Transitioning from Monolith to Microservices Handbook

Transitioning from Monolith to Microservices Handbook

Converting monoliths to the microservice architecture

Semaphore

Contents

Preface . . . 7
Who Is This Book for, and What Does It Cover? . . . 7
Additional recommended reading . . . 7
How to Contact Us . . . 8
About the Authors . . . 8
About the Editor . . . 8
About the Reviewer . . . 8

Chapter 1 — What Are Microservices? 9
1.1 What is microservice architecture? . . . 9
1.2 Microservice vs. monolith architectures . . . 9
1.3 Benefits of microservices . . . 10
1.3.1 Scalability . . . 10
1.3.2 Fault isolation . . . 10
1.3.3 Smaller teams . . . 10
1.3.4 The freedom to choose the tech stack . . . 10
1.3.5 More frequent releases . . . 10
1.4 The caveats of microservice design . . . 10
1.5 Microservice design challenges . . . 11
1.6 Reasons not to migrate to microservices . . . 12
1.6.1 Microservices are only viable for mature products . . . 12
1.6.2 Microservices are not a good fit for startups . . . 12
1.6.3 Microservices aren't the best for On-Premise installations . . . 13
1.6.4 If it's working, don't fix it . . . 13
1.6.5 Brooke's Law and developer productivity . . . 14
1.6.6 You may not be prepared for the transition . . . 15
1.7 Is it the right time for the switch? . . . 16
1.8 Revitalizing monoliths . . . 16
1.9 The modular monolith as an alternative to microservices . . . 16
1.10 Scalable monoliths . . . 18

Chapter 2 — How to Restructure Your Organization for Microservices 20
2.1 Hierarchical organizations . . . 20
2.2 The pod model . . . 22
2.3 Ownership is the key . . . 23
2.4 Cross-service finger-pointing . . . 23
2.5 Pods need support . . . 25
2.6 The STOSA model . . . 25

Chapter 3 — Design Principles for Microservices 27
3.1 What is Domain-Driven Design? . . . 28
3.1.1 Bounded Context (BC) . . . 29
3.1.2 Context Map . . . 29

3.2 Domain-Driven Design for microservices 30
3.3 Strategic phase 30
3.3.1 Types of relationships 32
3.4 Tactical phase 33
3.5 Domain-Driven Design is iterative 34
3.6 Complementary design patterns 34

Chapter 4 — From Monolith to Microservices 36
4.1 The single point of fragility 36
4.2 Slow development cycles 37
4.3 Preparing your monolith for transitioning to microservices 38
4.4 A migration plan 38
4.4.1 Put everything in a monorepo 38
4.4.2 Use a shared CI pipeline 39
4.4.3 Ensure you have enough testing 39
4.4.4 Install an API Gateway or HTTP Reverse Proxy 40
4.4.5 Consider the monolith-in-a-box pattern 41
4.4.6 Warm up to changes 42
4.4.7 Use feature flags 42
4.4.8 Modularize the monolith 43
4.4.9 The strangler fig pattern 44
4.4.10 The anticorruption layer pattern 45
4.4.11 Decouple the data 45
4.4.12 Add observability 47
4.5 Techniques for testing microservices 47
4.6 The testing pyramid for microservices 48
4.6.1 Unit tests for microservices 48
4.6.2 Contract testing 49
4.6.3 Integration tests 50
4.6.4 Component tests for microservices 51
4.6.5 In-process component testing 52
4.6.6 Out-of-process component testing 54
4.6.7 End-to-end testing in microservices 54
4.7 Changing the testing paradigm 55

Chapter 5 — Running Microservices 56
5.1 Deploying microservices 56
5.2 Ways to deploy microservices 56
5.2.1 Single machine, multiple processes 57
5.2.2 Multiple machines and processes 59
5.2.3 Deploy microservices with containers 60
5.2.4 Containers on servers 62
5.2.5 Serverless containers 62
5.2.6 Orchestrators 63
5.2.7 Deploy microservices as serverless functions 65

5.3 Which method is best to deploy microservices? 67
5.4 Release management for microservices 67
5.4.1 A common approach: one microservice, one repository 67
5.5 Maintaining multiple microservices releases 68
5.6 Managing microservices releases with monorepos 69
5.7 Never too far away from safety 71
5.8 When in doubt, try monorepos 72

Parting Words **73**
6.1 Share This Book With The World 73
6.2 Tell Us What You Think 73
6.3 About Semaphore 73

© 2022 Rendered Text. All rights reserved.

This work is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/4.0

This book is open source: https://github.com/semaphoreci/book-microservices

Published on the Semaphore website: https://semaphoreci.com/resources/microservices

Sep 2022: First edition v1.0 (revision e932d9d)

Share this book:

> *I've just started reading "Transitioning from Monolith to Microservices Handbook" a free ebook by @semaphoreci: https://bit.ly/3eWMTA0 (Tweet this!)*

Preface

Microservices are the most scalable way of developing software. As projects grow in size and complexity, one of the possible ways forward is to break the system into autonomous microservices and hand them out to different teams.

Given the advantages, one would be forgiven for thinking that microservices are the superior architecture. But there are some caveats that, if ignored, can lead to development hell. This book aims to help you decide when migrating your monolith to the microservice architecture is a good idea, if so, navigate the choppy waters ahead.

Who Is This Book for, and What Does It Cover?

This book is intended for software engineers at every level and tech leaders who are either exploring microservice architecture or are faced with serious scalability problems in their monolith applications.

- In chapter 1 we define microservices and weight this architecture model against the alternatives. If you're unsure if microservices is right for your project, be sure to not skip this chapter.
- Chapter 2 talks about the cultural transformation a company must undergo to be effective at microservice design and operations.
- Chapter 3 covers design techniques for microservice application. We take a deep dive into Domain-Driven Design and how it applies to microservices.
- Chapter 4 goes to the core of breaking up the monolith. We lay a plan for the migration, discuss the steps required to prepare the monolith before the transition, and explore techniques for testing microservice applications.
- Chapter 5 covers the operational side of running a microservice application, including deployment and release management.

Additional recommended reading

You won't learn absolutely everything you need to design and run microservices in this book. Instead, the focus is to break up a monolith since this is the most common (and even recommended) path to microservices. As supplementary material, we recommend the following free ebooks also published by Semaphore:

- CI/CD with Docker and Kubernetes: it's common practice to run microservices with containers and orchestrate them with Kubernetes. This book will introduce both concepts and show step-by-step how to work with them.
- CI/CD for Monorepos: monorepos are a popular way of organizing and developing microservice codebases. This book will show you the best ways of working with monorepos.

How to Contact Us

We would very much love to hear your feedback after reading this book. What did you like and learn? What could be improved? Is there something we could explain further?

A benefit of publishing electronically is that we can continuously improve it. And that's exactly what we intend to do based on your feedback.

You can send us feedback by sending an email to learn@semaphoreci.com.

Find us on Twitter: https://twitter.com/semaphoreci

Find us on Facebook: https://facebook.com/SemaphoreCI

Find us on LinkedIn: https://www.linkedin.com/company/rendered-text

About the Authors

Pablo Tomas Fernandez Zavalia is an electronic engineer and writer. He started out developing for the City Hall of Buenos Aires (buenosaires.gob.ar). After graduating, he joined British Telecom as head of the Web Services department in Argentina. He then worked for IBM as a database administrator, where he also did tutoring, DevOps, and cloud migrations. In his free time, he enjoys writing, sailing, and board games. Follow Tomas on Twitter at @tomfernblog.

Lee Atchison is a software architect, published author, and frequent public speaker on the topics of cloud computing and application modernization. Follow Lee at @leeatchison.

About the Editor

Marko Anastasov is a software engineer, author, and entrepreneur. Marko co-founded Rendered Text, the software company behind the Semaphore CI/CD service. He worked on building and scaling Semaphore from an idea to a cloud-based platform used by some of the world's best engineering teams. Follow Marko on Twitter at @markoa.

About the Reviewer

Dan Ackerson picked up most of his soft and hardware troubleshooting skills in the US Army. A decade of Java development drove home to operations, scaling infrastructure to cope with the thundering herd. Engineering coach and CTO of Teleclinic.

Chapter 1 — What Are Microservices?

Beloved by tech giants like Netflix and Amazon, microservices have become the darlings in modern software development. But, despite the benefits, this is a paradigm that is easy to get wrong. So, let's explore what microservices are and, more importantly, what they are not.

1.1 What is microservice architecture?

The microservice architecture is a software design approach that decomposes an application into small independent services. These services communicate over well-defined APIs, which means that services can be developed and maintained by autonomous teams, making it the most scalable method for software development.

1.2 Microservice vs. monolith architectures

Microservice design is the polar opposite of monolith development. A monolith is one big codebase ("the kitchen sink") that implements all functionalities. Everything is in one place, and no single component can work in isolation.

On the plus side, monoliths are easy to get up and running. Airbnb, to give an example, started with The Monorail, a Ruby on Rails application. While the company was still small, developers could iterate fast. Making broad changes was easy as the relationships between the different parts of the monolith were transparent.

As a company grows and teams increase in size, however, monolith development becomes troublesome. Soon, the system can no longer fit in a single head — there are just too many moving parts, so things slow down.

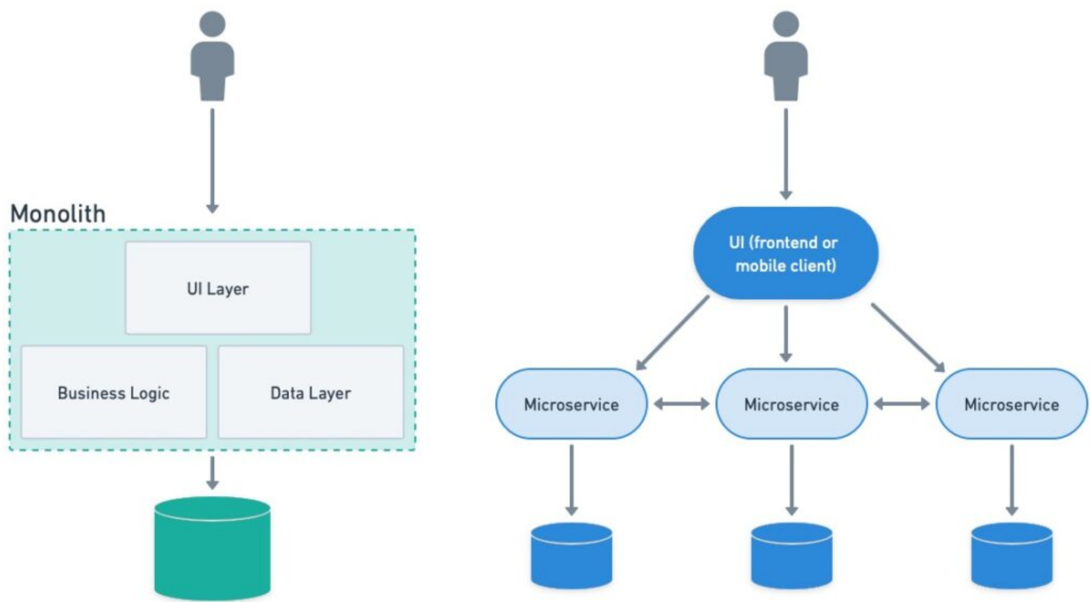

Figure 1: Monolith vs microservice architecture

1.3 Benefits of microservices

Microservices allow companies to keep teams small and agile. The idea is to decompose the application into small services that can be autonomously developed and deployed by tightly-knitted teams.

1.3.1 Scalability

The main reason that companies adopt microservices is **scalability**. Services can be developed and released independently without arranging large-scale coordination efforts within the organization.

1.3.2 Fault isolation

A benefit of having a distributed system is the ability to avoid single failure points. You can deploy microservices in different availability zones with cloud-enabled technologies, ensuring that your users never experience an outage.

1.3.3 Smaller teams

With microservices, the development team can stay small and cohesive. The smaller the group, the less communication overhead and the better the collaboration.

Amazon, as an example, takes team size to the extreme with their two pizza teams. Meaning that a team should be small enough to be fed by two pizzas.

1.3.4 The freedom to choose the tech stack

With a monolith, language and tech stack options are pretty much set in stone from the beginning. New developers must adapt to whatever choices were made in the past.

In contrast, each microservice can use the tech stack that is most appropriate for solving the task at hand. Thus, the team can pick the best tool for the job and based on their skills. For example, you can implement a high-performing service in Go or C and a high-tolerance microservice with Elixir.

1.3.5 More frequent releases

The development and testing cycle is shorter as small teams iterate quick. And, because they can also deploy their updates at any time, microservices can be updated much more frequently than a monolith.

1.4 The caveats of microservice design

At first glance microservices sound wonderful. They are modular, scalable, and fault tolerant. A lot of companies have had great success using this model, so microservices might naturally seem to be the superior architecture and the best way to start new applications.

However, most firms that have succeeded with microservices did not begin with them. Consider the examples of Airbnb and Twitter, which went the microservice route after outgrowing their monoliths and are now battling its complexities. Even successful companies that use microservices appear to still be figuring out the best way to make them work. It is evident that microservices come with their share of tradeoffs.

1.5 Microservice design challenges

Migrating from a monolith to microservices is also not a simple task, and creating an untested product as a new microservice is even more complicated. Microservices should only be seriously considered after evaluating the alternative paths. So, before embarking on a costly transition to microservices, we should talk about their shortcomings and limitations:

- **Small**: applies both to the team size and the codebase. A microservice must be small enough to be entirely understood by one person. As a rule of thumb, a microservice is too big if it would take it more than one sprint to rewrite it from scratch.
- **Focused on one thing**: a microservice must focus on one aspect of the problem or perform only one task.
- **Autonomous**: each microservice has its own database or persistence layer that is not shared with other services.
- **Aligned with the bounded context**: in software, we create models to represent the problem we want to solve. A bounded context represents the limits of a given model. Contexts are natural boundaries for services, so finding them is the most difficult and crucial part of designing a good microservice architecture.
- **Loosely-coupled**: while microservices can depend on other microservices, we must be careful about how they communicate. Each time a bounded context is crossed, some level of abstraction and translation is needed to prevent behavior changes in one service from affecting too much the rest.
- **Independently deployable**: being autonomous and loosely-coupled, a team can deploy their microservice with little external coordination or integration testing. Microservices should communicate over well-defined APIs and use translation layers to prevent behavior changes in one service from affecting the others.

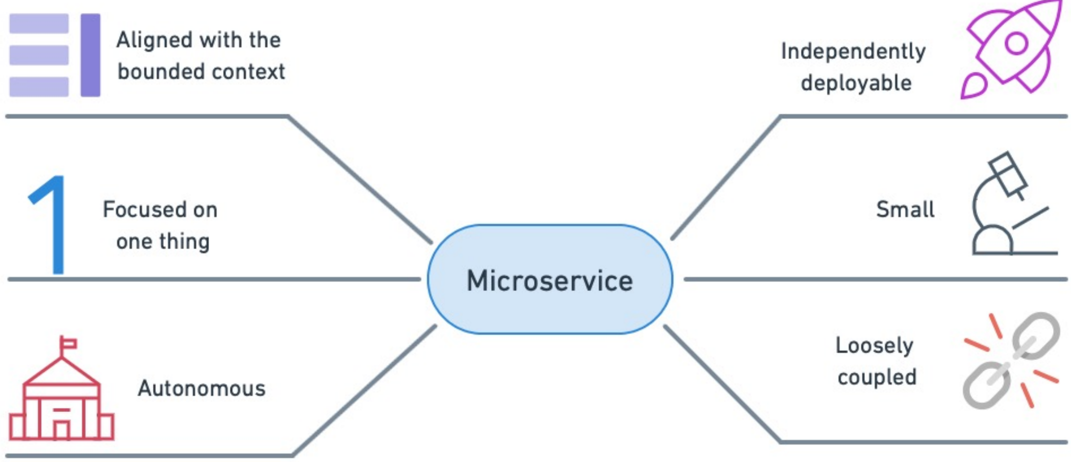

Figure 2: The key properties of microservice architecture

1.6 Reasons not to migrate to microservices

The caveats and strict limitations make microservices a bad fit for some types of workloads and applications. Let's see some common cases where microservices architecture is not recommended.

1.6.1 Microservices are only viable for mature products

On the topic of starting a new project with microservices, Martin Fowler observed that:

> 1. Almost all the successful microservice stories started with a monolith that got too big and was broken up.
> 2. Almost all the cases where a system that was built as a microservice system from scratch, ended up in serious trouble.
>
> This pattern has led many to argue that you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile.

The crux of the matter is that the first design is rarely optimal. The first few iterations of any new product are spent finding what users really need. Therefore, success hinges on staying agile and being able to quickly improve, redesign, and refactor. In this regard, microservices are manifestly worse than a monolith. If you don't nail the initial design, you're in for a rough start, as it's much harder to refactor a microservice than a monolith.

1.6.2 Microservices are not a good fit for startups

As a startup, you already are running against the clock, looking for a breakthrough before running out of capital. You don't need the scalability at this point (and probably not for a few years yet), so why make things harder by using a complicated architecture model?

A similar argument can be made when working on greenfield projects, which are unconstrained by earlier work and hence have nothing upon which to base decisions. Sam Newman, author of *Building Microservices: Designing Fine-Grained Systems*, stated that it is very difficult to build a greenfield project with microservices:

> I remain convinced that it is much easier to partition an existing "brownfield" system than to do so upfront with a new, greenfield system. You have more to work with. You have code you can examine, you can speak to people who use and maintain the system. You also know what 'good' looks like – you have a working system to change, making it easier for you to know when you may have got something wrong or been too aggressive in your decision-making process.

1.6.3 Microservices aren't the best for On-Premise installations

Microservice deployment needs robust automation because of all the moving parts involved. Under normal circumstances, we can rely on continuous deployment pipelines for the job.

This won't fly for on-premise environments, where developers publish a package and it's up to the customer to manually deploy and configure everything on their own. Microservices make all these tasks especially challenging, so this is a release model that does not fit nicely with microservice architecture.

To be clear, developing an On-Premise microservice application is entirely viable. Semaphore is accomplishing just that with Semaphore On-Premise. However, as we realized along the way, there are several challenges to overcome. Consider the following before deciding to adopt microservices design for On-Premise installations:

- Versioning rules for On-Premise microservices are more stringent. You must carefully track each individual microservice that participates in a release.
- You must carry out thorough integration and end-to-end testing, as you can't test in production.
- Troubleshooting a microservice application is substantially more difficult without direct access to the production environment.

1.6.4 If it's working, don't fix it

If we measure productivity as the number of value-adding features implemented over time, then it follows that switching architecture while productivity is strong makes little sense.

Teams working on monoliths tend to be more productive initially. Only after the monolith grows in complexity, microservices appear as a viable alternative. So, it's best to stick with monoliths until the point where productivity decreases.

