriented Thought Process \(2000\)](#)
- [Martin Fowler, Kent Beck, Don Roberts: Refactoring: Improving the Design of Existing Code \(1999\)](#)
- [Forsgren, Humble, and Kim: Accelerate: Building and Scaling High Performing Technology Organizations \(2018\)](#)
- [Jez Humble and David Farley: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation \(2010\)](#)
- [Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm: Design Patterns: Elements of Reusable Object-Oriented Software \(1994\)](#)

- [Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship \(2007\)](#)
- [Robert C. Martin: The Clean Coder: A Code of Conduct for Professional Programmers \(2011\)](#)
- [Robert C. Martin: Clean Craftsmanship: Disciplines, Standards, and Ethics \(2021\)](#)
- [Robert C. Martin: Clean Architecture: A Craftsman's Guide to Software Structure and Design \(2017\)](#)
- [Robert C. Martin: Clean Agile: Back to Basics \(2019\)](#)

Code

- [Building a Production Ready API \(demonstration\)](#)
- [Testable Systems Starter](#)
- [Better APIs workshop: Enhancing an API for quality, stability, and observability](#)
- [Microservices testing workshop](#)
- [ACME Corp. Potted Plants Web Shop: AWS microservices demo](#)
- [Domain Driven Microservices on AWS in Practice](#)

Tools

As the book itself, my focus is mostly from the eyes of a Node/TypeScript developer with some familiarity of the .NET Core universe.

Unit testing

- [Jest](#)
- [AVA](#)
- [XUnit](#)

End-to-end testing (UI)

- [Playwright](#)
- [Cypress](#)
- [CodeceptJS](#)
- [Puppeteer](#)

Cross browser testing

- [Lambdatest](#)
- [BrowserStack](#)

BDD testing

- [Cucumber](#)

Front end specific tooling

- [React Testing Library](#)
- [webhint](#)

Static code analysis (etc.)

- [Prettier](#)
- [ESLint](#)
- [Roslyn](#)

Online (primary)

- [DeepSource](#)
- [Codacy](#)
- [Codiga](#)
- [Embold](#)
- [SonarCloud](#)

Security focus

- [Snyk](#)
- [Checkov](#)
- [trivy](#)

Integration testing

- [Insomnia](#) for manual work and [Inso](#) for CI/CLI
- [Postman](#) for manual work and [Newman](#) for CI/CLI

API mocking

- [MSW](#)
- [Mockachino](#)

AWS mocking

- [aws-sdk-mock](#)

Contract testing

- [Pact](#)
- [TripleCheck](#)

Synthetic testing

- [CloudWatch Canaries](#)
- [Checkly](#)

Load or performance testing

- [k6](#)
- [Artillery](#)
- [Measuring response times with Bash](#)

Chaos testing

- [AWS Fault Injection Simulator](#)
- [Gremlin](#)
- [failure-lambda](#)

Websites

- [Google DevOps](#)
- [DORA research program](#)
- [Google Testing Blog](#)
- [Understand Legacy Code](#)
- [JavaScript Testing Best Practices](#)

Articles

Testing

- [Microsoft Azure DevOps: Shift testing left with unit tests](#)
- [Testing and debugging serverless applications](#)
- [Serverless Framework: Testing](#)
- [Testing serverless apps has never been easier!](#)
- [Serverless Testing: Adapt or Cry](#)
- [Testing in Production, the safe way](#)
- [Testing Microservices, the sane way](#)
- [Testing in Production: the hard parts](#)
- [Mocks Aren't Stubs](#)
- [Mockists Are Dead. Long Live Classicists.](#)
- [UnitTest](#)
- [TTDD - Tautological Test Driven Development \(Anti Pattern\)](#)
- [How deep are your unit tests?](#)
- [Simplifying Tests by Extracting Side-Effects](#)
- [A quick way to add tests when code has database or HTTP calls](#)
- [Test Flakiness - One of the main challenges of automated testing](#)
- [Stop mocking fetch](#)
- [Test Doubles — Fakes, Mocks and Stubs](#)
- [The serverless approach to testing is different and may actually be easier](#)
- [A Comprehensive Guide to Contract Testing APIs in a Service Oriented Architecture](#)
- [Serverless Testing in Production](#)

- [My unit testing epiphany](#)
- [Production-Readiness Checklist, by Susan Fowler](#)
- [Evaluate Your Microservice, by Susan Fowler](#)
- [Testing Event-Driven systems](#)
- [Contracts, Addressing, and APIs for Microservices](#)
- [Consumer-driven Contract Testing using Postman](#)
- [Using curl for API testing](#)
- [The only 3 steps you need to mock an API call in Jest](#)

Trunk-based development

- [DevOps tech: Trunk-based development](#)
- [Trunk-Based Development](#)
- [Patterns for Managing Source Code Branches](#)
- [Why I love Trunk Based Development \(or pushing straight to master\)](#)
- [Straight to Prod](#)

Deployment

- [The Why and How You Should Test in Production](#)
- [Testing in Production: rethinking the conventional deployment pipeline](#)
- [Why you should use temporary stacks when you do serverless](#)
- [Fulfilling the promise of CI/CD](#)
- [Deployment Strategies Defined](#)
- [There can be only one \(environment\): Production](#)

Feature flags

- [Four Shades of Progressive Delivery](#)
- [Coding with Feature Flags: How-to Guide and Best Practices](#)
- [Progressive Delivery](#)
- [Feature Toggles \(aka Feature Flags\)](#)
- [Feature Flag Testing—Strategies and Example Tests](#)
- [Testing in Production to Stay Safe and Sensible](#)
- [What Is Progressive Delivery All About?](#)
- [Testing in Production to Stay Safe and Sensible](#)

Thank you for reading this book

Thank you for investing your time, energy, and money into reading this book. With every book and article I write, I strive to make it as useful as possible. Books allow us to delve deeper—or sometimes broader—into topics than we typically can at work or in short-form articles. Technical books, in particular, are unique creatures: they are both products of their time and, when well-crafted, can become timeless resources within their field. I hope this book remains relevant for (at least a few!) years to come.

I write the books I wish I had read earlier in my career and life. I've tried to be generous with references to other content, such as books and articles that have helped me improve in this subject. There are so many great authors out there and so much knowledge to keep up with.

If you found this book helpful, I would greatly appreciate it if you could rate it on the platform where you purchased it.

Please don't be a stranger! Connect with me on LinkedIn or wherever else I may be when you're reading this.

Once again, thank you, and I hope you found value in the time we spent together.