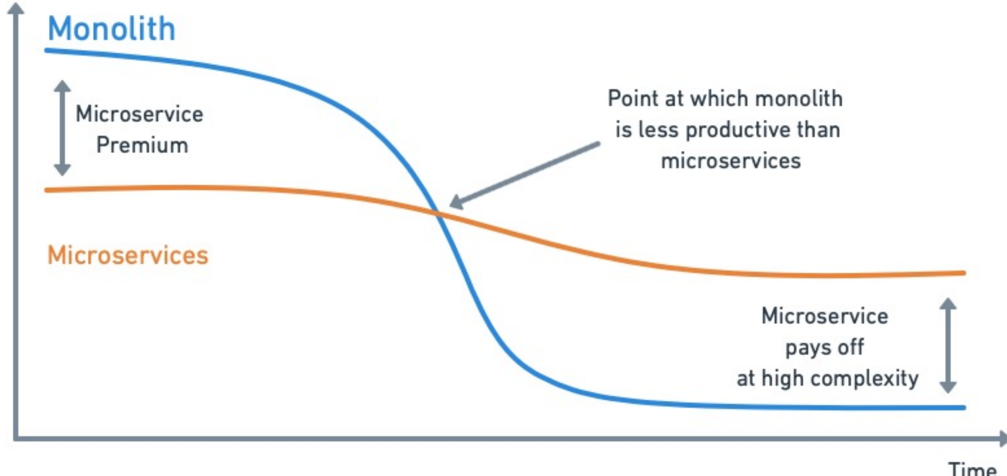

Figure 3: Microservices are initially the less productive architecture due to maintenance overhead. As the monolith grows, it gets more complex, and it's harder to add new features. Microservice only pays off after the lines cross.

1.6.5 Brooke's Law and developer productivity

In *The Mythical Man Month* (1975), Fred Brook Jr. stated that "adding manpower to a late software project only makes things worse". This happens because new developers must be mentored before they can work on a complex codebase. Also, as the team grows, the communication overhead increases. It's harder to get organized and make decisions.

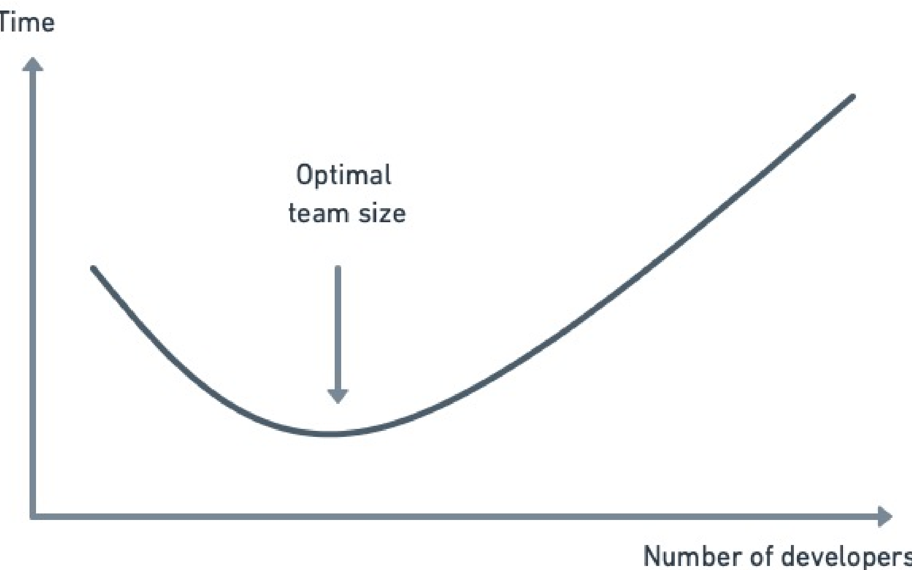

Figure 4: Brook's law applied to complex software development states that adding more developers to a late software project only makes it take longer.

Microservices are one method of reducing the impact of Brooke's Law. You get smaller, more agile and communicative teams. Before deciding on using microservices, however, you must determine if Brooke's Law is affecting your team. Switching to microservices too soon would not be a wise investment.

1.6.6 You may not be prepared for the transition

Some conditions must be met before you can begin working with microservices. Along with preparing your monolith, you'll need to:

- Set up continuous integration and continuous delivery for automatic deployment.
- Implement quick provisioning to build infrastructure on demand.
- Learn about cloud-native tech stacks, including containers, Kubernetes, and serverless.
- Get acquainted with Domain-Driven Design, Test-Driven Development, and Behavior-Driven Development.
- Reorganize the teams to be cross-functional, removing silos and flattening hierarchies to allow for innovation.
- Foster a DevOps culture in which the lines between developer and operations jobs are blurred.

Changing the culture of an organization can take years. Learning all that there is to know will take months. Without preparation, transitioning to microservices is unlikely to succeed.

We'll talk more about restructuring organizations in the next chapter.

1.7 Is it the right time for the switch?

Microservices are the most scalable way we have to develop software, no doubt about that. But they are not free lunches. They come with some risks that are easy to run afoul of if you're not cautious. They are great when the team is growing and you need to stay fast and agile. But you need to have a good understanding of the problem to solve, or you can end up with a distributed monolith.

We can summarize this whole discussion about transitioning to microservices in one sentence: don't do it unless you have a good reason. Companies that embark on the journey to microservices unprepared and without a solid design will have a very tough time. You need to achieve a critical mass of engineering culture and scaling know-how before microservices should be considered as an option.

1.8 Revitalizing monoliths

You've downloaded this book because you're interested in microservices. Presumably, because you are not satisfied with your monolith. As an alternative to microservices, let's discuss a few ways in which you can revitalize your monolith and squeeze a few more good years out of it.

There are two moments in the lifetime of a project in which microservices might seem the only way forward:

- **Tangled codebase**: it's hard to make changes and add features without breaking other functionality.
- **Performance**: you're having trouble scaling the monolith.

There are ways to address both problems.

1.9 The modular monolith as an alternative to microservices

A common reason developers want to avoid monoliths is their proclivity to deteriorate into a tangle of code (the "big ball of mud"). It's challenging to add new features when we get to this point since everything is interconnected.

But a monolith does not have to be a mess. Take the example of Shopify: with over 3 million lines of code, theirs is one of the largest Rails monoliths in the world. At one point, the system grew so large it caused much grief to developers:

> The application was extremely fragile with new code having unexpected repercussions. Making a seemingly innocuous change could trigger a cascade of unrelated test failures. For example, if the code that calculates our shipping rate is called into the code that calculates tax rates, then making changes to how we calculate tax rates could affect the outcome of shipping rate calculations, but it might not be obvious why. This was a result of high coupling and a lack of boundaries, which also resulted in tests that were difficult to write, and very slow to run on CI.

Instead of rewriting their entire monolith as microservices, Shopify chose modularization as the solution.

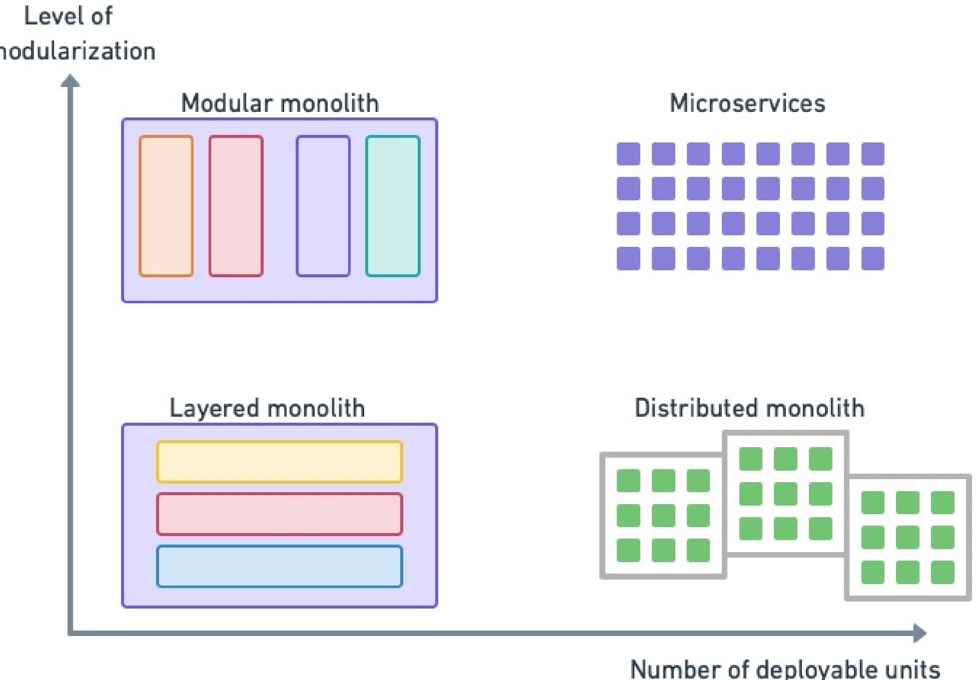

Figure 5: Modularization helps design better monoliths and microservices. Without carefully defined modules, we either fall into the traditional layered monolith (the big ball of mud) or, even worse, as a distributed monolith, which combines the worst features of monoliths and microservices.

Modularization is a lot of work, that's true. But it also adds a ton of value because it makes development more straightforward. New developers do not have to know the whole application before they can start making changes. They only need to be familiar with one module at a time. Modularity makes a large monolith feel small.

Modularization is a required step before transitioning to microservices, and for some, it may be a better solution than microservices. The modular monolith, like in microservices, solves the tangled codebase problem by splitting the code into independent modules. Unlike with microservices, where communication happens over a network, the modules in the monolith communicate over internal API calls.

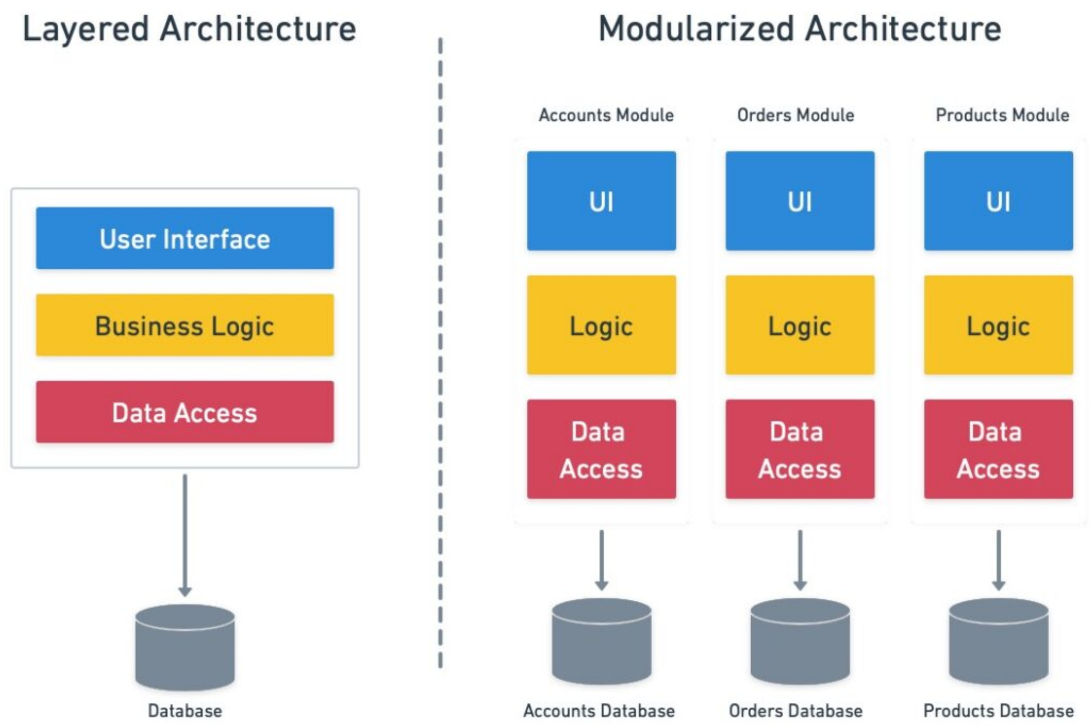

Figure 6: Layered vs modular monoliths. Modularized monoliths share many of the characteristics of microservice architecture sans the most difficult challenges.

1.10 Scalable monoliths

Another misconception about monoliths is that they can't scale. If you're experiencing performance issues and think that microservices are the only way out, think again. Shopify has shown us that sound engineering can make a monolith work on a mind-boggling scale.

Figure 7: Shopify bragging about their Black Friday stats

The architecture and technology stack will determine how the monolith can be optimized; a process that almost invariably will start with modularization and can leverage cloud technologies for scaling:

- Deploying multiple instances of the monolith and using load balancing to distribute the traffic.
- Distributing static assets and frontend code using CDNs.
- Using caching to reduce the load on the database.
- Implementing high-demand features with edge computing or serverless functions.

Chapter 2 — How to Restructure Your Organization for Microservices

When companies think about how to restructure their organizations, they often focus on the new roles that must be filled and the skills that employees need to learn. However, restructuring your organization to support microservice-based applications goes beyond a few roles and job titles. A company restructuring for microservices requires an entire culture shift and new way of working.

2.1 Hierarchical organizations

In order to take advantage of much of the value of a service-oriented architecture, you must change your traditional hierarchical organizational structure to a more horizontal one.

In a traditional hierarchical organization, such as that shown in Figure 8, your engineering company is organized around roles and job functions. Here, multiple development teams are created, and each team is responsible for building part of the application. Once they've completed their assigned job functions, responsibility is handed to the next group, often a QA group, which performs testing and the remainder of their job functions. Finally, responsibility is transferred to an IT operations team for hosting and operating the application. Additionally, other groups have their own roles and job functions, such as security, which is responsible for keeping the product and the company safe and secure.

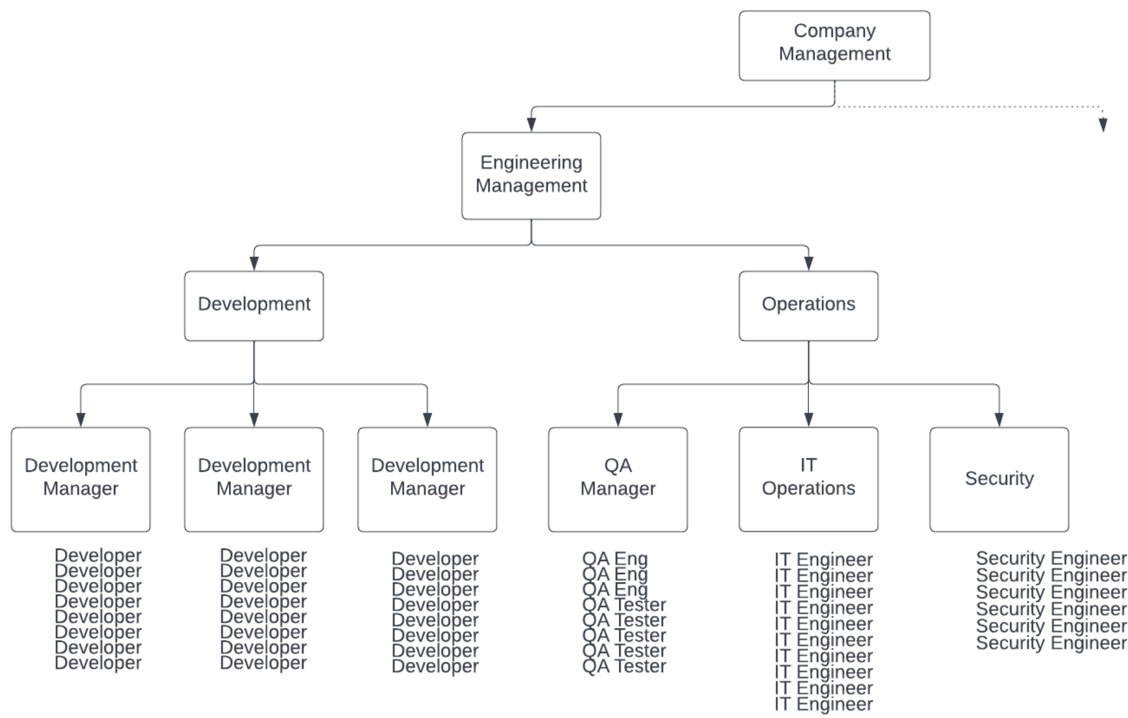

Figure 8: Traditional hierachical organization structure.

Everyone has their own defined job function. Everyone has their assigned roles. The problem is, nobody is *responsible* for the product as a whole. Nobody *owns* the application. Organizationally, you have to go all the way to the highest level of engineering management—such as the VP of engineering or CTO/CPO—before you find someone who owns and manages the product *as a whole*. This type of structure leads to finger-pointing and a "not-my-problem" mentality.

Everyone has a *role* to fill, but no one has *responsibility*.

When you build your application using microservices, one of the advantages is the ability to define and manage smaller chunks of the application as a whole. This advantage isn't as useful when you keep the traditional organizational structure. You have just moved from having one large application **with no owner**, to hundreds of smaller applications **with no owners**.

To fully take advantage of the structural benefits of a microservice application architecture, you must modify your organizational structure to match that model. Most importantly, you must move from a *roles and job functions* assignment model to a *ownership and responsibility* assignment model.

2.2 The pod model

In the pod model, your organization is not split by job functions; instead, it's split into small, cross-functional teams, called pods. Each team has the capabilities, the resources, and the support required to be *completely responsible* for a small portion of the overall application—a service or two in the microservice architecture model.

A pod that owns microservices within the application typically consists of 6-10 people with the following types of job skills:

- Team management
- Software development
- Software validation
- Service operation
- Service scaling and maintaining availability
- Security
- Operational support and maintenance
- Infrastructure operational maintenance (servers, etc.)

It's important that the team has the necessary skills to perform these jobs. But, in a pod model, the pod as a whole has responsibilities, and no single person is assigned specific *job functions*. In other words, there is no "security person," or "DevOps person," or "QA person" in the pod. Instead, everyone in the pod shares the entire pod's responsibilities.

Figure 9 shows the same organization using a pod model. The pods are each independent and peers of one another, and each pod provides cross-functional responsibilities.

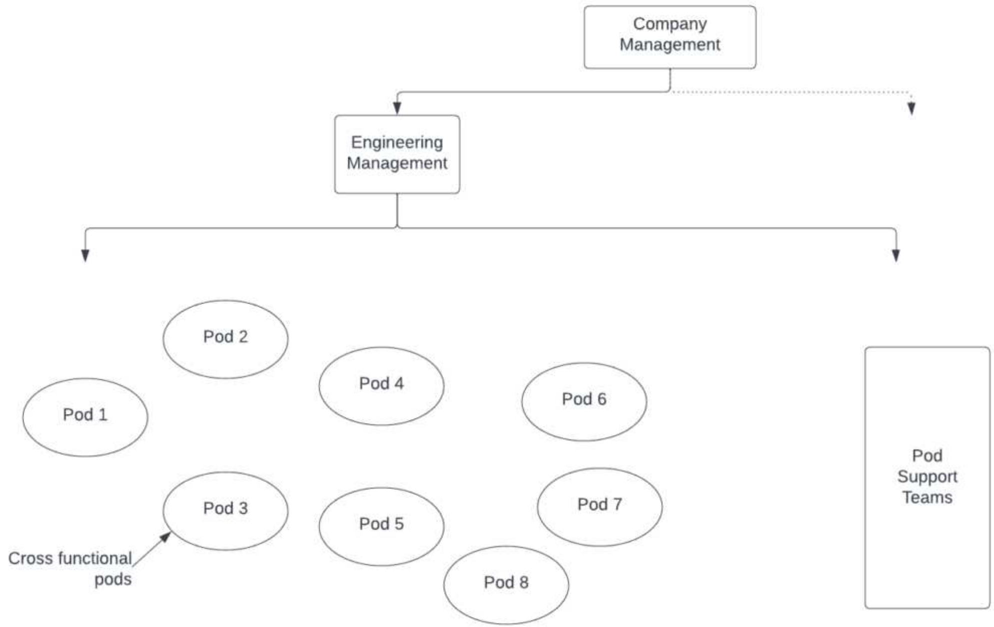

Figure 9: Pod-based organizational structure

2.3 Ownership is the key

The key to successfully operating the pod model is to create pods with responsibilities that aren't specific job functions. Rather, their responsibilities are *ownership*. A pod *owns* a service or set of services. Ownership means they have complete responsibility for the architecture, design, creation, testing, operation, maintenance, and security of the service. No one else has ownership, only the assigned pod. If anything happens to that service, it is the pod's responsibility to manage and fix. This completely removes the ability to finger-point to another team when a service fails. The service's owning pod is the one responsible. This is illustrated in Figure 10, where interconnected services are represented in blue, and the pods that own those services are shown in red.

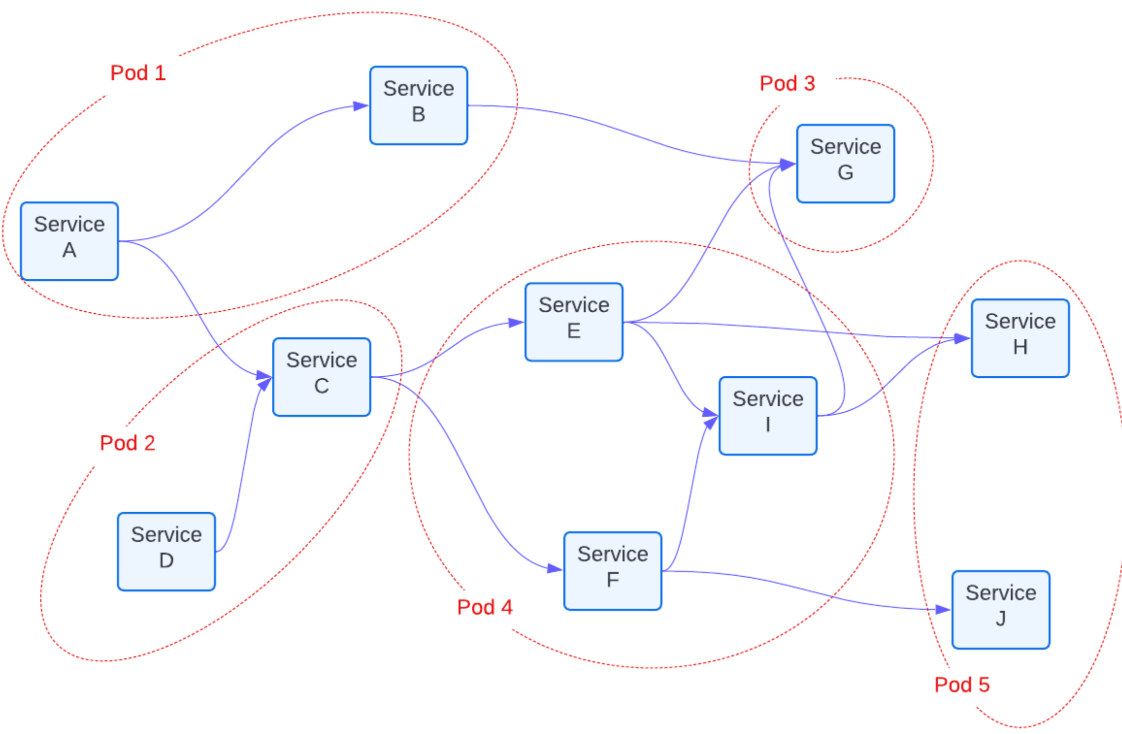

Figure 10: Every service has an owner.

Every service has exactly one owner, and if something is failing in a service, it is completely clear which pod is responsible for resolving the issue.

2.4 Cross-service finger-pointing

But what happens if problems cross service boundaries? For example, what happens if Service E in Figure 10 is causing problems for Service C? In that case, it may appear that both services are having problems, and it may not be clear where the root cause of the problem

resides. Because the two services are owned by different pods, which pod owns the problem? The answer may be difficult and complex to determine. Finger pointing between Pod 2 and Pod 4 is definitely a possibility.

If you have successfully set up a pod model and have ingrained a strong *ownership* mindset into the members of the pods, the likelihood of finger-pointing in this case should be low. What *should* happen in a high-quality team organization is both Pod 2 and Pod 4 work together to resolve the issue.

Although this is the way things *should* work, that's not sufficient. The model must help resolve these ownership issues quickly and decisively in order to keep your application working, at scale, and maintain high availability. This is where two characteristics of your microservice architecture are critical: **Well-designed and documented APIs** and **solid, maintainable SLAs**. Not everyone who promotes moving to microservice architectures drives these two characteristics; but in my mind, they are the two most important characteristics of a solid microservice architecture, and they are critical to the *ownership* organizational model. Let's look at these two microservice characteristics:

- **Well-designed and documented APIs**. Each and every service in your application must have a well-designed API describing how the service should be used and how to talk to it, and this API must be well-documented across the organization. We are used to well-designed and documented APIs when we are talking about APIs exposed to customers. But it's equally important to design quality APIs among internal services as well. **No service should talk to any other service without using a well-defined and documented API to that service**. This makes sure that expectations on what each service does and does not do is clear, and those expectations drive higher-quality interactions and hence fewer application issues.
- **Solid, maintainable SLAs**. Besides having APIs, a set of performance expectations around those APIs must be established. If Service C is calling Service E's API, it's critical that Service C understand the performance expectations it can expect from Service E. What will the latency be for the API calls it makes? What happens to latency if the call rate increases? Are there limits on how often a service can be called? What happens when those limits are reached?

APIs are about *understanding*, and SLAs are about *expectations*. APIs help us know what other services do and what they are responsible for. SLAs help us know what we can expect from a performance standpoint from the service.

If Service E in Figure 10 has a well-defined and documented API, and has well-defined SLAs on how it should be used and it keeps those SLAs, then as long as Service C is using the service in accordance with the documented API and keeping within the defined SLAs, Service C should be able to expect reasonable performance from Service E.

Now, in the hypothetical example above, Service E was causing problems for Service C. In this case, it should be obvious in the measured performance compared with the documented SLAs that Service E has the problem and not Service C. By using monitoring, and API/SLA management, diagnosing problems becomes far easier.

2.5 Pods need support

In the pod model, pods have a lot of responsibility and a lot of authority. There is no way that a small team (6-10 people) that composes a pod can handle all aspects of the breadth and depth of responsibility for all aspects of service ownership without support.

To give them support, horizontal service teams are created to provide tools and support to the service-owning pods. These teams can handle common pod-independent problems such as creating CI/CD pipelines, understanding global security issues, creating tooling to manage infrastructures, and maintaining vendor relationships. The pods can then leverage these teams to augment the pod and give support to the pod. This is illustrated in Figure 20.

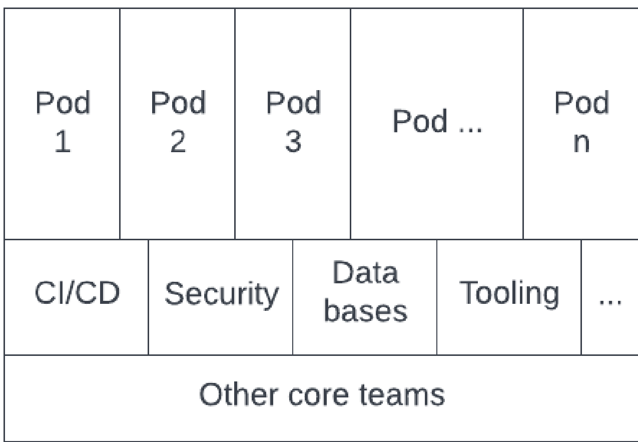

Figure 11: Support teams assisting pods.

It's important to note that these support teams are *supporting the pods*, and do not—can not—take ownership responsibility away from the pods. If a security issue exists in a service, responsibility for the issue lies with the pod that owns the service—not with the security support team. The pods have ultimate control and decision-making responsibilities—and hence ultimate responsibility—for all aspects of the operation of the services they own.

2.6 The STOSA model

Moving to a service/microservice architecture for your application architecture is a valuable tool to building and managing large, complex applications. However, just changing the architecture is not sufficient. You must update your organization structure to support the new architecture model or you won't be able to effectively utilize the advantages service architectures offer. Without also organizing your teams around these changes, you risk falling back into old habits and processes that will result in lack of ownership and responsibility, and the same general problems you had before you moved to the service architecture.

The pod ownership model is part of the STOSA framework. STOSA stands for *Single Team Oriented Service Architecture*. It defines a model where service teams — pods — own all aspects of building and operating individual services.

The model was developed and introduced in Lee Atchison's book *Architecting for Scale*. It's now available as a standalone model documented at stosa.org. We recommend checking it out.

Chapter 3 — Design Principles for Microservices

How do you know if you're doing proper microservice design? If your team can deploy an update at any time without coordinating with other teams, and if other teams can similarly deploy their changes without affecting you, congratulations, you got the knack of microservices.

The surest way of losing the benefits microservices offer is by not respecting the decoupling rule. If we look closely, we see that microservices are all about autonomy. When this autonomy is lost, teams must coordinate during development and deployment. Perfect integration testing is required to make sure all microservices work together.

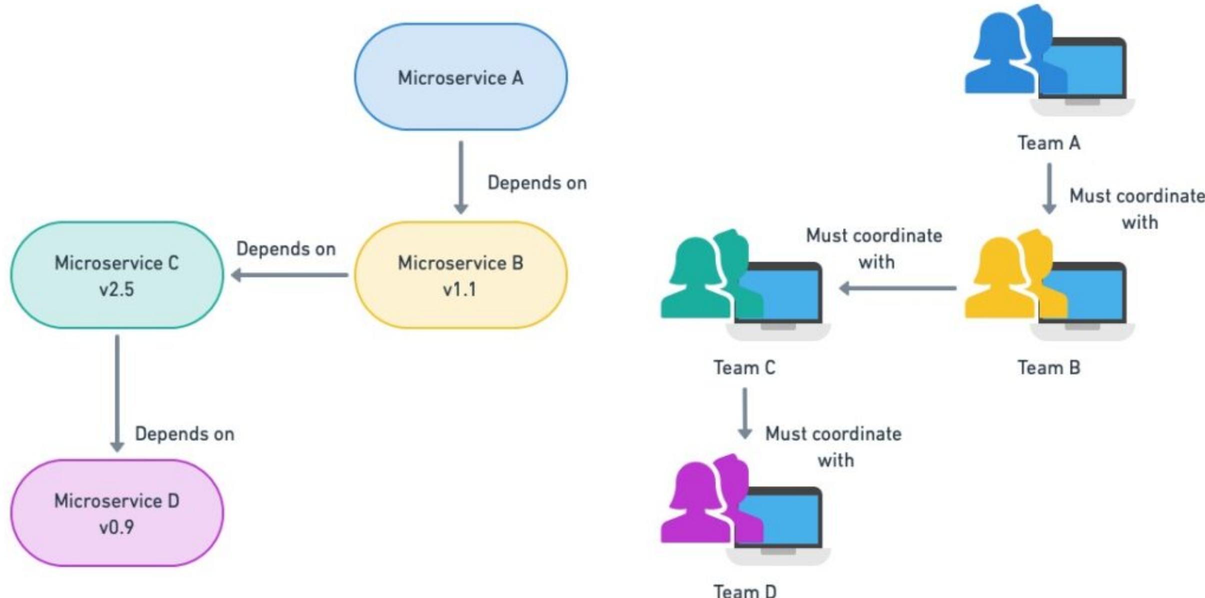

Figure 12: Tight service dependencies create team dependencies and communication bottlenecks.

These are all problems that come with distributed computing. If you've ever used a cloud service you'll know that spreading services or machines over many geographical locations is not the same as running everything on the same site. A distributed system has a higher latency, can have synchronization issues, and is a lot harder to manage and debug. This highly-coupled service architecture is really, deep down, a *distributed monolith*, with the worst of both worlds and none of the benefits microservices should bring.

If you cannot deploy without coordinating with another team or relying on specific versions of other microservices to deploy yours, you're only distributing your monolith.

Domain-Driven Development allows us to plan a microservice architecture by decomposing the larger system into self-contained units, understanding the responsibilities of each, and identifying their relationships.

3.1 What is Domain-Driven Design?

Domain-Driven Design (DDD) is a software design method wherein developers construct models to understand the business requirements of a domain. These models serve as the conceptual foundation for developing software.

According to Eric Evans, author of *Domain-Driven Design: Tackling Complexity in the Heart of Software*, a domain is:

> *A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.*

How well one can solve a problem is determined by one's capacity to understand the domain. Developers are smart, but they can't be specialists in all fields. They need to collaborate with domain experts to guarantee that the code is aligned with business rules and client needs.

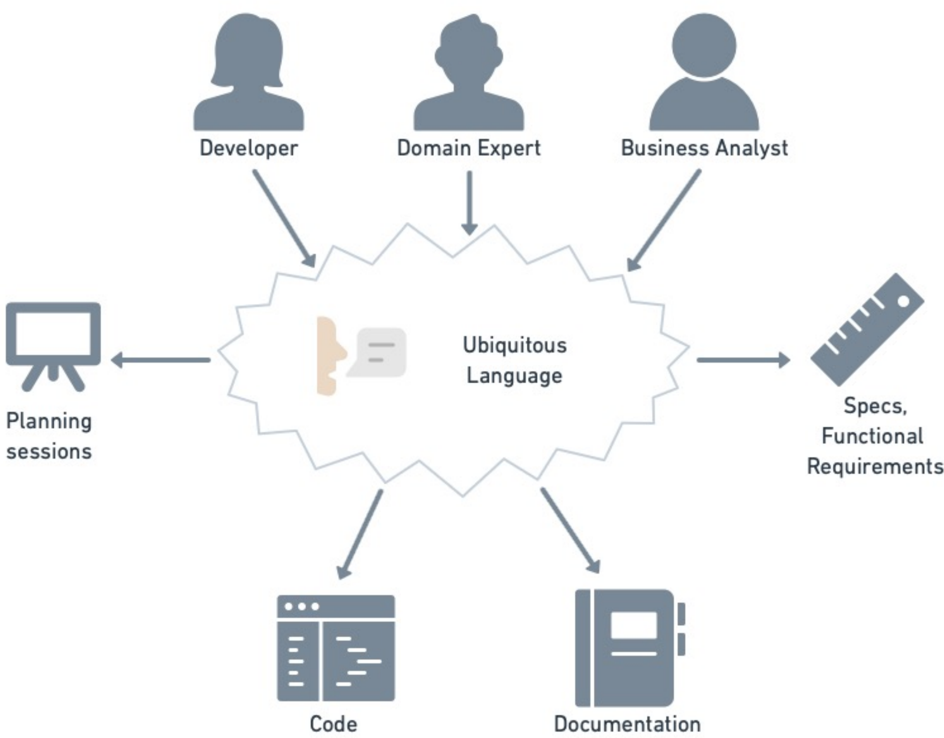

Figure 13: Developers and domain experts use a unified language to share knowledge, document, plan, and code.

The two most important DDD concepts for microservice architecture are: *bounded contexts* and *context maps*.

3.1.1 Bounded Context (BC)

The setting in which a word appears determines its meaning. Depending on the context, "book" may refer to a written piece of work, or it may mean "to reserve a room". A *bounded context* (BC) is the space in which a term has a definite and unambiguous meaning.

Before DDD it was common practice to attempt to find a model that spanned the complete domain. The problem is that the larger the domain, the more difficult it is to find a consistent and unified model. DDD's solution is to break down the domain into more manageable subdomains.

Figure 14: The relevant properties of the "book" change from context to context.

In software, we need to be exact. That is why defining BCs is critical: it gives us a precise vocabulary, called *ubiquitous language*, that can be used in conversations between developers and domain experts. The ubiquitous language is present throughout the design process, project documentation, and code.

3.1.2 Context Map

The presence of a BC anticipates the need for communication channels. For instance, if we're working in an e-commerce domain, the salesperson should check with inventory before selling a product. And once it's sold, it's up to shipping to ensure delivery of the product to the correct address. In DDD, these relationships are depicted in the form of a *context map*.

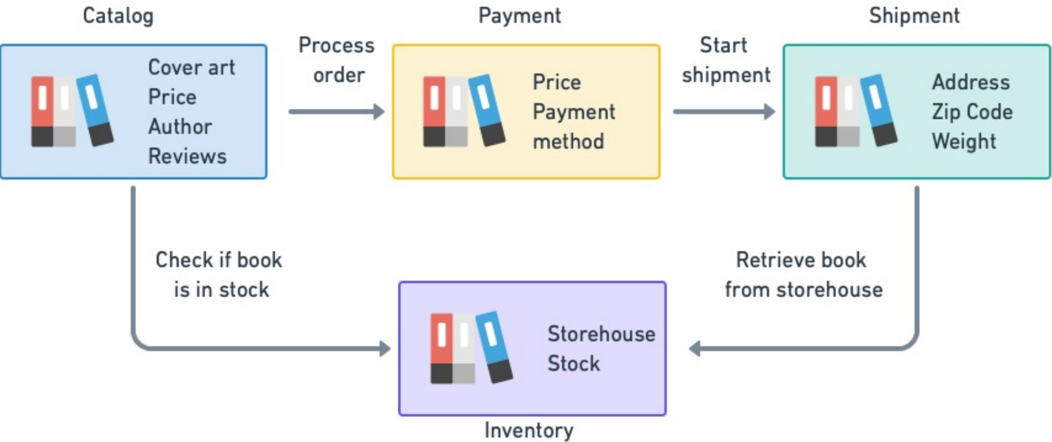

Figure 15: Bounded context communication used to achieve a high-level task.

3.2 Domain-Driven Design for microservices

DDD takes place in two phases:

1. In the *strategic phase* we identify the BCs and map them out in a context map.
2. In the *tactical phase* we model each BC according to the business rules of the subdomain.

Let's see how each phase plays a role in microservice architecture design.

3.3 Strategic phase

During this phase, we invite developers, domain experts, product owners, and business analysts to brainstorm, share knowledge and make an initial plan. With the aid of a facilitator, this can take the form of an Event Storming workshop session, where we build models and identify business requirements starting from significant events in the domain.

Figure 16: An Event Storming session, domain events are used as the catalyst for sharing knowledge and identifying business requirements.

In strategic DDD, we take a high-level, top-to-bottom approach to design. We begin by analyzing the domain in order to determine its business rules. From this, we derive a list of BCs.

Figure 17: Strategic Domain-Driven Design helps us identify the logical boundaries of individual microservices.

The boundaries act as natural barriers, protecting the models inside. As a result, every BC represents an opportunity to implement at least one microservice.

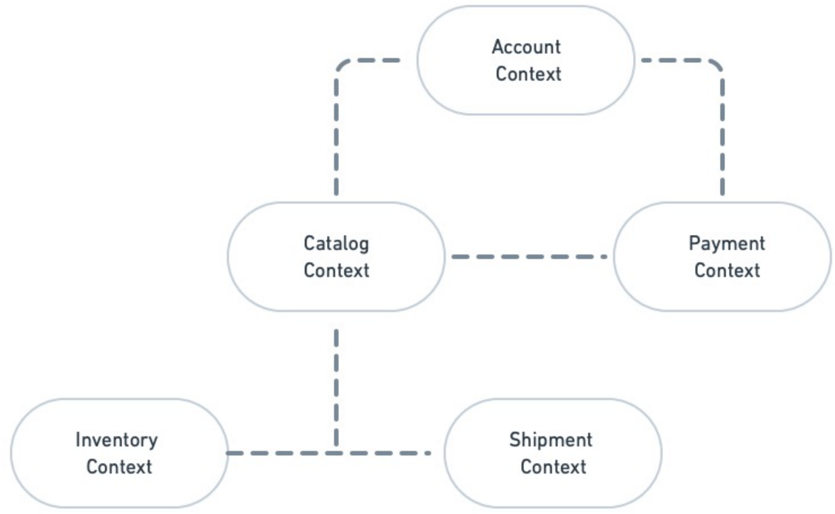

Figure 18: Bounded relationships

3.3.1 Types of relationships

Next, we must decide how BCs will communicate. Eric Evans lists seven types of relationships, while other authors list six of them. Regardless of how we count them, at least three (shared kernel, customer/supplier, and conformist) imply tight coupling, which we do not want in a microservice design and can be ignored. That leaves us with four types of relationships:

- **Open Host Service** (OHS): the service provider defines an open protocol for others to consume. This is an open-ended relationship, as it is up to the consumers to conform to the protocol.
- **Published Language** (PL): this relationship uses a well-known language such as XML, JSON, GraphQL, or any other fit for the domain. This type of relationship can be combined with OHS.
- **Anticorruption Layer** (ACL): this is a defensive mechanism for service consumers. The anti-corruption layer is an abstraction and translation wrapping layer implemented in front of a downstream service. When something changes upstream, the consumer service only needs to update the ACL.
- **Separate ways**: this happens when integration between two services is found, upon further analysis, to be of little value. This is the opposite of a relationship — it means that the BCs have no connection and do not need to interact.

At the end of our strategic DDD analysis, we get a context map detailing the BCs and their relationships.

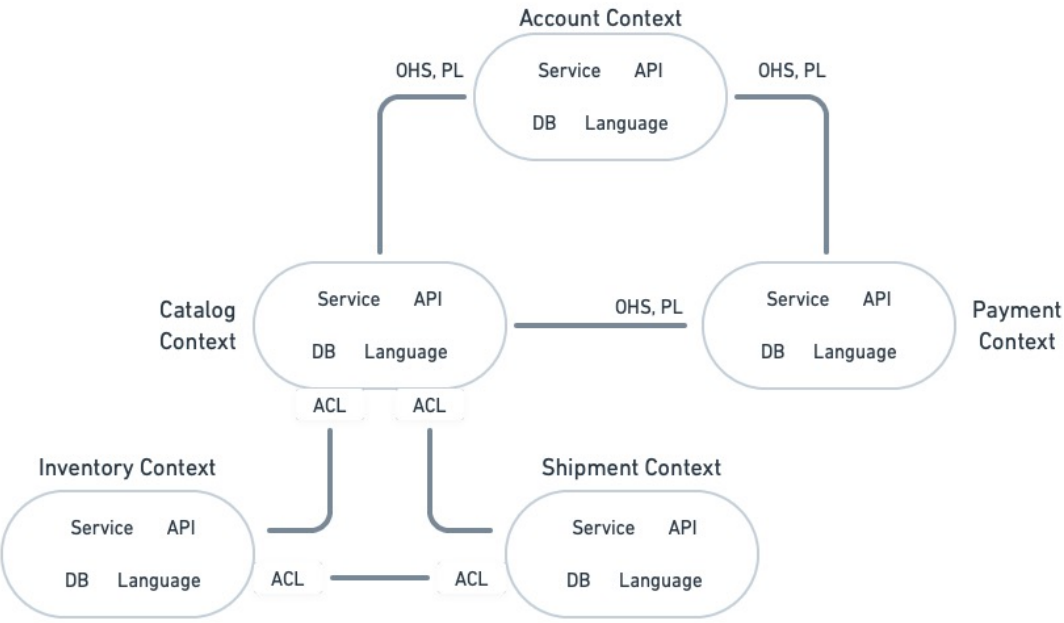

Figure 19: ACL is implemented downstream to mitigate the impact of upstream changes. OHS does the opposite. It's implemented upstream to offer a stable interface for services downstream.

3.4 Tactical phase

Deep down, software development is a modeling exercise; we describe a real-life scenario as a model and then solve it with code. In the previous stage, we identified BCs and mapped their relationships. In this stage we zoom in on each context to construct a detailed model.

The models created with DDD are technology-agnostic — they do not say anything about the stack underneath. We focus, instead, on modeling the subdomain. The main building block of our models are:

- **Entities**: entities are objects with an identity that persists over time. Entities must have a unique identifier (for example, the account number for a customer). While entity identifiers may be shared among context boundaries, the entities themselves don't need to be identical across every BC. Each context is allowed to have a private version of a given entity.
- **Value objects**: value objects are immutable values without identity. They represent the primitives of your model, such as dates, times, coordinates, or currencies.
- **Aggregates**: aggregates create relationships between entities and value objects. They represent a group of objects that can be treated as a single unit and are always in a consistent state. For example, customers place orders and own books, so the entities customer, order, and book can be treated as an aggregate. Aggregates must always be referenced by a main entity, called the *root entity*.
- **Domain services**: these are stateless services that implement a piece of business logic

or functionality. A domain service can span multiple entities.

- **Domain events**: essential for microservice design, domain events notify other services when something happens. For instance, when a customer buys a book, a payment is rejected, or that a user has logged in. Microservices can simultaneously produce and consume events from the network.
- **Repositories**: repositories are persistent containers for aggregates, typically taking the form of a database.
- **Factories**: factories are responsible for creating new aggregates.

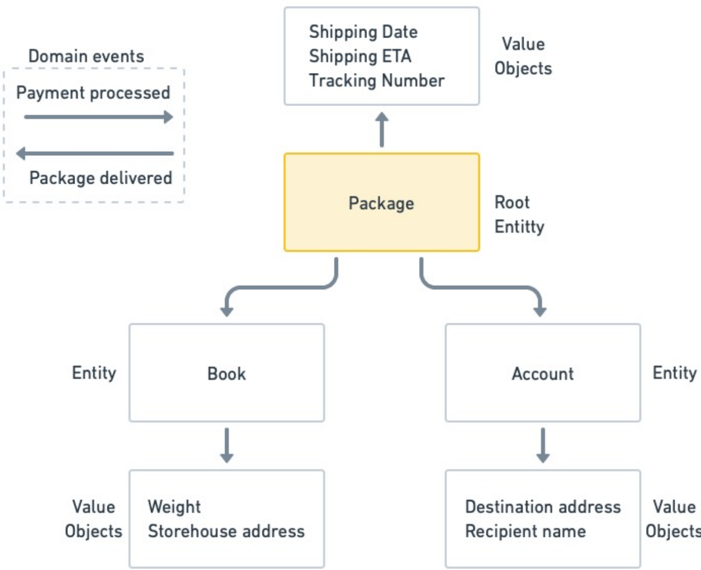

Figure 20: The shipping aggregate consists of a package containing books shipped to an address.

3.5 Domain-Driven Design is iterative

While it may appear that we must first write an exhaustive description of the domain before we can begin working on the code, the reality is that DDD, like all software design, is an iterative process.

On paper, bounded contexts and context maps may appear OK, but when implemented, they may translate into services that are too big to be rightly called microservices. Conversely, chatty microservices with overlapping responsibilities may need to be merged into one.

As development progresses and you have a better understanding of the domain, you'll be able to make better judgments, enhance models, and communicate more effectively.

3.6 Complementary design patterns

DDD is undoubtedly a theory-heavy design pattern. As a result, it is mostly used for designing complex systems.

Other methods such as Test-Driven Development (TDD) or Behavior-Driven Development (BDD) may be enough for smaller, simpler systems. TDD is the fastest to start with and works best when on single microservices or even on applications consisting of only a few services.

On a bigger scale, we can use BDD, which forces us to validate the wholesale behavior with integration and acceptance tests. BDD may work well if you work on low to medium-complexity designs.

You can also combine these three patterns, choosing the best one for each stage of development. For example:

1. Identify microservices and their relationships with strategic DDD.
2. Model each microservice with tactical DDD.
3. Since each team is autonomous, they can choose to adopt BDD or TDD (or a mix of both) for developing a microservice or a cluster of microservices.

DDD can feel daunting to learn and implement, but its value for developing a microservice architecture is well worth the effort. If you're interested in learning more, we recommend picking up the relevant books by Eric Evans and Vaughn Vernon.

Chapter 4 — From Monolith to Microservices

In the previous chapters, we discussed the downsides of microservices and examined ways of making a monolith remain viable despite growing pressures. The goal was never to dissuade you from microservices; only to consider all options before taking action. In this chapter, we'll talk about the signs crumbling monoliths show.

Overweight monoliths exhibit two classes of problems: degrading system performance and stability, and slow development cycles. So, whatever we do next comes from the desire to escape these technical and social challenges.

4.1 The single point of fragility

Today's typical large monolithic systems started off as web applications written in an MVC framework, such as Ruby on Rails. These systems are characterized by either being a single point of failure, or having severe bottlenecks under pressure.

Of course, having **potential** bottlenecks, or having an entire system that is a single point of failure is not inherently a problem. When you're in the third month of your MVP, this is fine. When you're working in a team of a few developers on a client project which serves 100 customers, this is fine. When most of your app's functionality are well-designed CRUD operations based on human input with a linear increase of load, things are probably going to be fine for a long time.

Also, there's nothing inherently wrong about big apps. If you have one and you're not experiencing any of these issues, there's absolutely no reason to change your approach. You shouldn't build microservices solely in the service of making the app smaller — it makes no sense to replace the parts that are doing their job well.

Problems begin to arise after your single point of failure has actually started failing under heavy load.

At that point, having a large attack surface can start keeping the team in a perpetual state of emergency. For example:

- An outage in non-critical data processing brings down your entire website. With Semaphore, we had events where the monolith was handling callbacks from many CI servers, and when that part of the system failed, it brought the entire service down.
- You moved all time-intensive tasks to one huge group of background workers, and keeping them stable gradually becomes a full-time job for a small team.
- Changing one part of the system unexpectedly affects some other parts even though they're logically unrelated, which leads to some nasty surprises.

As a consequence, your team spends more time solving technical issues than building cool and useful stuff for your users.

4.2 Slow development cycles

The second big problem is when making any change happen begins to take too much time.

There are some technical factors that are not difficult to measure. A good question to consider is how much time it takes your team to ship a hotfix to production. Not having a fast delivery pipeline is painfully obvious to your users in the case of an outage.

What's less obvious is how much the slow development cycles are affecting your company over a longer period of time. How long does it take your team to get from an idea to something that customers can use in production? If the answer is weeks or months, then your company is vulnerable to being outplayed by competition.

Nobody wants that, but that's where the compound effects of monolithic, complex code bases lead to.

- **Slow CI builds**: anything longer than a few minutes leads to too much unproductive time and task switching. As a standard for web apps we recommend setting the bar at 10 minutes. Slow CI builds are one of the first symptoms of an overweight monolith, but the good news is that a good CI tool can help you fix it. For example, on Semaphore you can split your test suite into parallel jobs.
- **Slow deployment**: this issue is typical for monoliths that have accumulated many dependencies and assets. There are often multiple app instances, and we need to replace each one without having downtime. Moving to container-based deployment can make things even worse, by adding the time needed to build and copy the container image.
- **High bus factor on the old guard, long onboarding for the newcomers**: it takes months for someone new to become comfortable with making a non-trivial contribution in a large code base. And yet, all new code is just a small percentile of the code that has already been written. The idiosyncrasies of old code affect and constrain all new code that is layered on top of the old one. This leaves those who have watched the app grow with an ever-expanding responsibility. For example, having five developers that are waiting for a single person to review their pull requests is an indicator of this problem.
- **Emergency-driven context switching**: we may have begun working on a new feature, but an outage has just exposed a vulnerability in our system. So, healing it becomes a top priority, and the team needs to react and switch to solving that issue. By the time they return to the initial project, internal or external circumstances can change and reduce its impact, perhaps even make it obsolete. A badly designed distributed system can make this even worse — hence one of the requirements for making one is having solid design skills. However, if all code is part of a single runtime hitting one database, our options for avoiding contention and downtime are very limited.
- **Change of technology is difficult**: our current framework and tooling might not be the best match for the new use cases and the problems we face. It's also common for monoliths to depend on outdated software. For example, GitHub upgraded to Rails 3 four years after it was released. Such latency can either limit our design choices, or generate additional maintenance work. For example, when the library version that you're using is no longer receiving security updates, you need to find a way to patch it

yourself.

4.3 Preparing your monolith for transitioning to microservices

A rewrite is never an easy journey, but by moving from monolith to microservices, you are changing more than the way you code; you are changing the company's operating model. As we've already mentioned, not only do you have to learn a new, more complex tech stack but management will also need to adjust the work culture and reorganize your teams.

4.4 A migration plan

It takes a lot of preparation to tear down a monolith, especially when the old system must remain operational while the transition is made.

The migration steps can be tracked with tickets and worked towards in each sprint like any other task. This not only helps in gaining momentum (to actually someday achieve the migration), but gives transparency to the business owners regarding how the team is planning on implementing such a large change.

During planning, you have to:

- Disentangle dependencies within the monolith.
- Identify the microservices needed.
- Design data models for the microservices.
- Develop a method to migrate and sync data between monolith and microservices databases.
- Design APIs and plan for backward compatibility.
- Capture the baseline performance of the monolith.
- Set up goals for the availability and performance of the new system.

Let's examine a few practices that will help you successfully make the transition.

4.4.1 Put everything in a monorepo

As you break apart the monolith, a lot of code will be moved away from it and into new microservices. A monorepo helps you keep track of these kinds of changes. In addition, having everything in one place can help you recover from failures more quickly.

In all likelihood, your monolith is already contained in one repository. So, it's just a matter of creating new folders for the microservices.

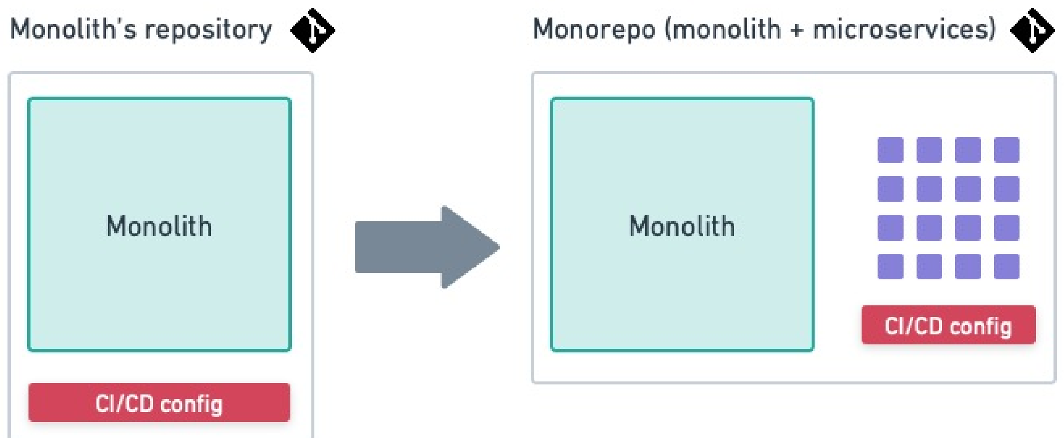

Figure 21: A monorepo is a shared repository containing the monolith and the new microservices.

4.4.2 Use a shared CI pipeline

During development, you'll not only be constantly shipping out new microservices but also re-deploying the monolith. The faster and more painless this process is, the more rapidly you can progress. Set up continuous integration and delivery (CI/CD) to test and deploy code automatically.

If you are using a monorepo for development, you'll have to keep a few things in mind:

- Keep pipelines fast by enabling change-based execution or using a monorepo-aware build tool such as Bazel or Pants. This will make your pipeline more efficient by only running changes on the updated code.
- Configure multiple promotions, one for each microservice and one more for the monolith. Use these promotions for continuous deployment.
- Configure test reports to quickly spot and troubleshoot failures.

4.4.3 Ensure you have enough testing

Refactoring is much more satisfying and effective when we are sure that the code has no regressions. Automated tests give the confidence to continuously ship out monolith updates.

An excellent place to start is the testing pyramid. You will need a good amount of unit tests, some integration tests, and a few acceptance tests.

The Test Pyramid

End to end

Integration

Unit

Figure 22: The testing pyramid.

Aim to run the tests as often on your local development machine as you do in your continuous integration pipeline.

4.4.4 Install an API Gateway or HTTP Reverse Proxy

As microservices are deployed, you have to segregate incoming traffic. Migrated features are provided by the new services, while the not-yet-ready functionality is served by the monolith.

There are a couple of ways of routing requests, depending on their nature:

- An API gateway lets you forward API calls based on conditions such as authenticated users, cookies, feature flags, or URI patterns.
- An HTTP reverse proxy does the same but for HTTP requests. In most cases, the monolith implements the UI, so most traffic will go there, at least at first.

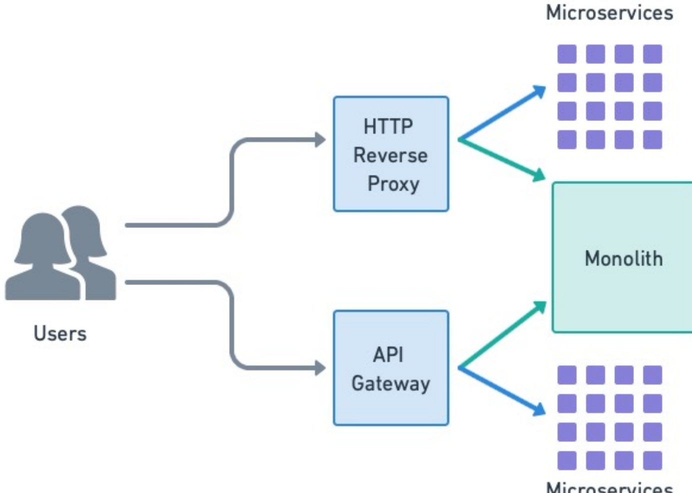

Figure 23: Use API gateways and HTTP reverse proxies to route requests to the appropriate endpoint. You can toggle between the monolith and microservices on a very fine-grained level.

Once the migration is complete, the gateways and proxies will remain – they are a standard component of any microservice application since they offer forwarding and load balancing. They can also function as circuit breakers if a service goes down.

4.4.5 Consider the monolith-in-a-box pattern

OK, this one only applies if you plan to use containers or Kubernetes for the microservices. In that case, containerization can help you homogenize your infrastructure. The monolith-in-a-box pattern consists of running the monolith inside a container such as Docker.

If you've never worked with containers before, this is a good opportunity to get familiar with the tech. That way, you'll be one step closer to learning about Kubernetes down the road. It's a lot to learn, so plan for a steep learning curve:

1. Learn about Docker and containers.
2. Run your monolith in a container.
3. Develop and run your microservices in a container.
4. Once the migration is done and you've mastered containers, learn about Kubernetes.
5. As the work progresses, you can scale up the microservices and gradually move traffic to them.

Figure 24: Containerizing your monolith is a way of standardizing deployment, and it is an excellent first step in learning Kubernetes.

4.4.6 Warm up to changes

It takes time to get used to microservices, so it's best to start small and warm up to the new paradigm. Leave enough time for everyone to get in the proper mindset, upskill, and learn from mistakes without the pressure of a deadline.

During these first tentative steps you'll learn a lot about distributed computing. You'll have to deal with cloud SLA, set up SLAs for your own services, implement monitoring and alerts, define channels for cross-team communication, and decide on a deployment strategy.

Pick something easy to start with, like edge services that have little overlap with the rest of the monolith. You could, for instance, build an authentication microservice and route login requests as a first step.

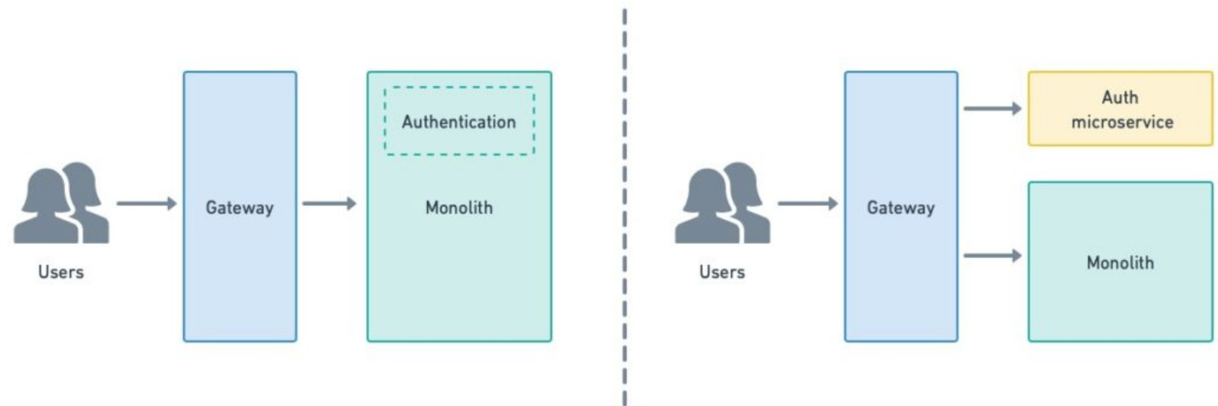

Figure 25: Pick something easy to start, like a simple edge service.

4.4.7 Use feature flags

Feature flags are a software technique for changing the functionality of a system without having to re-deploy it. You can use feature flags to turn on and off portions of the monolith as they are migrated, experiment with alternative configurations, or run A/B testing.

An typical workflow for a feature-flag-enabled migration is:

1. Identify a piece of the monolith's functionality to migrate to a microservice.
2. Wrap the functionality with a feature flag. Re-deploy the monolith.
3. Build and deploy the microservice.
4. Test the microservice.
5. Once satisfied, disable the feature on the monolith by switching the feature off.
6. Repeat until the migration is complete.

Because feature flags allow us to deploy inactive code to production and toggle it at any time, we can decouple feature releases from actual deployment. This gives developers an enormous degree of flexibility and control.

4.4.8 Modularize the monolith

If your monolith is a tangle of code, you may very well end up with a tangle of *distributed* code once the migration is done. Like tidying up a house before a total renovation, modularizing the monolith is a necessary preparation step.

The modular monolith is a software development pattern consisting of vertically-stacked modules which are independent and interchangeable. The opposite of a modular monolith is the classic N-tier, or layered, monolith.

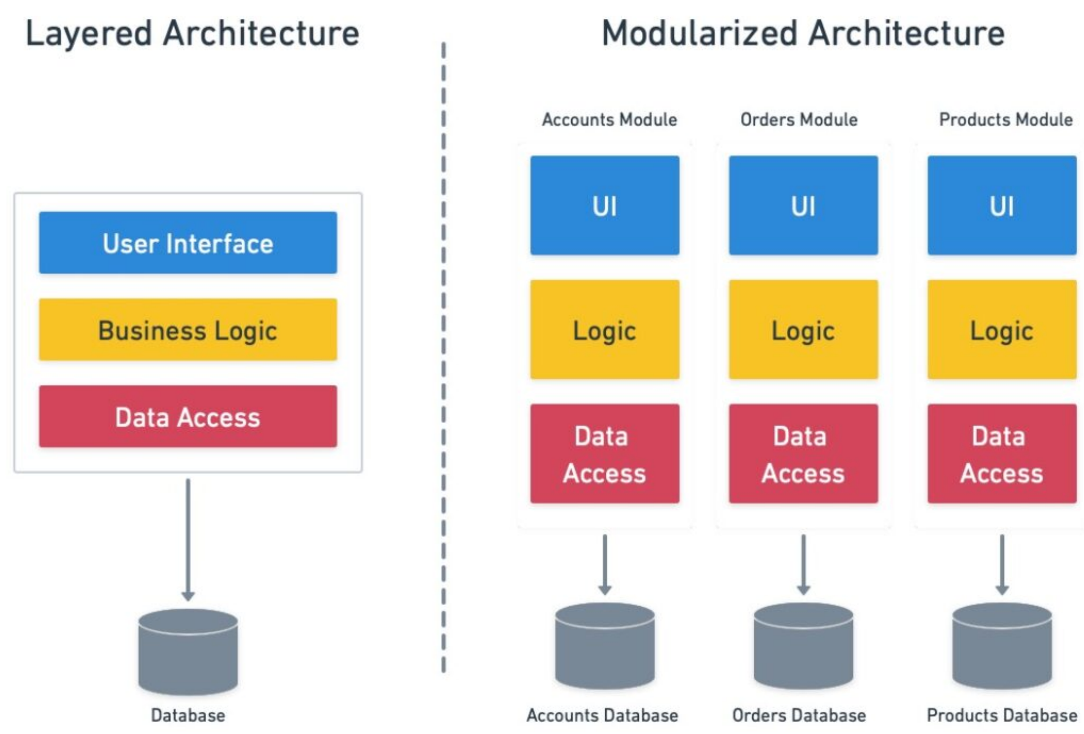

Figure 26: Layered vs. modular monolith architectures.

Layered monoliths are hard to disentangle – code tends to have too many dependencies (sometimes circular), making changes difficult to implement.

A modular monolith is the next best thing to microservices and a stepping stone towards them. The rule is that modules can only communicate over public APIs and everything is private by default. As a result, the code is less intertwined, relationships are easy to identify, and dependencies are clear-cut.

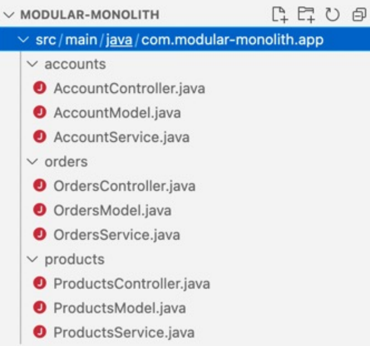

Figure 27: This Java monolith has been split into independent modules.

Two patterns can help you refactor a monolith: the Strangler Fig and the Anticorruption Layer.

4.4.9 The strangler fig pattern

In the Strangler Fig pattern, we refactor the monolith from the edge to the center. We chew at the edges, progressively rewriting isolated functionality until the monolith is entirely redone.

Calls between modules are routed through the "strangler façade," which emulates and interprets the legacy code's inputs and outputs. Bit by bit, modules are created and slowly replace the old monolith.

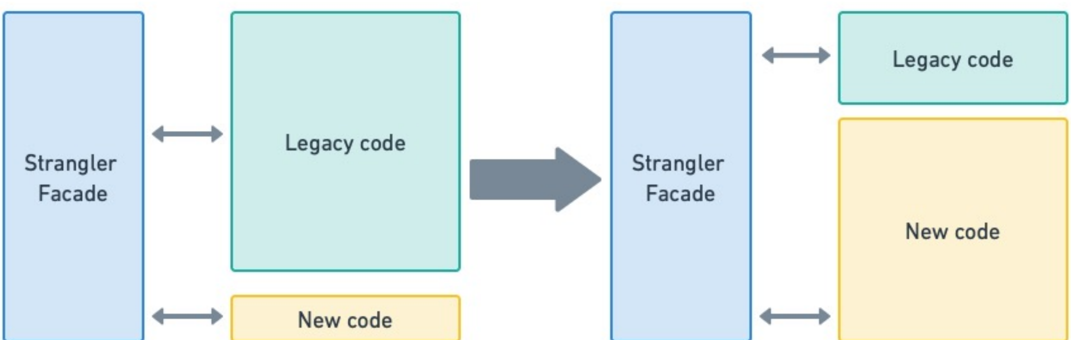

Figure 28: The monolith is modularized one piece at a time. Eventually, the old monolith is gone and is replaced by a new one.

4.4.10 The anticorruption layer pattern

You will find that, in some cases, changes in one module propagate into others as you refactor the monolith. To combat this, you can create a translation layer between rapidly-changing modules. This anticorruption layer prevents changes in one module from impacting the rest.

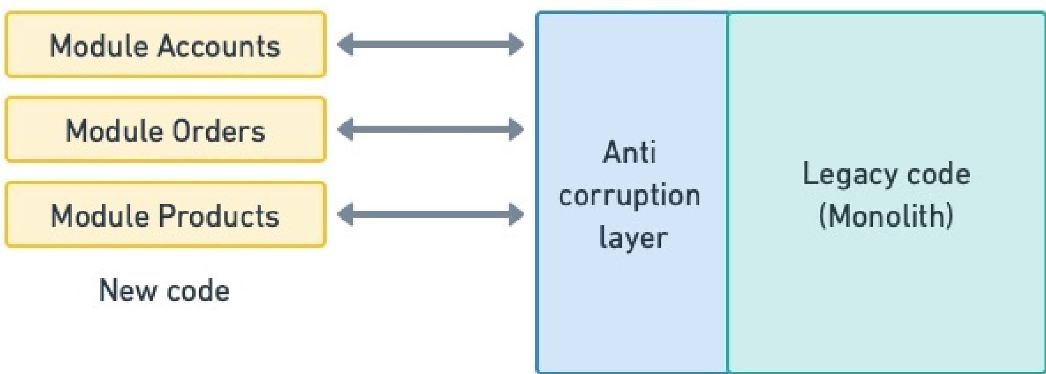

Figure 29: The anticorruption layer prevents changes from propagating by translating calls between modules and the monolith.

4.4.11 Decouple the data

The superpower microservices give you is the ability to deploy any microservice at any time with little or no coordination with other microservices. This is why data coupling must be avoided at all costs, as it creates dependencies between services. Each microservice must have a private and independent database.

It can be shocking to realize that you have to denormalize the monolith's shared database into (often redundant) smaller databases. But data locality is what will ultimately let microservices work autonomously.

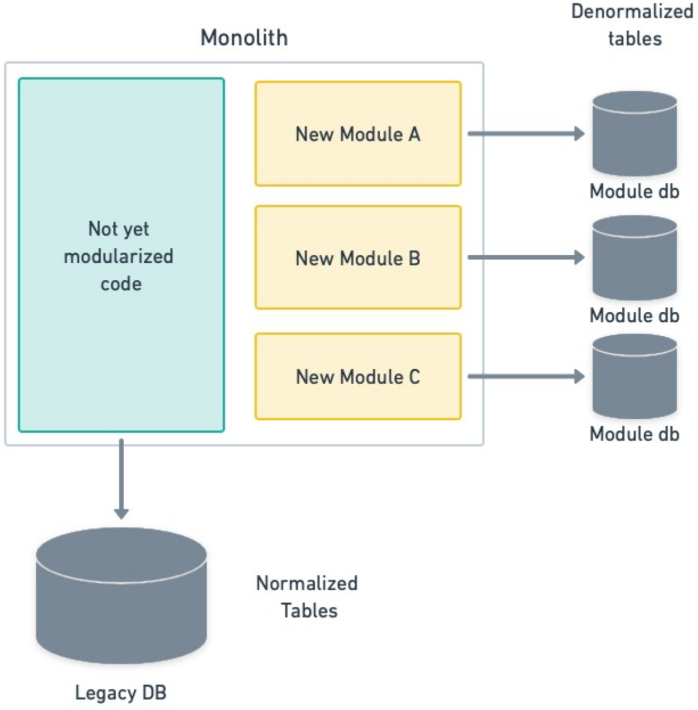

Figure 30: Decoupling data into separate and independent databases.

After decoupling, you'll have to install mechanisms to keep the old and new data in sync while the transition is in progress. You can, for example, set up a data-mirroring service or change the code, so transactions are written to both sets of databases.

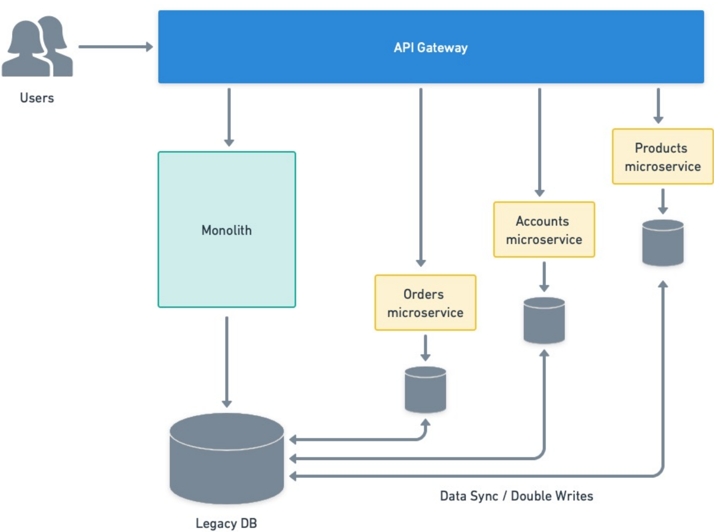

Figure 31: Use data duplication to keep tables in sync during development.

4.4.12 Add observability

The new system must be faster, more performant, and more scalable than the old one. Otherwise, why bother with microservices?

You need a baseline to compare the old with the new. Before starting the migration, ensure you have good metrics and logs available. It may be a good idea to install some centralized logging and monitoring service, since it's a key component for the observability of any microservice application.

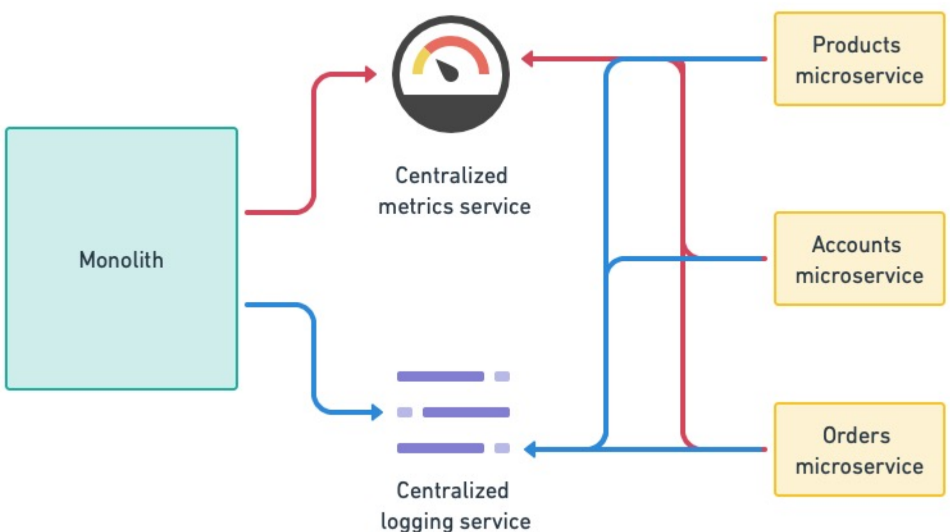

Figure 32: Metrics are used to compare the performance.

4.5 Techniques for testing microservices

How do we test a microservice application? How do we test when third party services are involved and network disruptions are a possibility? The microservice architecture is a paradigm shift so profound that we must reconsider conventional testing techniques. Microservices differ from the classic monolithic structure in many ways:

- **Distributed**: microservices are deployed across multiple servers, potentially across geographical locations, adding latency and exposing the application to network disruptions. Tests that rely on the network can fail due to no fault of the code, interrupting the CI/CD pipelines and blocking development.
- **Autonomous**: as long as they don't break API compatibility, development teams are free to deploy their microservices at any time.
- **Increased test area**: since each microservice exposes at least a few API endpoints, there are many more testable surfaces to cover.
- **Polyglot**: development teams can choose the best language for their microservice. In a big system, it's unlikely that we'll find a single test framework that works for all components.

- **Production is a moving target**: because microservices are independently-deployable and built by autonomous teams, extra checks and boundaries are required to assure they will all still function correctly together when deployed.

All these characteristics force us to think of new testing strategies.

4.6 The testing pyramid for microservices

The testing pyramid is a planning tool for automated software testing. In its traditional form, the pyramid uses three types of tests:

- Unit tests
- Integration tests
- End-to-end tests.

The microservice pyramid adds two new types: **component** and **contract** tests.

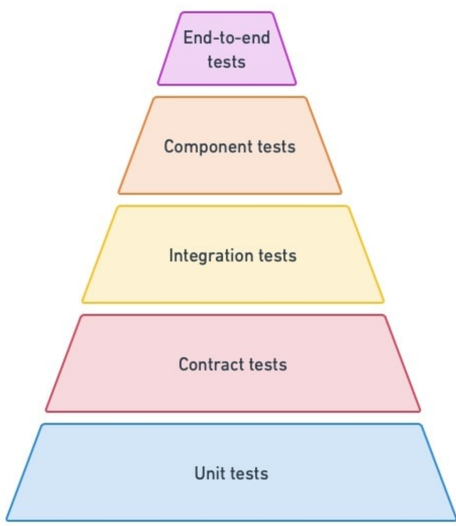

Figure 33: One version of the microservice testing pyramid.

Let's see how each pyramid layer works in further detail.

4.6.1 Unit tests for microservices

Unit tests are one of the most fine-grained — and numerous — forms of testing. A unit consists of a class, method, or function that can be tested in isolation. Unit testing is an inseparable part of development practices like Test-Driven Development or Behavior-Driven Development.

Compared to a monolith, a unit in a microservice has a much higher chance of requiring a network call to fulfill its function. When this happens, we can either let the code access the external service — accepting some latency and uncertainty — or replace the call with a test double, giving us two ways of dealing with microservice dependencies:

- **Solitary unit tests**: this should be used when we need the test result to always be deterministic. We use mocking or stubbing to isolate the code under test from external dependencies.
- **Sociable unit tests**: sociable tests are allowed to call other services. In this mode, we push the complexity of the test into the test or staging environment. Sociable tests are not deterministic, but we can be more confident in their results when they pass.

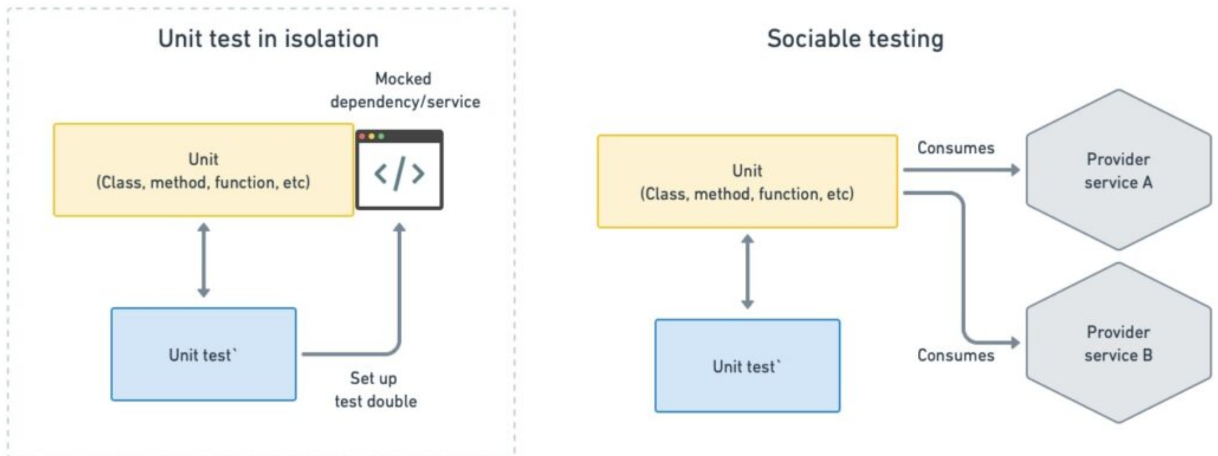

Figure 34: We can run unit tests in isolation using test doubles. Alternatively, we can allow tested code to call other microservices, in which case we're talking about sociable tests.

As you'll see, balancing confidence vs. stability will be a running theme throughout the entire chapter. Mocking makes things faster and reduces uncertainty, but the more you mock, the less you can trust the results. Sociable tests, despite their downsides, are more realistic. So, you'll likely need to strike a good balance of both types.

If you want to check examples of solitary vs sociable tests, check out this nice post from Dylan Watson at dev.to.

4.6.2 Contract testing

A contract is formed whenever two services couple via an interface. The contract specifies all the possible inputs and outputs with their data structures and side effects. The consumer and producer of the service must follow the rules stated in the contract for communication to be possible.

Contract tests ensure that microservices adhere to their contract. They do not thoroughly test a service's behavior; they only ensure that the inputs and outputs have the expected characteristics and that the service performs within acceptable time and performance limits.

Depending on the relationship between the services, contract tests can be run by the producer, the consumer, or both.

- **Consumer-side contract tests** are written and executed by the downstream team. During the test, the microservice connects to a fake or mocked version of the producer service to check if it can consume its API.
- **Producer-side contract tests** are run in the upstream service. This type of test emulates the various API requests clients can make, verifying that the producer matches the contract. Producer-side tests let the developers know when they are about to break compatibility for their consumers.

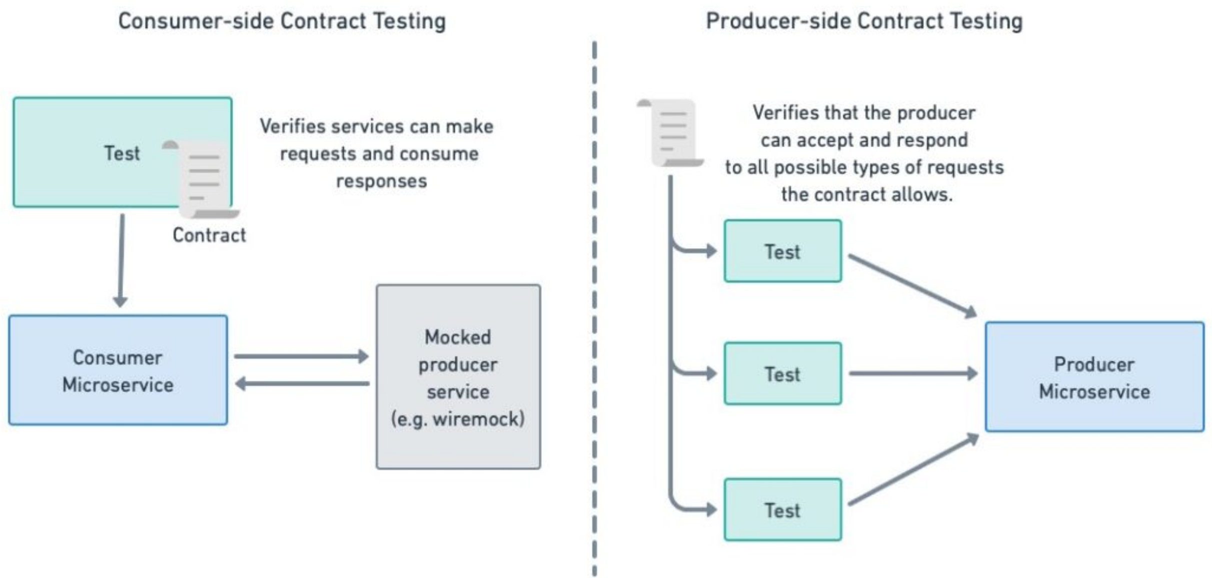

Figure 35: Contract tests can run on the upstream or downstream. Producer tests check that the service doesn't implement changes that would break depending services. Consumer tests run the consumer-side component against a mocked version of the upstream producer (not the real producer service) to verify that the consumer can make requests and consume the expected responses from the producer. We can use tools such as Wiremock to reproduce HTTP requests.

If both sides of the contract tests pass, the producers and consumers are compatible and should be able to communicate. Contract tests should always run in continuous integration to detect incompatibilities before deployment.

You can play with contract testing online in the Pact 5-minute getting started guide. Pact is a HTTP-based testing tool to write and run consumer- and producer-based contract tests.

4.6.3 Integration tests

Integration tests on microservices work slightly differently than in other architectures. The goal is to identify interface defects by making microservices interact. Unlike contract tests, where one side is always mocked, integration tests use real services.

Integration tests are not interested in evaluating behavior or business logic of a service. Instead we want to make sure that the microservices can communicate with one another and their own databases. We're looking for things like missing HTTP headers and mismatched request/response pairings. And, as a result, integration tests are typically implemented at the interface level.

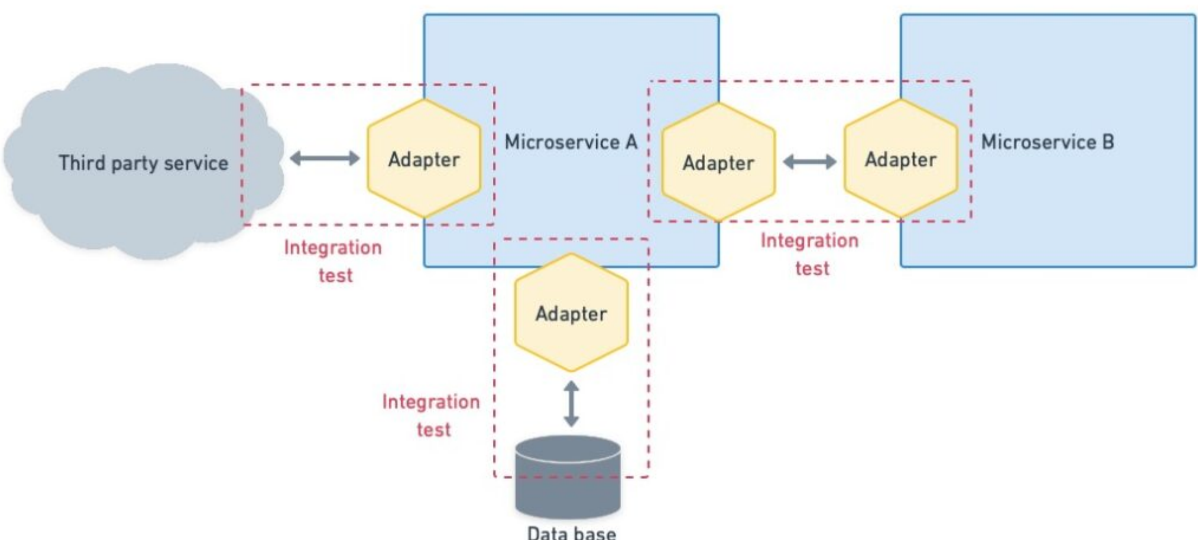

Figure 36: Using integration tests to check that the microservices can communicate with other services, databases, and third party endpoints.

Check out Vitaly Baum's post on stubbing microservices to see integration code tests in action.

4.6.4 Component tests for microservices

A component is a microservice or set of microservices that accomplishes a role within the larger system.

Component testing is a type of acceptance testing in which we examine the component's behavior in isolation by substituting services with simulated resources or mocking.

Component tests are more thorough than integration tests because they travel happy and unhappy paths — for instance, how the component responds to simulated network outages or malformed requests. We want to know if the component meets the needs of its consumer, much like we do in acceptance or end-to-end testing.

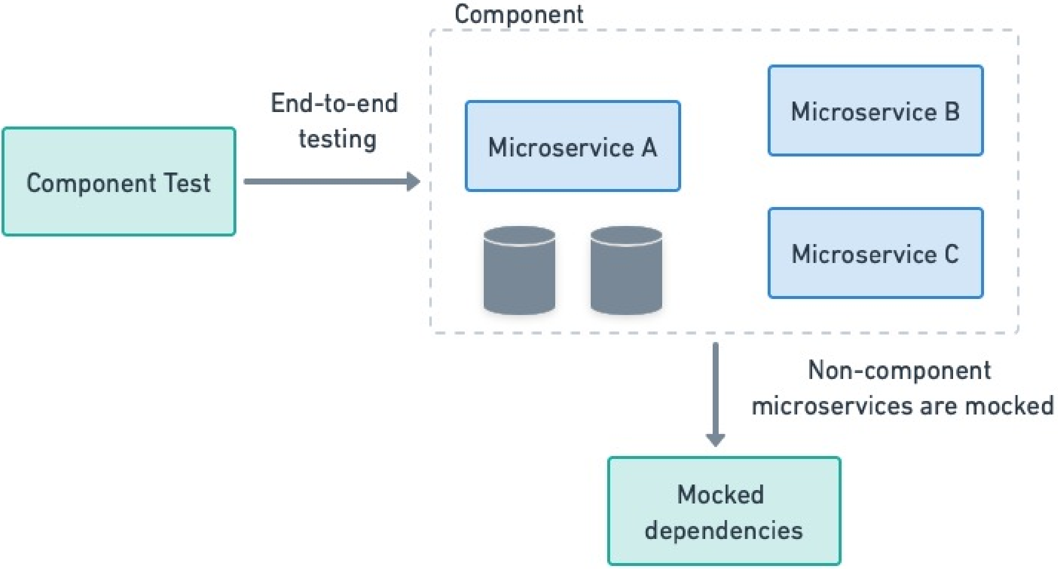

Figure 37: Component testing performs end-to-end testing to a group of microservices. Services outside the scope of the component are mocked.

There are two ways of performing component testing: in-process and out-of-process.

4.6.5 In-process component testing

In this subclass of component testing, the test runner exists in the same thread or process as the microservice. We start the microservice in an “offline test mode”, where all its dependencies are mocked, allowing us to run the test without the network.

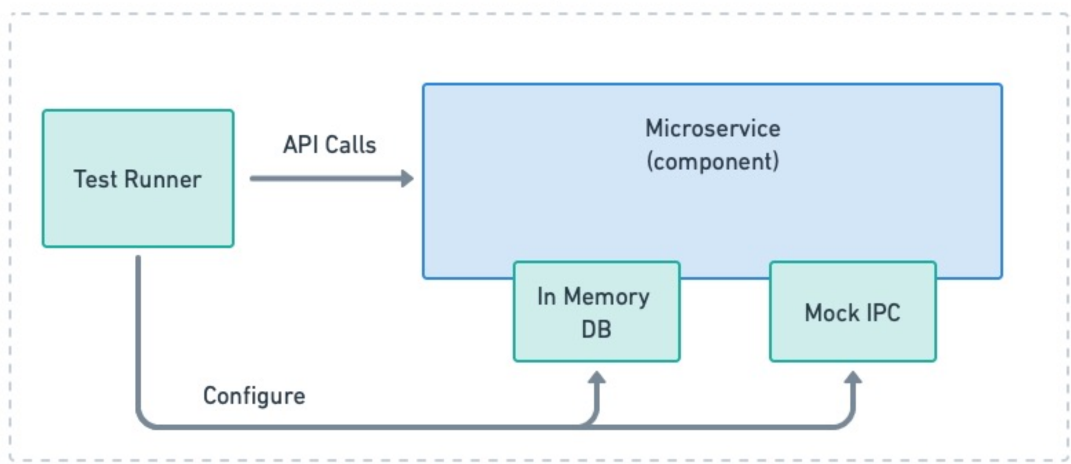

Figure 38: Component test running in the same process as the microservice. The test injects a mocked service in the adapter to simulate interactions with other components.

In-process testing only works when the component is a single microservice. On a first glance, component tests look very similar to end-to-end or acceptance tests. The only difference is that component tests pick one part of the system (the component) and isolate it from the rest. The component is thoroughly tested to verify that it performs the functions its users or consumers need.

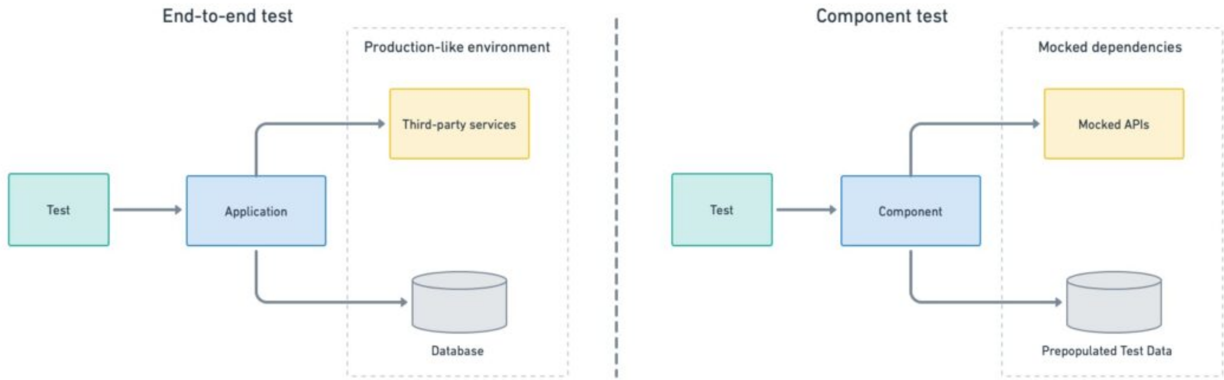

Figure 39: Component and end-to-end testing may look similar. But the difference is that end-to-end tests the complete system (all the microservices) in a production-like environment, whereas a component does it on an isolated piece of the whole system. Both types of tests check the behavior of the system from the user (or consumer) perspective, following the journeys a user would perform.

We can write component tests with any language or framework, but the most popular ones are probably Cucumber and Capybara.

4.6.6 Out-of-process component testing

Out-of-process tests are appropriate for components of any size, including those made up of many microservices. In this type of testing, the component is deployed — unaltered — in a test environment where all external dependencies are mocked or stubbed out.

Figure 40: In this type of component tests the complexity is pushed out into the test environment, which should replicate the rest of the system.

To round out the concept of contract testing you may explore example code for contract testing on Java Spring. Also, if you are a Java developer, this post has code samples for testing Java microservices at every level.

4.6.7 End-to-end testing in microservices

So far, we have tested the system piecemeal. Unit tests were used to test parts of a microservice, contract tests covered API compatibility, integration tests checked network calls, and component tests were used to verify a subsystem's behavior. Only at the very top of the automated testing pyramid do we test the entire system.

End-to-end (E2E) testing ensures that the system meets users needs and achieves their business objectives. The E2E suite should cover all the microservices in the application using the same interfaces that users would–often with a combination of UI and API tests.

The application should run in an environment as close as possible to production. Ideally, the test environment would include all the third-party services that the application usually needs, but sometimes, these can be mocked to cut costs or prevent abuse.

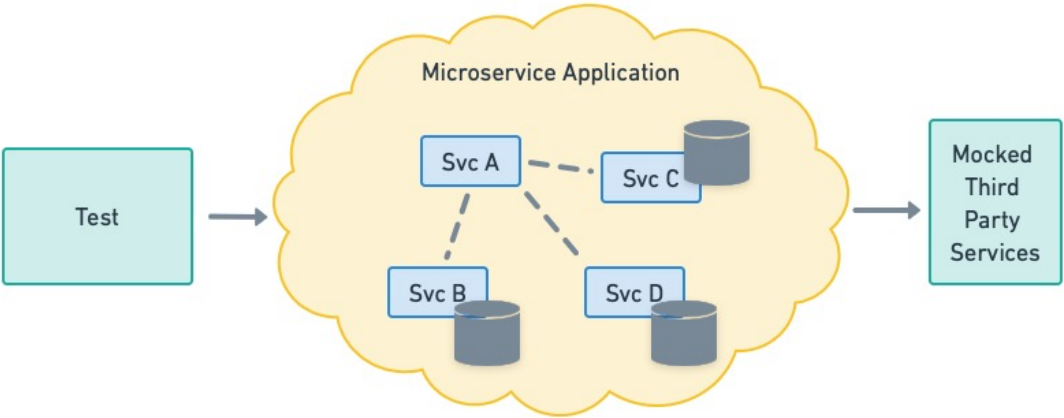

Figure 41: End-to-end are automated tests that simulate user interaction. Only external third-party services might be mocked.

As depicted by the testing pyramid, E2E tests are the least numerous, which is good because they are usually the hardest to run and maintain. As long as we focus on the user's journeys and their needs, we can extract a lot of value with only a few E2E tests.

4.7 Changing the testing paradigm

A different paradigm calls for a change in strategies. Testing in a microservice architecture is more important than ever, but we need to adjust our techniques to fit the new development model. The system is no longer managed by a single team. Instead, every microservice owner must do their part to ensure that the application works as a whole.

Some organizations might decide that unit, contract, and component tests are enough. Others, not content without end-to-end and integration testing, may choose to establish a QA team to facilitate cross-team test coverage.

Chapter 5 — Running Microservices

A microservice application is a group of distributed programs that communicate over networks, occasionally interfacing with third-party services and databases. Microservices, by their networked nature, provide more points of failure than a traditional monolith. As a result of this, we need a different, broader approach running them.

5.1 Deploying microservices

Processes or containers? Run on my servers or use the cloud? Do I need Kubernetes? When it comes to the microservice architecture, there is such an abundance of options and it is hard to know which is best.

As we'll see, the perfect place to host a microservice application is largely determined by its size and scaling requirements. So, let's go over the five most common ways we can deploy microservices.

5.2 Ways to deploy microservices

Microservice applications can run in many ways, each with different tradeoffs and cost structures. What works for small applications spanning a few services will likely not suffice for large-scale systems.

From simple to complex, here are the **five ways of running microservices**:

1. **Single machine, multiple processes**: buy or rent a server and run the microservices as processes.
2. **Multiple machines, multiple processes**: the obvious next step is adding more servers and distributing the load, offering more scalability and availability.
3. **Containers**: packaging the microservices inside a container makes it easier to deploy and run along with other services. It's also the first step towards Kubernetes.
4. **Orchestrator**: orchestrators such as Kubernetes or Nomad are complete platforms designed to run thousands of containers simultaneously.
5. **Serverless**: serverless allows us to forget about processes, containers, and servers, and run code directly in the cloud.

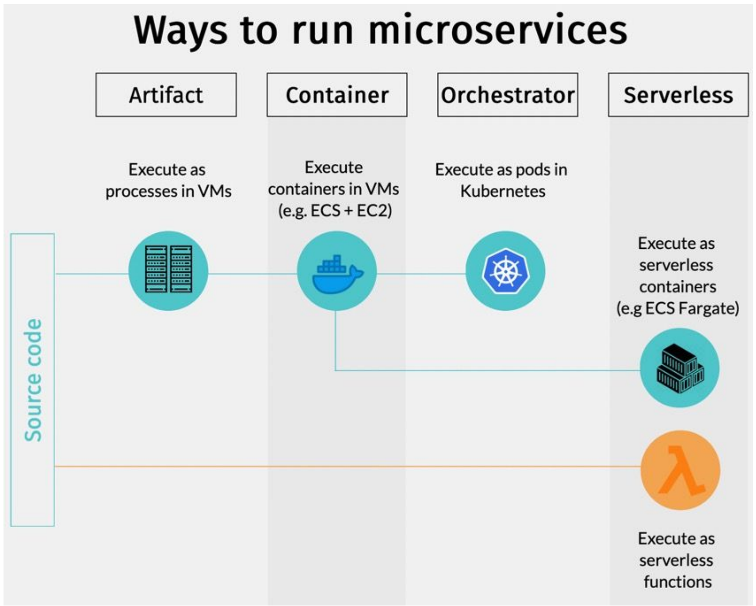

Figure 42: Two paths ahead: one goes from process, to containers, and ultimately, to Kubernetes. The other goes the serverless route.

Let's see each one in more detail.

5.2.1 Single machine, multiple processes

At the most basic level, we can run a microservice application as multiple processes on a single machine. Each service listens to a different port and communicates over a loopback interface.

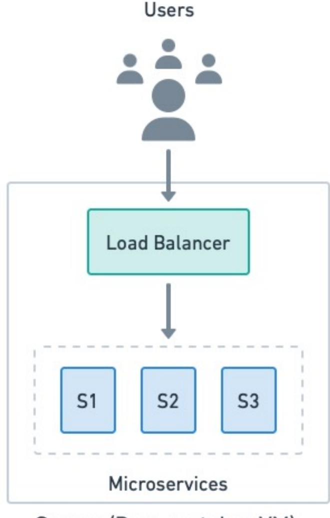

Figure 43: The most basic form of microservice deployment uses a single machine. The application is a group of processes coupled with load balancing.

This simple approach has some clear benefits:

- **Lightweight**: there is no overhead as it's just processes running on a server.
- **Convenience**: it's a great way to experience microservices without the learning curve that other tools have.
- **Easy troubleshooting**: everything is in the same place, so finding a problem or reverting to a working configuration in case of trouble is very straightforward, if you have continuous delivery in place.
- **Fixed billing**: we know how much we'll have to pay each month.

The DIY approach works best for small applications with only a few microservices. Past that, it falls short because:

- **No scalability**: once you max out the resources of the server, that's it.
- **Single point of failure**: if the server goes down, the application goes down with it.
- **Fragile deployment**: we need custom deployment and monitoring scripts to ensure that services are installed and running correctly.
- **No resource limits**: any microservice process can consume any amount of CPU or memory, potentially starving other services and leaving the application in a degraded state.

Continuous integration (CI) for this option will follow the same pattern: build and test the artifact in the CI pipeline, then deploy with continuous deployment.

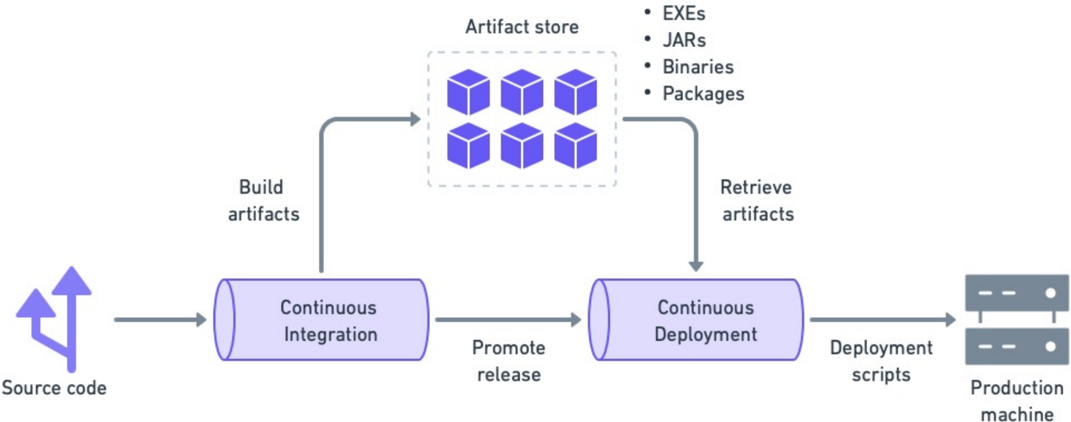

Figure 44: Custom scripts are required to deploy the executables built in the CI pipeline.

This is the best option to learn the basics of microservices. You can run a small-scale microservice application to get familiarized. A single server will take you far until you need to expand, at which time you can upgrade to the next option.

5.2.2 Multiple machines and processes

This option is essentially an upgrade of option 1. When the application exceeds the capacity of a server, we can scale up (upgrade the server) or scale sideways (add more servers). In the case of microservices, horizontally scaling into two or more machines makes more sense since we get improved availability as a bonus. And, once we have a distributed setup, we can always scale up by upgrading servers.

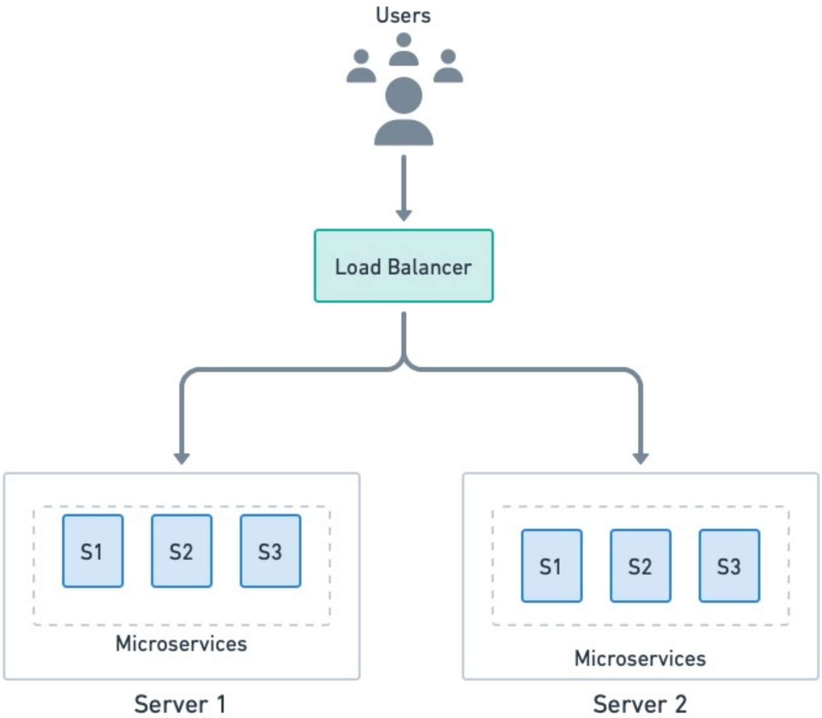

Figure 45: The load balancer still is a single point of failure. To avoid this, multiple balancers can run in parallel.

Horizontal scaling is not without its problems, however. Going past one machine poses a few critical points that make troubleshooting much more complex and typical problems that come with using the microservice architecture emerge.

- How do we correlate log files distributed among many servers?
- How do we collect sensible metrics?
- How do we handle upgrades and downtime?
- How do we handle spikes and drops in traffic?

These are all problems inherent to distributed computing, and are something that you will experience (and have to deal with) as soon as more than one machine is involved.

This option is excellent if you have a few spare machines and want to improve your application's availability. As long as you keep things simple, with services that are more or less uniform (same language, similar frameworks), you will be fine. Once you pass a certain complexity threshold, you'll need containers to provide more flexibility.

5.2.3 Deploy microservices with containers

While running microservices directly as processes is very efficient, it comes at a cost.

- The server must be meticulously maintained with the necessary dependencies and tools.

- A runaway process can consume all the memory or CPU.
- Deploying and monitoring the microservices is a brittle process.

All these shortcomings can be mitigated with containers. Containers are packages that contain everything a program needs to run. A container image is a self-contained unit that can run on any server without having to install any dependencies or tools first (other than the container runtime itself).

Containers provide just enough virtualization to run software in isolation. With them, we get the following benefits:

- **Isolation**: contained processes are isolated from one another and the OS. Each container has a private filesystem, so dependency conflicts are impossible (as long as you are not abusing volumes).
- **Concurrency**: we can run multiple instances of the same container image without conflicts.
- **Less overhead**: since there is no need to boot an entire OS, containers are much more lightweight than VMs.
- **No-install deployments**: installing a container is just a matter of downloading and running the image. There is no installation step required.
- **Resource control**: we can put CPU and memory limits on containers so they don't destabilize the server.

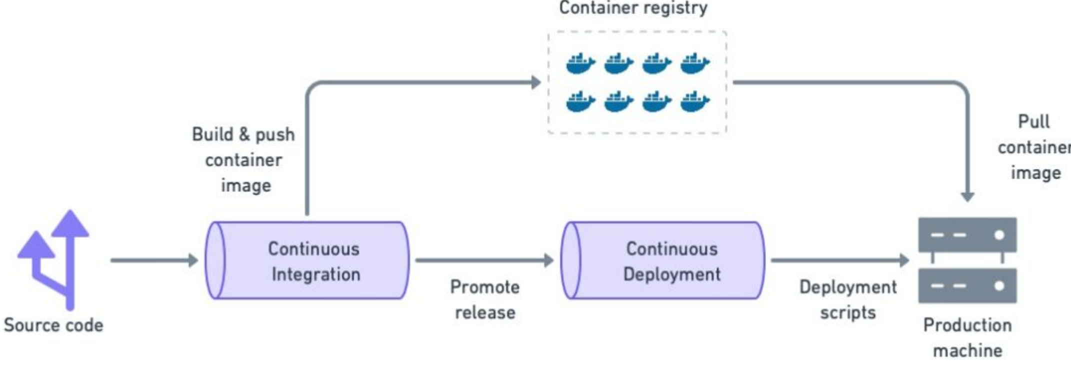

Figure 46: Containerized workloads require an image build stage on the CI/CD.

To learn more about containers, check these posts:

- Dockerizing a Node.js Web Application
- Dockerizing a Python Django Web Application
- How To Deploy a Go Web Application with Docker
- Dockerizing a Ruby on Rails Application

We can run containers in two ways: directly on servers or via a managed service.

5.2.4 Containers on servers

This approach replaces processes with containers since they give us greater flexibility and control. As with option 2, we can distribute the load across any number of machines.

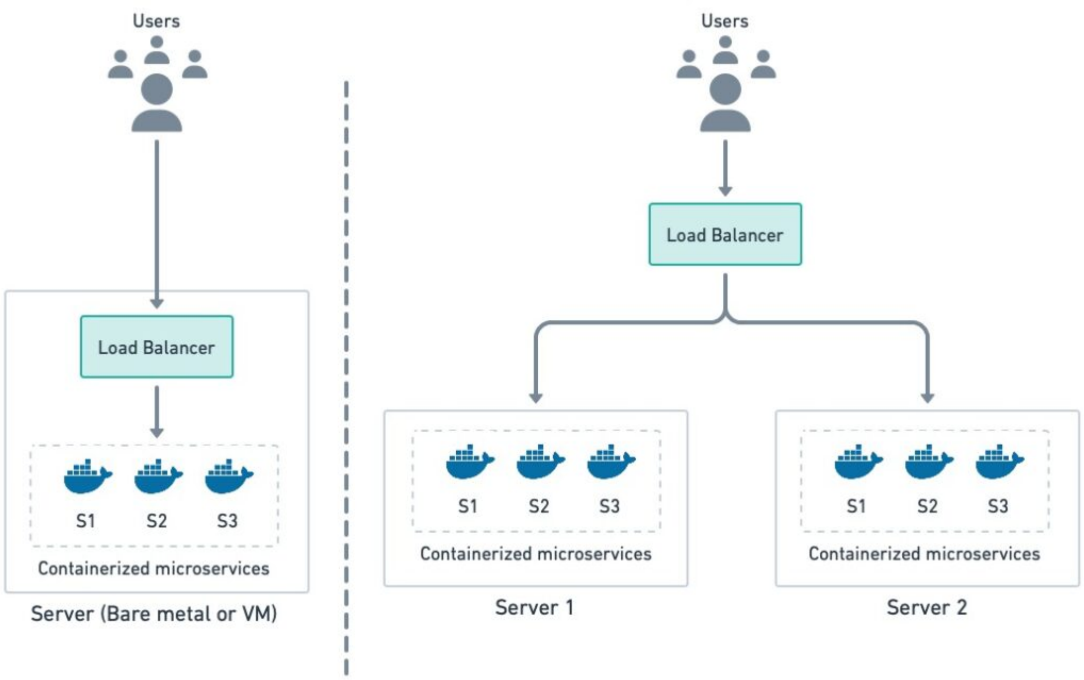

Figure 47: Wrapping microservices processes in containers make them more portable and flexible.

5.2.5 Serverless containers

All the options described up to this point were based on servers. But software companies are not in the business of managing servers — servers that must be configured, patched, and upgraded — they are in the business of solving problems with code. So, it shouldn't be surprising that many companies prefer to avoid servers whenever possible.

Containers-as-a-Service offerings such as AWS Fargate and Heroku (which unfortunately discontinued its free plan) make it possible to run containerized applications without having to deal with servers. We only need to build the container image and point it to the cloud provider, which will take care of the rest: provision up virtual machines, and download , start and monitor images. These managed services typically include a built-in load balancer, which is one less thing to worry about.

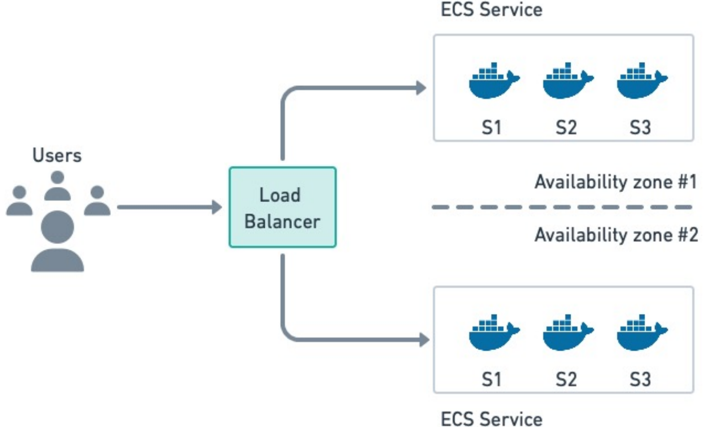

Figure 48: Elastic Container Service (ECS) with Fargate allows us to run containers without having to rent servers. They are maintained by the cloud provider.

Here are some of the benefits a managed container service has:

- **No servers**: there is no need to maintain or patch servers.
- **Easy deployment**: just build a container image and tell the service to use it.
- **Autoscaling**: the cloud provider can provide more capacity when demand spikes or stop all containers when there is no traffic.

Before jumping in, however, you have to be aware of a few significant downsides:

- **Vendor lock-in**: this is the big one. Moving away from a managed service is always challenging, as the cloud vendor provides and controls most of the infrastructure.
- **Limited resources**: managed services impose CPU and memory limits that cannot be avoided.
- **Less control**: we don't have the same level of control we get with other options. You're out of luck if you need functionality that is not provided by the managed service.

Either container option will suit small to medium-sized microservice applications. If you're comfortable with your vendor, a managed container service is easier, as it takes care of a lot of the details for you.

For large-scale deployments, needless to say, both options will fall short. Once you get to a certain size, you're more likely to have team members that have experience with (or willingness to learn about) tools such as Kubernetes, which completely change the way containers are managed.

5.2.6 Orchestrators

Orchestrators are platforms specialized in distributing container workloads over a group of servers. The most well-known orchestrator is Kubernetes, a Google-created open-source

project maintained by the Cloud Native Computing Foundation.

Orchestrators provide, in addition to container management, extensive network features like routing, security, load balancing, and centralized logs — everything you may need to run a microservice application.

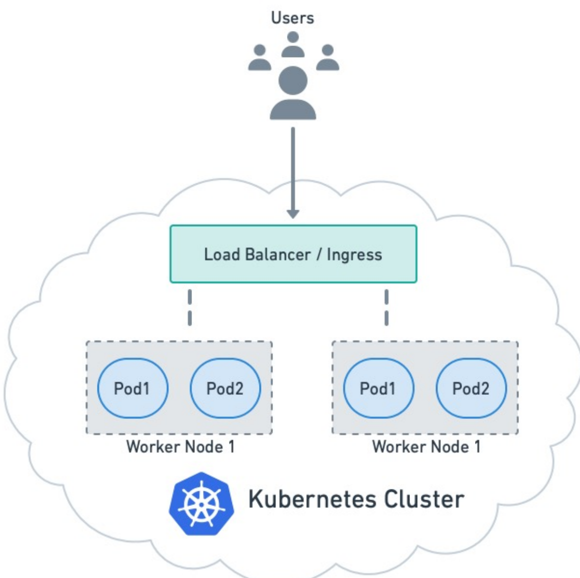

Figure 49: Kubernetes uses pods as the scheduling unit. A pod is a group of one or more containers that share a network address.

With Kubernetes, we step away from custom deployment scripts. Instead, we codify the desired state with a manifest and let the cluster take care of the rest.

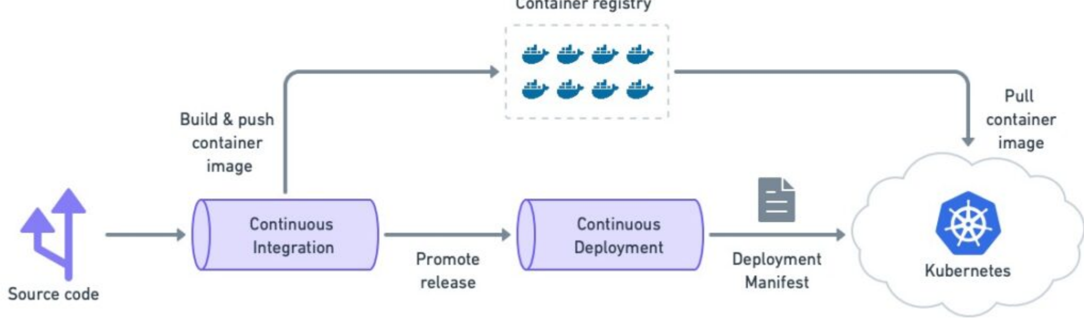

Figure 50: The continuous deployment pipeline sends a manifest to the cluster, which takes the steps required to fulfill it.

Kubernetes is supported by all cloud providers and is the *de facto* platform for microservice deployment. As such, you might think this is the absolute best way to run microservices. For many companies, this is true, but they're also a few things to keep in mind:

- **Complexity**: orchestrators are known for their steep learning curve. It's not uncommon to shoot oneself in the foot if not cautious. For simple applications, an orchestrator is overkill.
- **Administrative burden**: maintaining a Kubernetes installation requires significant expertise. Fortunately, every decent cloud vendor offers managed clusters that take away all the administration work.
- **Skillset**: Kubernetes development requires a specialized skillset. It can take weeks to understand all the moving parts and learn how to troubleshoot a failed deployment. Transitioning into Kubernetes can be slow and decrease productivity until the team is familiar with the tools.

Check out deploying applications with Kubernetes in these tutorials:

- A Step-by-Step Guide to Continuous Deployment on Kubernetes
- CI/CD for Microservices on DigitalOcean Kubernetes
- Kubernetes vs. Docker: Understanding Containers in 2022
- Continuous Blue-Green Deployments With Kubernetes

Kubernetes is the most popular option for companies making heavy use of containers. If that's you, choosing an orchestrator might be the only way forward. Before making the jump, however, be aware that a recent survey revealed that the greatest challenge for most companies when migrating to Kubernetes is finding skilled engineers. So if you're worried about finding skilled developers, the next option might be your best bet.

5.2.7 Deploy microservices as serverless functions

Serverless functions deviate from everything else we've discussed so far. Instead of servers, processes, or containers, we use the cloud to simply run code on demand. Serverless offerings like AWS Lambda and Google Cloud Functions handle all the infrastructure details required for scalable and highly-available services, leaving us free to focus on coding.

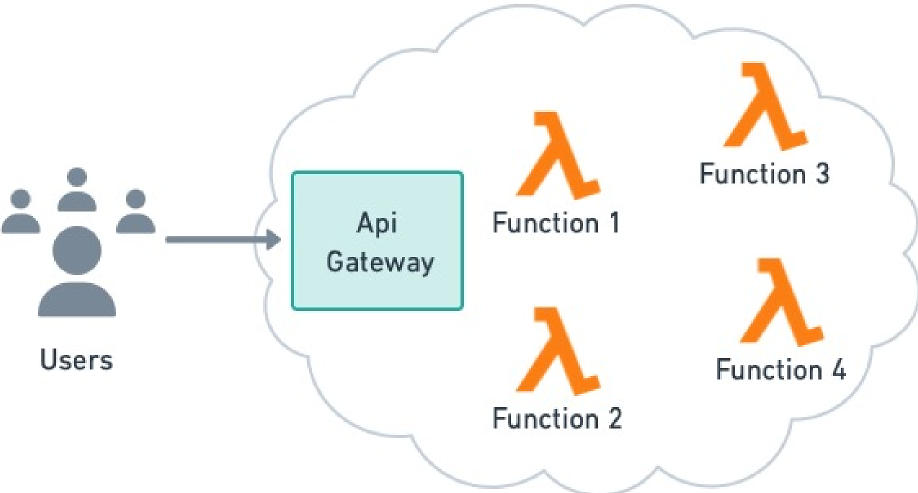

Figure 51: Serverless functions scale automatically and have per-usage billing.

It's an entirely different paradigm with different pros and cons. On the plus side, we get:

- **Ease of use**: we can deploy functions on the fly without compiling or building container images, which is great for trying things out and prototyping.
- **Easy to scale**: you get (basically) infinite scalability. The cloud will provide enough resources to match demand.
- **Pay per use**: you pay based on usage. If there is no demand, there's no charge.

The downsides, nevertheless, can be considerable, making serverless unsuitable for some types of microservices:

- **Vendor lock-in**: as with managed containers, you're buying into the provider's ecosystem. Migrating away from a vendor can be demanding.
- **Cold starts**: infrequently-used functions might take a long time to start. This happens because the cloud provider spins down the resources attached to unused functions.
- **Limited resources**: each function has a memory and time limit–they cannot be long-running processes.
- **Limited runtimes**: only a few languages and frameworks are supported. You might be forced to use a language that you're not comfortable with.

Imprevisible bills: since the cost is usage-based, it's hard to predict the size of the invoice at the end of the month. A usage spike can result in a nasty surprise.

Learn more about serverless below:

- AWS Serverless With Monorepos
- A CI/CD Pipeline for Serverless Cloudflare Workers

Serverless provides a hands-off solution for scalability. Compared with Kubernetes, it doesn't give you as much control, but it's easier to work with as you don't need specialized skills

for serverless. Serverless is an excellent option for small companies that are rapidly growing, provided they can live with its downsides and limitations.

5.3 Which method is best to deploy microservices?

The best way to run a microservice application is determined by many factors. A single server using containers (or processes) is a fantastic starting point for experimenting or testing prototypes.

If the application is mature and spans many services, you will require something more robust such as managed containers or serverless, and perhaps Kubernetes later on as your application grows.

Nothing prevents you from mixing and matching different options. In fact, most companies use a mix of bare-metal servers, VMs, and Kubernetes. A combination of solutions like running the core services on Kubernetes, a few legacy services in a VM, and reserving serverless for a few strategic functions could be the best way of taking advantage of the cloud at every turn.

5.4 Release management for microservices

Imagine a microservices application consisting of dozens of continuously-deployed autonomous services. Each of the application's constellation of services has its own repository, with a different versioning scheme and a different team continually shipping new versions. How can I tell the (whole) application's version? Being that the change history is scattered among dozens of repositories, what's the most efficient approach to keeping track of changes? And how do we manage application releases?

Any team using microservice architecture, including ours at Semaphore, must deal with these questions.

5.4.1 A common approach: one microservice, one repository

The most common way to start out with a microservice architecture application is to use the *multirepo* approach:

1. Apply Domain-Driven Design to plan how to break up the monolith into services.
2. A separate repository is created for each microservice (where the "multi" in multirepo comes from).
3. Each repository has an independent CI/CD pipeline to continuously deploy the microservice to production.

Let's call this fine-grained form of continuous deployment a *micro-deployment* for lack of a better term. With micro-deployments, the microservice versions are bumped and deployed independently, with little need for integration testing.

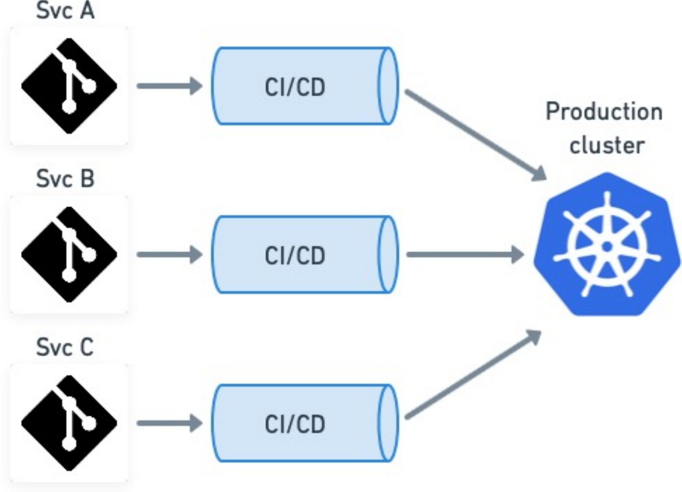

Figure 52: Each microservice has a separate CI/CD pipeline.

Micro-deployment is a side effect of organizing the code into multirepos. For reference, this is how we currently deploy microservices for Semaphore CI/CD.

5.5 Maintaining multiple microservices releases

Continuous micro-deployments are ideal for hosted apps like Netflix or Semaphore CI/CD, where users or customers are unaware of (or uninterested in) individual microservice versions running behind the scenes.

Things, however, are very different for someone running the same application on-premise. Continuous deployments don't work in this scenario. We're back to release schedules, only in this case, a release consists of a bundle of microservices pinned at specific versions.

Of course, a private Airbnb doesn't make sense, but a private CI/CD platform does. For instance, you can run a fully-functional version of Semaphore CI/CD behind your firewall with Semaphore On-Premise.

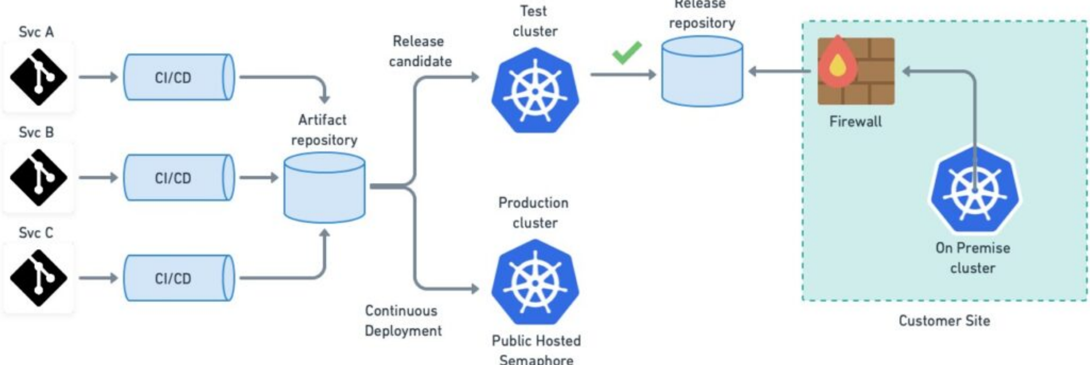

Figure 53: Micro-deployments to the hosted version of the application combined with releases for the on-premise instances of the product.

The steps needed to release an application organized into multirepos usually go like this:

1. In each repo, tag the versions of microservices that will go into the release.
2. For each microservice, build a Docker image and map the microservice version to the image tag.
3. Test the release candidate in a separate test environment. This usually involves a mix of integration testing, acceptance testing, and perhaps some manual testing.
4. Go over every repository and compile a list of changes for the release changelog before updating the documentation.
5. Identify hotfixes required for older releases.
6. Publish the release.

Considering that an application can consist of dozens of microservices (and repositories), it's easy to see how releasing this way could entail a lot of repeated admin overhead.

5.6 Managing microservices releases with monorepos

As we've seen, multirepos are better suited for continuous deployment than for periodical, non-continuous releases. So, let's see what happens on the other end of the spectrum; when we gather all the microservices into a shared repository. This is the monorepo approach, which companies like Google, Airbnb, and Uber have been using for years.

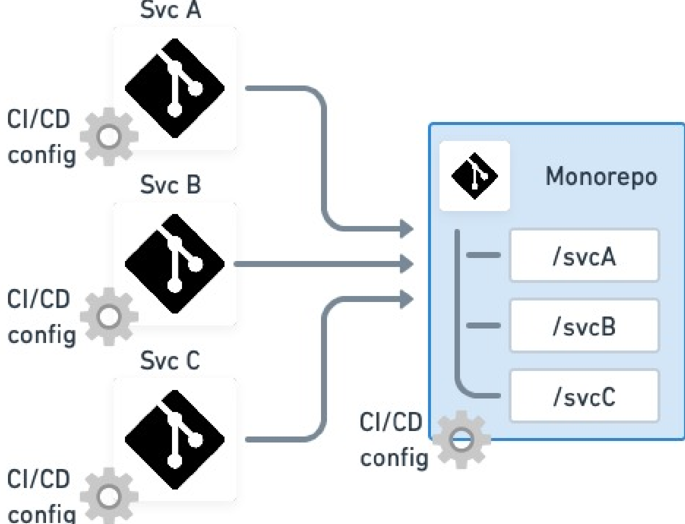

Figure 54: A monorepo contains all the microservices and a unified CI/CD deployment pipeline.

The monorepo strategy makes microservices feel more like a monolith, but in a good way:

- Creating a release is as simple as creating branches and using tags.
- A single CI/CD process standardizes testing and deployment.
- Integration and acceptance testing are a lot easier to implement.
- A single Git history is much clearer to understand, simplifying the process of writing a changelog and updating documentation.

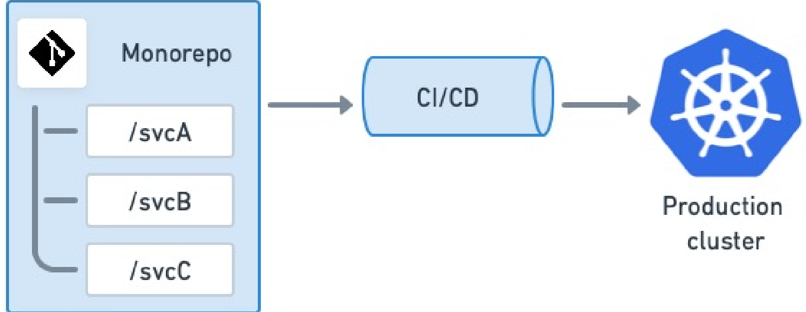

Figure 55: One CI/CD to rule them all

As always, changing the paradigm involves some tradeoffs:

- Because all changes are committed in one place, the CI server is under more strain. We can deal with this by using change-based execution or a monorepo-aware tool like Bazel or Pants.

- Git has no built-in code protection features. So, if trust is a factor, we should use a feature like Bitbucket or GitHub CODEOWNERS.
- Finding errors in the CI build can feel overwhelming when the test suite spans many separate services. Features like test reports can help you identify and analyze problems at a glance.
- A monorepo CI/CD configuration can have a lot of repetitive parts. We can use environment variables or parametrized pipelines to reduce boilerplate.

5.7 Never too far away from safety

Up to this point, we've only focused on the application's releasability, but there is another factor that might give monorepos an edge.

Version control not only allows us to collaborate, share knowledge, keep track of the code, and manage changes, it also provides the ability to recover when something breaks. As long as we have access to every change in the project, we can always go back.

Multirepos, however, cannot offer a complete picture. There is no record of the relationships between the microservices, i.e. there is no snapshot of the individual service versions running in production at any given time. As a result, diagnosing integration issues can be time-consuming, and there could be instances when fixing a problem by rolling back microservices is impossible.

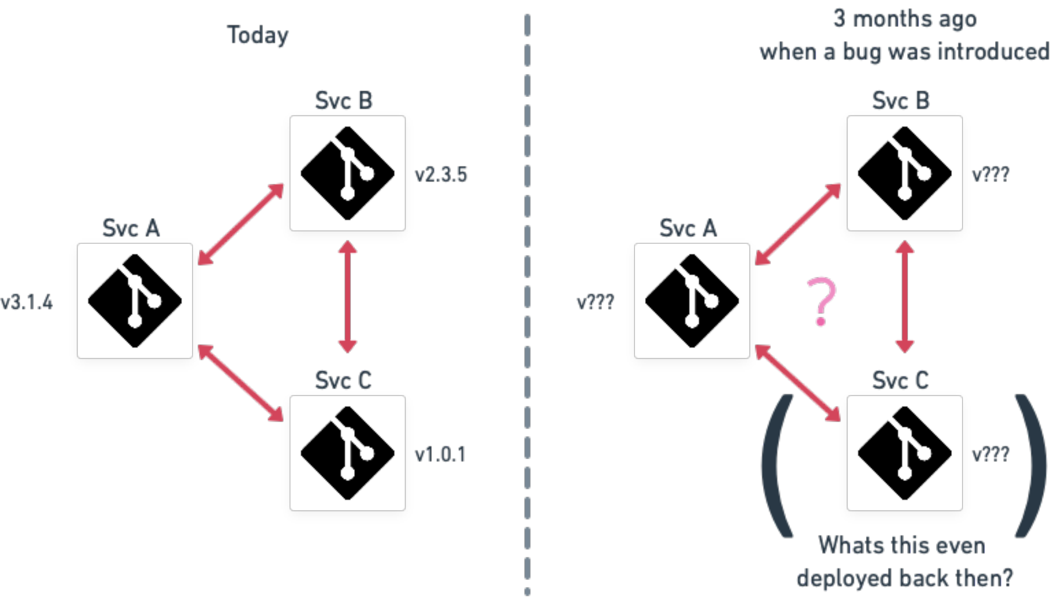

Figure 56: Multirepos make it challenging to find the root cause of a failure. It's difficult to find the "last working microservice configuration".

Monorepos don't suffer from this. A monorepo captures the complete snapshot of the system.

We have all the details needed to return to any point in the project's history. So, we'll always be able to find an appropriate place to retreat to when there's a problem.

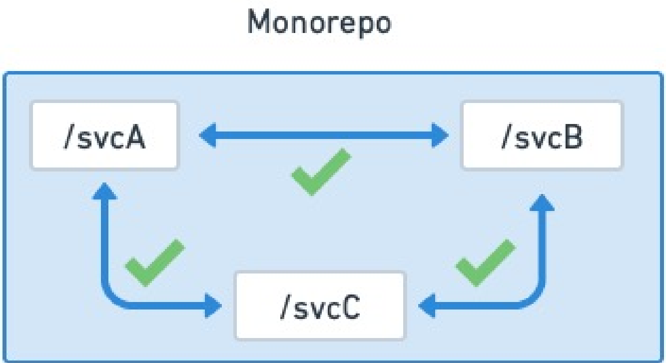

Figure 57: A monorepo has all the microservice relationship details needed to go back to any point in the project's history.

5.8 When in doubt, try monorepos

One pipeline and one repo per microservice? Or one shared repo and a global pipeline for everything? There is no single best answer that will fit every scenario. If your microservices are loosely coupled, either a multirepo or a monorepo will work perfectly fine. Multirepos require more work but provide more autonomy. However, if your services are somewhat coupled, it's best to make that relationship explicit by using a monorepo. So, when in doubt, a monorepo can be a safer bet, provided you can live with the tradeoffs.

Continuous micro-deployments have worked well for us at Semaphore, but Semaphore On-Premise is forcing us to adjust. While the final solution is still a matter of some debate, it is almost certain that it will involve migrating the core microservices to a monorepo.

Parting Words

Hopefully, by now you have a much better grasp of microservices, what they are, what they aren't, and what a migration from a monolith would entail. This may be the end of the handbook but certainly not the end of the road. We wish you the best of luck on your microservice journey!

6.1 Share This Book With The World

Please share this book with your colleagues, friends, and anyone who you think might benefit from it.

Share the book online

6.2 Tell Us What You Think

We would absolutely love to hear your feedback. What did you get out of reading this book? How easy/hard was it to follow? Is there something that you'd like to see in a new edition?

This book is open source and available at *https://github.com/semaphoreci/book-microservices*.

- Send comments and feedback, ask questions, and report problems by opening a new issue.
- Contribute to the quality of this book by submitting pull requests for improvements to explanations, code snippets, etc.
- Write to us privately at learn@semaphoreci.com.

6.3 About Semaphore

Semaphore https://semaphoreci.com helps developers continuously build, test and deploy code at the push of a button. It provides the fastest, enterprise-grade CI/CD pipelines as a serverless service. Trusted by thousands of organizations around the globe, Semaphore can help your team move faster too.